Making Searchable Symmetric Encryption Schemes Smaller and Faster

Debrup Chakraborty[1], Avishek Majumder[2], and Subhabrata Samajder[3]

[1] Cryptology and Security Research Unit, Indian Statistical Institute, Kolkata, India.
debrupchakraborty@gmail.com
[2] School of Computer Science, UPES, Dehradun, India. avishek.majumder1991@gmail.com
[3] Institute of Advancing Intelligence (IAI), TCG CREST, Kolkata, India and Academy of Scientific and Innovative Research (AcSIR), Ghaziabad, India.
subhabrata.samajder@gmail.com

Abstract. Searchable Symmetric Encryption (SSE) has emerged as a promising tool for facilitating efficient query processing over encrypted data stored in un-trusted cloud servers. Several techniques have been adopted to enhance the efficiency and security of SSE schemes. The query processing costs, storage costs and communication costs of any SSE are directly related to the size of the encrypted index that is stored in the server. To our knowledge, there is no work directed towards minimizing the index size. In this paper we introduce a novel technique to directly reduce the index size of any SSE. Our proposed technique generically transforms any secure single keyword SSE into an equivalently functional and secure version with reduced storage requirements, resulting in faster search and reduced communication overhead. Our technique involves in arranging the set of document identifiers $\mathsf{db}(w)$ related to a keyword w in leaf nodes of a complete binary tree and eventually obtaining a succinct representation of the set $\mathsf{db}(w)$. This small representation of $\mathsf{db}(w)$ leads to smaller index sizes. We do an extensive theoretical analysis of our scheme and prove its correctness. In addition, our comprehensive experimental analysis validates the effectiveness of our scheme on real and simulated data and shows that it can be deployed in practical situations.

Keywords: Outsourced Storage, Searchable Symmetric Encryption, Binary Tree, Tree Cover

1 Introduction

Delegating an organization's or individual's storage requirements to a third-party server has become a common practice. Storage as a Service, commonly referred to as cloud storage, is a model in which providers offer storage solutions to clients. These solutions are advantageous as they alleviate users from costs associated with storage-related hardware, software, and maintenance. While outsourcing storage requirements offers convenience and cost-effectiveness, it also raises significant security concerns. Among these concerns, the foremost is the confidentiality of stored data. Specifically, the potential exposure of users' data to the un-trusted server.

Historically, the challenge of maintaining data confidentiality has been addressed through encryption. Secure encryption algorithms ensure that the resulting ciphertext is indistinguishable from random strings, rendering it practically unreadable for the server. However, this solution presents a dilemma, as users also require the ability to query and update the delegated data. Encrypting the stored data with a traditional encryption algorithm would hinder the server's capability to execute queries or updates on behalf of the client.

Searchable Symmetric Encryption (SSE) schemes, as introduced by Song et al. [29] and Curtmola et al. [16], are specialized encryption schemes that allow users to store encrypted data on untrustworthy cloud servers while enabling search queries and updates on the encrypted data. SSE schemes have now emerged as a viable strategy for encrypting data destined for storage on un-trusted third-party servers.

An SSE scheme abstracts a database as a set of documents denoted as $\mathcal{D}$. Each document $\mathsf{d}_i \in \mathcal{D}$ is associated with a unique identifier id_i. These documents are viewed

as collections of keywords from a predefined finite set, denoted as $\mathcal{W}$. For each keyword, $w \in \mathcal{W}$, $\mathsf{db}(w)$ represents the set of identifiers of those documents containing the keyword w. For a $w \in \mathcal{W}$, let $t_w = \mathsf{db}(w) \times \{w\} = \{(\mathsf{id}, w) : \mathsf{id} \in \mathsf{db}(w)\}$. Generally, the input to an SSE scheme consists of the set of tuples $S = \bigcup_{w \in \mathcal{W}} t_w$. An SSE encrypts this set, comprising keyword-identifier pairs, using a specialized structure often referred to as an "*encrypted multi-map*" [13] (EMM) or an inverted index. This encrypted multi-map is stored on the server and facilitates both searching and updating the encrypted database. To initiate a keyword search, the client provides a *search token* for the encrypted multi-map to the server. For updates (such as additions and deletions to/from the database), the client supplies an *update token* to the server, enabling modifications to the encrypted multi-map. Subsequently, these search and update tokens are used by the server to transmit encrypted search results to the client and update the database as needed. SSE schemes which support only search operations are called Static SSE schemes, whereas the schemes which support both search and updates are called Dynamic SSE (DSSE) schemes. Note, in the context of an SSE the encryption of the tuples in S is only considered. The encryption of the documents in $\mathcal{D}$ does not fall under the purview of an SSE scheme, which can be separately encrypted and stored in the server.

1.1 Our Contributions

In general, the efficiency of an SSE scheme is measured by the time it takes for search and update. Most schemes are optimized to get better search and update times maintaining adequate security guarantees.

Optimizing the size of the encrypted database has not yet received adequate attention in the literature as it is always assumed that the server is computationally powerful and has sufficient storage. But, optimizing the storage size has an immediate effect on search time and communication overhead. If the input set of tuples for an SSE is S, then the size of the encrypted database in any existing SSE scheme, to the best of our knowledge, is at least $|S|$. In fact in most schemes (particularly in dynamic SSE schemes) the storage overhead is much more than $|S|$. Similarly, the size of the result of a query w is at least $|\mathsf{db}(w)|$, and in most schemes, the size of the search result which is to be communicated to the client is much more than $|\mathsf{db}(w)|$. In most Dynamic SSEs the size of the encrypted database and the query result increases with the number of updates performed on the database.

Recently, some effort has been made to restrict the size of the search result to at most $\mathcal{O}(\mathsf{db}(w))$ for any given keyword w [30,12,17,11]. To the best of our knowledge, no SSE scheme has considered reducing the storage size beyond $|S|$ and the result size beyond $|\mathsf{db}(w)|$. In this study, we specifically address the following questions.

1. Can we build a functional and secure SSE whose storage size is smaller than $|S|$ on average?
2. Can a search query for a keyword w on average be answered with a result size smaller than $|\mathsf{db}(w)|$?

We answer both these questions in the affirmative. In order to achieve this, we introduce a generic method for transforming any single keyword SSE scheme into an equivalent secure SSE scheme where the size of the encrypted database required to store for searching is much smaller than in the original SSE scheme. This results in a more compact representation of both the encrypted database and the set $\mathsf{db}(w)$ compared to the original SSE scheme. Note, that a reduction of the size of the encrypted database and $\mathsf{db}(w)$ also results in a reduction of search time and communication costs.

The Basic Technique: Given $\mathsf{db}(w)$ for any keyword w, we convert $\mathsf{db}(w)$ to a new set c_w, which we call as the *"cover of the keyword"* w. We ensure that, on average, $|c_w|$ is smaller than $|\mathsf{db}(w)|$, while retaining the ability to fully recover $\mathsf{db}(w)$ from c_w. We provide $S' = \bigcup_{w \in \mathcal{W}} c_w \times \{w\}$ as an input to any standard secure SSE. Note that S' is on average less than the size of $S = \bigcup_{w \in \mathcal{W}} \mathsf{db}(w) \times \{w\}$, which is the input to the original SSE. This generic transformation of $\mathsf{db}(w)$ to its cover c_w produces a considerable reduction in both the storage size and result size, resulting in shorter search time and reduced communication size.

The heart of our technique lies in representing $\mathsf{db}(w)$ for each w as a full binary tree. For each keyword $w \in \mathcal{W}$, we construct a virtual full binary tree where each leaf node is associated with an identifier id_i. We label the leaf nodes with $+$ if the identifier corresponding to it is present in $\mathsf{db}(w)$ and label it with $-$ otherwise. This tree with the labels in the leaf nodes uniquely represents $\mathsf{db}(w)$. We devise a scheme by which this tree can be represented uniquely by a small set of nodes of the tree, and we call this set as the cover of the tree. We show how we can use this representation on any existing SSE scheme to achieve considerable savings in storage size and the size of query results. Furthermore, we also propose several efficient algorithms to generate covers of a labeled tree for different scenarios. We do in-depth combinatorial analysis of these algorithms and prove them to be optimal in our setting.

Experimental Validation: We validate our efficiency cla-ims with extensive experimentation on both synthetic and real data. We use synthetic data to validate our theoretical claims regarding the average reduction of database sizes by exhaustively computing covers for all possible configurations of the set $\mathsf{db}(w)$. Our theoretical bound on the cover sizes closely matches our experimental results. We also report the results of our schemes when applied to the Enron Email database [18]. Our proposed scheme achieves a significant reduction (between 35% to 60%) in the size of the encrypted database over a base SSE scheme. We also simulate the dynamic setting in SSE using a synthetic database, and we demonstrate a significant advantage of our scheme in the dynamic setting as well. Furthermore, we provide results on extra overhead incurred by adopting our technique and show that the overhead of our scheme is fairly reasonable in a practical context. The basic codes and data used for our experiment are publicly available at [1].

1.2 Related Work.

The notion of SSE was initially introduced by Song et al. [29], and was first formally defined by Curtmola et al. [16]. While the definition provided in [16] was for static databases, the first *dynamic scheme*, accommodating database updates, was formalized and proposed by Kamara et al. [23]. This was followed by several Dynamic SSE schemes [9,6,7,19,15,12,31]. Efforts such as [21,8] began studying SSE security through inherent leakages in the schemes. The authors of [34] improved upon the previous line of work and showed that the notion of security originally proposed in [23] was insufficient by suggesting some realistic attacks. To mitigate such attacks, two notions of privacy, namely, *"forward privacy"* and *"backward privacy"*, were first mentioned in [30]. The first formal definitions and construction of forward privacy and backward privacy was due to Bost [6], and Bost et.al. [7] respectively. Since then, many efficient SSE schemes meeting both forward and backward privacy with different efficiency and security have been proposed [36,15,12,31]. Secure SSE schemes supporting more complex queries like range queries [35,25], conjunctive queries [10,32,24,28,33], wildcard sear-ch [20,14] etc. have also been reported.

Few efforts [30,17,11] were made by researchers to propose SSE schemes with *optimal search complexity*. Consider a database containing N many keyword-identifier pairs. Let i_w be the number of additions, and d_w be the number of deletions for a keyword w. Then the total number of updates u_w for the keyword w is given by $u_w = i_w + d_w$. Let $n_w = i_w - d_w$ denote the actual number of keyword identifier pairs for the keyword w currently present in the database. Then, an SSE with optimal search refers to those schemes that achieve a $\mathcal{O}(n_w)$ search complexity. In other words, for an SSE with optimal search, the search time for a keyword w should depend only on the number of documents currently present in the database that contains w, and not on total updates performed for the keyword. Most SSE schemes [36,15,12,31] do not achieve this complexity as they treat deletion also as an addition with a specific tag and thus the search time also depends on the number of deletions.

The tree structure that we propose for representing the set $\mathsf{db}(w)$ is motivated by the idea of *subset difference scheme* used in the context of broadcast encryption [26]. The subset-difference (SD) techniques were first introduced by Naor et al. in [26]. It was primarily used for key pre-distribution in symmetric key broadcast encryption schemes. The main goal of this technique was to reduce the broadcast header length, which is the extra amount of data that needs to be broadcasted for proper decryption. The scheme in [26] was later generalized in [4,5]. Though the basic idea of our scheme is derived from subset difference scheme but the details of our method and its application in the context of SSE is totally new.

1.3 Structure of the Paper

We begin with the definitions of SSE in Section 2. Sections 3 and 4 develops the basic tools required by us. Particularly, Section 3 defines the important concept of tree cover whereas Section 4 is devoted to the description of cover generation and reconstruction algorithms and proving their correctness. Section 4 also contains the calculations related to computing the average cover size of a configuration of a binary tree. Sections 3 and 4 make no reference to SSE schemes and thus can be of independent interest; they describe and solve some combinatorial problems related to binary trees, and these solutions are later used to design SSEs. Sections 5 and 6 are devoted to designing SSEs with tree covers where we describe in detail how to convert a given SSE to a new SSE using the idea of tree covers. In Section 7, we discuss some existing SSEs and the concrete improvements that can be obtained if these schemes are used as a base SSE in our protocol. In Section 8 we define security of SSE schemes and describe formally the security of our proposed schemes. In Section 9, we report the experimental results, and finally, in Section 10, we conclude this article and point to some future directions of work.

2 Searchable Symmetric Encryption

General Notations: For a finite set X, $|X|$ denotes the cardinality of X. Let X, Y be two finite sets then, $X \times Y$ denotes the Cartesian products of X and Y. For a non-negative integer n, $[n]$ denotes the set $\{1, 2, \ldots, n\}$, and for non-negative integers i, j, $i < j$, $[i, j]$ represents the set $\{i, i+1, ..., j\}$. The set $\{0,1\}^*$ represents the set of all binary strings, including the empty string, and for a positive integer n, $\{0,1\}^n$ denotes the set of all n bit strings. For $x, y \in \{0,1\}^*$, $x||y$ denote the concatenation of the strings x and y. Let $X_1, X_2, \ldots, X_n$ be finite sets, then $\mathsf{minCard}(X_1, \ldots, X_n)$ gives a nonempty set of minimum cardinality among the sets $X_1, X_2, \ldots, X_n$.

Searchable Symmetric Encryption (SSE) scheme is a protocol between a client and a server. It enables a client to store a collection of documents, a database, in an encrypted

manner on the server such that the server learns "almost nothing" about the client's data and queries. Each document in the database is identified by a unique identifier and has some keywords associated with it. The protocol must support search operations on the keywords, i.e., a search for a keyword w must return all the documents (identifiers) associated with it. In addition, an SSE may support updates, i.e., insertion of new keyword-identifier pair, or deletion of keywords from existing document identifiers. An SSE scheme that does not support updates is called a Static SSE scheme and the ones which support updates are called Dynamic SSE schemes.

Let $\mathcal{D} = \{\mathsf{d}_1, \ldots, \mathsf{d}_n\}$ denote the collection of all documents that the client wants to store in the server. Let $\mathcal{W}$ denote the set of all possible keywords. Each document $\mathsf{d}_i \in \mathcal{D}$, corresponds to a set of keywords $\mathsf{w}_i \subseteq \mathcal{W}$. Define $\mathsf{W} = \bigcup_{i=1}^{n} \mathsf{w}_i \subseteq \mathcal{W}$ to be the collection of all keywords present in the database DB.

Let $\mathsf{ID} : \mathcal{D} \to \{0,1\}^\lambda$, for a fixed constant λ (in general considered as the security parameter), be an injective map. We call $\mathsf{id}_i = \mathsf{ID}(\mathsf{d}_i)$ as the identifier of the document d_i. The set of all identifiers is denoted with $\mathcal{I} = \{\mathsf{id}_1, \ldots, \mathsf{id}_n\}$. It is customary to abstract out a database as a collection of identifiers and their associated keywords, and we would only be concerned about storing keyword-identifier (id, w) pairs and retrieving those. Note that, in general, the mapping ID is known only to the client and thus the client can associate documents with the identifiers securely.

For $i \in [n]$, let $S_i = \{(\mathsf{id}_i, w) : w \in \mathsf{w}_i\}$. Then a database DB can be defined as a set of tuples, as $\mathsf{DB} = \bigcup_{i \in [n]} S_i$. For each keyword $w \in \mathsf{W}$, define the set of identifiers containing the keyword w as $\mathsf{db}(w) = \{\mathsf{id} : (\mathsf{id}, w) \in \mathsf{DB}\}$.

Thus, if $U_w = \mathsf{db}(w) \times \{w\}$, then a database DB can also be viewed as

$$\mathsf{DB} = \bigcup_{w \in \mathsf{W}} U_w.$$

For ease of explanation, throughout this paper, we write identifier id containing keyword w to mean that the keyword w is present in a document d with $\mathsf{id} = \mathsf{ID}(\mathsf{d})$.

With this abstraction of a database, an SSE scheme Σ is defined as follows. It closely resembles the definitions of Bost et al. [7] and Chatterjee et al. [15].

Definition 1 (SSE). *An SSE scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *is a tuple of one algorithm and two protocols between the client (*C*) and the server (*S*) defined as follows.*

- $(k, \sigma_\mathsf{C}, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB})$*: The* Setup *is a probabilistic polynomial time (*PPT*) algorithm run by the client that takes as input the security parameter* 1^λ *and the initial database* DB*. It outputs a key* k*, the client's state* σ_C *and the encrypted database* EDB*. The client stores the key* k*,* σ_C*, and the encrypted database* EDB *is sent to the server.*
- $\mathsf{Update}(k, \sigma_\mathsf{C}, q; \mathsf{utoken}, \mathsf{EDB})$*: The* Update *protocol is executed between the client and the server. It comprises the following two algorithms.*
  - $(\sigma_\mathsf{C}, \mathsf{utoken}) \leftarrow \mathsf{Update}_\mathsf{C}(k, \sigma_\mathsf{C}, q)$*: This (possibly probabilistic) algorithm is run by the client. On input the key* k*, client's state* σ_C *and an update query* $q = (\mathsf{op}, \mathsf{in})$*, the algorithm outputs an update token* utoken *and client's updated state* σ_C*, where* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ *and* $\mathsf{in} = (\mathsf{id}, \mathsf{w})$*.*
  - $\mathsf{EDB} \leftarrow \mathsf{Update}_\mathsf{S}(\mathsf{utoken}, \mathsf{EDB})$*: This is a deterministic algorithm run by the server, which takes as input the encrypted database* EDB *and the update token* utoken*. It then outputs an updated encrypted database* EDB*.*
- $\mathsf{Search}(k, \sigma_\mathsf{C}, q; \mathsf{stoken}_q, \mathsf{EDB})$*: The* Search *protocol is executed between the client and the server. It comprises of the following two algorithms.*

- $(\sigma_{\mathsf{C}}, \mathsf{stoken}) \leftarrow \mathsf{Search}_{\mathsf{C}}(k, \sigma_{\mathsf{C}}, q)$*: This (possibly probabilistic) algorithm is run by the client. On input the key* k*, client's state* σ_{C} *and search query* q*, the algorithm outputs a search token* stoken *and client's updated state* σ_{C}*.*
- $\mathsf{res} \leftarrow \mathsf{Search}_{\mathsf{S}}(\mathsf{stoken}, \mathsf{EDB})$*: This is a deterministic algorithm run by the server. It takes as input the encrypted database* EDB *and the search token* stoken*. It outputs a set* res *of identifiers matching the search results for the search query* q*.*

Remark. In this work, we only consider single keyword search queries, i.e., we consider $q = w$ for search queries. For updates, we will only consider update queries of the type $q = (\mathsf{op}, \mathsf{in})$, where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, $\mathsf{in} = (\mathsf{id}, \mathsf{w})$ and $\mathsf{w} = \{w\}$. For bulk updates, the Update algorithm is executed multiple times with different (id, w) pairs, where $w \in \mathsf{w}$.

Static and Dynamic SSE Scheme. Definition 1 captures the setting of a dynamic SSE Σ. Recall that for a Static SSE scheme update is not allowed. Therefore, a Static SSE scheme can be defined as in Definition 1 but without the Update protocol. That is a Static SSE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search})$ is only a tuple of one algorithm and one protocol.

Correctness. An SSE scheme is said to be *correct* if, for any search query on w, the Search protocol returns the correct search result (i.e., $\mathsf{db}(w)$), except with negligible probability. A more formal definition of correctness is given in [9], but in our context, we will not require this formalism.

3 Binary Trees and Tree Covers

Our main object of interest is a *complete binary tree*, i.e., a binary tree where all levels are present. We fix some basic terminology first.

The *depth* of a node i of a binary tree is the number of edges in the path from i to the root. The root has a depth of 0. In a complete binary tree, there are exactly 2^d nodes at depth d, and consequently, all leaf nodes are at the same depth. The height of a node i is the number of edges in a path from i to the deepest leaf. The height of the tree is the height of the root of the tree. A complete binary tree of height h, has exactly $2^{h+1} - 1$ nodes, with 2^h leaf nodes. The structure of a complete binary tree can be specified only using its height. Let $\mathcal{T}$ be a tree and i a node of $\mathcal{T}$, then by $\mathcal{T}(i)$ we denote the subtree rooted at i. Thus, if i is the root of $\mathcal{T}$, then $\mathcal{T}(i) = \mathcal{T}$ and if i is a leaf node then $\mathcal{T}(i)$ is a tree containing the single node i. Unless specifically mentioned, by a binary tree, we will mean a complete binary tree.

For convenience, we will denote nodes of a complete binary tree by integers. The root will be denoted by zero. The nodes at depth d will be denoted by 2^d consecutive integers starting with $2^d - 1$, counting left to right. Thus, for a tree $\mathcal{T}$ of height h, the nodes of $\mathcal{T}$ would be denoted by the set of integers $\mathsf{nodes}(\mathcal{T}) = \{0, 1, \ldots, 2^{h+1} - 2\}$ and the leaf nodes would be denoted by the set $\mathsf{leaves}(\mathcal{T}) = \{2^h - 1, 2^h, \ldots, 2^{h+1} - 2\}$. For any $i \in \mathsf{nodes}(\mathcal{T})$, $\mathsf{leaves}(\mathcal{T}(i)) \subseteq \mathsf{leaves}(\mathcal{T})$ will denote the leaf nodes of the subtree rooted at i.

We will sometimes use an alternative representation of the leave nodes. Given a tree $\mathcal{T}$ of height h with $\mathsf{leaves}(\mathcal{T}) = \{2^h - 1, 2^h, \ldots, 2^{h+1} - 2\}$. Let $\phi_h : \mathsf{leaves}(\mathcal{T}) \to [1, 2^h]$, be a bijection defined as $\phi_h(i) = i - 2^h + 2$. Note that for any $i \in \mathsf{leaves}(\mathcal{T})$, if $j = \phi_h(i)$, then the leaf node i in $\mathcal{T}$ is the j^{th} leaf node from the left. As ϕ_h is a bijection, we have $\phi_h^{-1} : [1, 2^h] \to \mathsf{leaves}(\mathcal{T})$ defined as $\phi_h^{-1}(j) = j + 2^h - 2$.

Definition 2 (Configuration). *Let* $\mathcal{T}$ *be a complete binary tree of height* h *and* $\mathsf{leaves}(\mathcal{T})$ *be the set of leaf nodes of* $\mathcal{T}$*. A configuration of* $\mathcal{T}$ *is a function* $\Lambda_{\mathcal{T}} : S \to \{+, -\}$*, where*

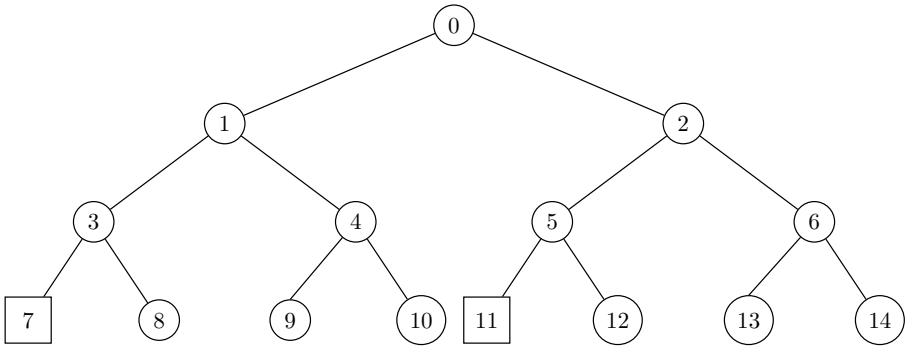


Fig. 1. The tree $\mathcal{T}$ used in Example 1.

$S \subseteq \mathsf{leaves}(\mathcal{T})$. *The configuration is complete if* $S = \mathsf{leaves}(\mathcal{T})$, *i.e., a complete configuration of* $\mathcal{T}$ *is a map* $\Lambda_{\mathcal{T}} : \mathsf{leaves}(\mathcal{T}) \to \{+, -\}$. *We will represent* $\Lambda_{\mathcal{T}}$ *by a subset of* $\mathsf{leaves}(\mathcal{T}) \times \{+, -\}$.

Thus, a configuration put labels from the set $\{+, -\}$ to the leaves of a tree. If all leaves of a tree are labelled then the configuration is called a *complete configuration*. We represent a configuration $\Lambda_{\mathcal{T}} : S \to \{+, -\}$, where $S \subseteq \mathsf{leaves}(\mathcal{T})$ by the set $L = \{(i, s) : i \in S, s = \Lambda_{\mathcal{T}}(i)\}$.

Example 1. Consider the full binary tree $\mathcal{T}$ of height 3 shown in Figure 1. The nodes of $\mathcal{T}$ are elements of the set $\mathsf{nodes}(\mathcal{T}) = \{0, 1, \ldots, 14\}$ and $\mathsf{leaves}(\mathcal{T}) = \{7, 8, \ldots, 14\}$. $L = \{(7, -), (8, +), (9, +), (10, +), (11, -), (12, +), (13, +), (14, +)\}$, is a complete configuration of $\mathcal{T}$, where the nodes 7 and 11 are labeled $-$ and all other nodes are labeled $+$. Whereas, the configuration $\{(7, -), (8, +), (9, -), (10, +), (11, -)\}$ is not complete, as the leaf nodes 12, 13 and 14 are not assigned any labels.

3.1 Cover of a Configuration

Our goal is to have a succinct representation of the configuration of a binary tree. For a tree $\mathcal{T}$ the naive representation of its configuration can be done through a subset of $\mathsf{leaves}(\mathcal{T}) \times \{+, -\}$, i.e, by listing all the leaf nodes along with their labels. We are interested in a more compact representation. Informally, a cover is a succinct representation of a configuration.

We define a cover of a configuration through a pair of algorithms *cover generation* and *cover reconstruction*.

Definition 3 (Syntax of a Tree Cover Scheme). *A tree cover scheme* $\Psi = (\mathsf{CoverGen}, \mathsf{ReConstruct})$ *is a tuple of two algorithms defined as follows.*

- $C \leftarrow \mathsf{CoverGen}(h, L)$*: On input a configuration* L *of a tree* $\mathcal{T}$ *of height* h, CoverGen *outputs a cover* $C \subset \mathsf{nodes}(\mathcal{T}) \times \{+, -\}$.
- $L \leftarrow \mathsf{ReConstruct}(h, C)$*: On input the height* h *of a tree* $\mathcal{T}$ *and a cover* $C \subset \mathsf{nodes}(\mathcal{T}) \times \{+, -\}$, ReConstruct *outputs a configuration* $L \subset \mathsf{leaves}(\mathcal{T}) \times \{+, -\}$.

Correctness: We say a tree cover scheme Ψ is correct if for any configuration L of a tree of height h,

$$L = \Psi.\mathsf{ReConstruct}(h, \Psi.\mathsf{CoverGen}(h, L)).$$

A tree cover scheme is interesting only if the covers produced by its cover generation algorithm have a size much smaller than the configuration (which it receives as input) for most configurations. The cover generation schemes described later satisfy this property.

We do not yet provide any structural characterisation of a cover of a configuration. A cover is just what is output by a cover generation algorithm, with the guarantee that there is a corresponding reconstruction algorithm which can reconstruct the configuration from which the cover was generated. This syntactic definition of a cover would be enough for us to develop SSE schemes with desirable properties, but at this point, it may be useful to see what a cover generation algorithm can possibly do.

Let us take the example of Figure 1, which represents a complete configuration $L = \{(7,-), (8,+), (9,+), (10,+), (11,-), (12,+), (13,+), (14,+)\}$, of a tree $\mathcal{T}$ of height 3. Nodes drawn in circles that is the nodes $\{8, 9, 10, 12, 13, 14\}$ are marked as $+$ and nodes drawn in squares that is $\{7, 11\}$ are marked as $-$. According to our definition so far, there can be several algorithms for cover generation which can possibly output different covers for the same tree with the same labeled leaf nodes. We focus on two such simple instances.

1. L itself can be a cover. In this case, on input L the cover generation algorithm simply outputs L as the cover and the reconstruction algorithm on input any set C, just outputs C.
2. The set $C = \{(7,-), (11,-)\}$, can be a cover. Here the cover generation algorithm on input L outputs the set $C = \{(i,-) : (i,-) \in L\}$, i.e., outputs those nodes (along with their labels) which are labeled $-$. The cover reconstruction algorithm on input C labels those nodes in $\mathsf{leaves}(\mathcal{T})$ which are not in C with $+$ and calls those labeled nodes as L_1 and outputs $C \cup L_1$. It is easy to see that this reconstruction always works when L is a complete configuration.

The above two instances are trivial covers. Note that in case (2) we already have a cover whose size is much smaller than the configuration, at least for the example that we consider. In Section 4 we systematically study much more complex cover generation algorithms which on average give covers whose sizes are smaller than the configuration. For designing Dynamic SSEs a notion of covers for a dynamic configuration would be necessary which we provide in the next subsection. The design of the SSEs presented in Sections 5 and 6 only assumes the definitions of cover generation and reconstruction algorithms. Exact instances of cover generation algorithms, as discussed in Section 4, are not required to follow the material in Sections 5 and 6.

3.2 Cover of a Dynamic Configuration

We will be interested in trees where the configuration is dynamic, i.e., we may start with an initial tree along with a configuration, and the tree may change by more nodes getting added to it and/or by the leaves changing labels.

For the sake of modeling dynamic trees, we will use the alternative representation of leaf nodes. Recall for a tree $\mathcal{T}$ of height h, if $i \in \mathsf{leaves}(\mathcal{T}) = [2^h - 1, 2^{h+1} - 2]$, then $\phi_h(i) = i - 2^h + 2$, is the position of the leaf i from the left. Let for a configuration L of tree $\mathcal{T}$ with height h, we define

$$\phi_h(L) = \{(\phi_h(i), s) : (i, s) \in L\}.$$

Thus, $\phi_h(L)$ is just a different representation of the configuration L, where the leaf nodes are specified by their position from the left. Similarly, for any $S \subseteq [1, 2^h] \times \{+, -\}$, we define $\phi_h^{-1}(S) = \{(\phi_h^{-1}(i), s) : (i, s) \in S\}$.

Let T_a, T_b be two complete binary trees of height h_a and h_b respectively. Let L_a and L_b be complete configurations of T_a and T_b respectively. Let $\tilde{L}_a = \phi_{h_a}(L_a)$ and $\tilde{L}_b = \phi_{h_b}(L_b)$.

Let $s \in \{+,-\}$. If $s = +$, then $\bar{s} = -$ and vice versa. We define two sets $\tilde{D}$ and $\tilde{A}$ as

$$\tilde{D} = \left\{(i,s) \in \tilde{L}_b : (i,\bar{s}) \in \tilde{L}_a\right\},$$
$$\tilde{A} = \left\{(i,s) \in \tilde{L}_b : (i,s) \notin \tilde{L}_a \bigwedge (i,\bar{s}) \notin \tilde{L}_a\right\}.$$

Note, the set $\tilde{D}$ contains those labeled leaf nodes in $\tilde{L}_a$ whose labels have changed in $\tilde{L}_b$ and $\tilde{A}$ contains those labeled nodes in $\tilde{L}_b$ which are not present in $\tilde{L}_a$. As, both L_a and L_b are complete configurations of the trees T_a and T_b, then by our definitions of $\tilde{D}$ and $\tilde{A}$ they are disjoint. We define the *change in configuration* of L_a and L_b as

$$\Delta(L_a, L_b) = \phi_{h_b}^{-1}\left(\tilde{D} \cup \tilde{A}\right). \tag{1}$$

Consider a finite sequence of trees $T_1, T_2, \ldots, T_\ell$ and their corresponding complete configurations $L_1, \ldots, L_\ell$. Let

$$L_i^\delta = \Delta(L_i, L_{i+1}). \tag{2}$$

Note that, L_i^δ represents a configuration (not necessarily, a complete one) of the tree T_{i+1}. It is easy to see that given T_i along with L_i^δ one can reconstruct L_{i+1}, i.e., the complete configuration of T_{i+1}.

We assume a scenario where we have an initial tree T_1 with a certain complete configuration and over time the tree changes where new labeled nodes are added to the tree or the labels in its leaf node change. We are given the initial tree and the changes that take place in each step, i.e., we have access to T_1 along with $L_1^\delta, L_2^\delta, \ldots, L_{\ell-1}^\delta$.

A dynamic cover generation algorithm outputs a cover on input a configuration. A dynamic cover reconstruction algorithm when given a sequence of covers generates a configuration.

Definition 4 (Dynamic Tree Cover Scheme). *A dynamic tree cover scheme* $\Psi_d = (\mathsf{dCoverGen}, \mathsf{dReConstruct})$ *is a tuple of two algorithms defined as follows.*

- $C \leftarrow \mathsf{dCoverGen}(h, L)$*: On input the height of the tree* h *and a configuration* L*,* $\mathsf{dCoverGen}$ *outputs a cover* C*.*
- $L \leftarrow \mathsf{dReConstruct}\left(\{h_i, C_i\}_{i\in[\ell]}\right)$*: On input a sequence of tree height* h_i *and corresponding cover* C_i*,* $\mathsf{dReConstruct}$ *outputs a configuration* L*.*

Correctness: Let $L_1, L_2, \ldots, L_\ell$ be as before, let $\emptyset$ denote the empty configuration corresponding to an empty tree and let

$$L_1^\delta = \Delta(\emptyset, L_1),$$
$$L_i^\delta = \Delta(L_i, L_{i+1}), 2 \le i \le \ell - 1.$$

We say, Ψ_d is correct if for all $j \le \ell$,

$$\Psi_d.\mathsf{dReConstruct}\left(\{h_i, \mathsf{dCoverGen}\left(h_i, L_i^\delta\right)\}_{i\in[j]}\right) = L_j.$$

4 Cover Generation Algorithms

For ease of exposition, we will sometimes impose colors on the nodes of the trees. For a node i, $\mathsf{color}(i)$ will denote its color. For a node i, $\mathsf{leftChild}(i)$ and $\mathsf{rightChild}(i)$ will denote its left and right child respectively. Recall, that configuration of a tree $\mathcal{T}$ is a set of labeled leaf nodes of $\mathcal{T}$, thus elements of a configuration are ordered pairs (i,s) where i is a node and $s \in \{+,-\}$ is its label, we will sometimes denote the label of i by $\mathsf{sign}(i)$.

We call a configuration L a *pure configuration* if, for every $(i,s) \in L$, s is the same, i.e., in a pure configuration, every node is either labeled $+$ or $-$. A configuration which is not pure is called a *mixed configuration.*

4.1 Pure Cover: A Tree Cover Scheme for pure Configurations

For the algorithms that follow, we assume that each tree node i is endowed with two fields $\mathsf{color}(i)$ and $\mathsf{sign}(i)$.

We start with a simple scheme which generates covers only for pure configurations. Let L_p be a pure configuration of a tree $\mathcal{T}$ of height h, and we want to construct a cover of L_p. As a first step, we color the nodes of the tree according to the scheme $\mathsf{PureColoring}(h, L_p)$ described in the algorithm in Figure 2. We start from the leaf nodes and color each node which is in the configuration with the color "GREEN". We proceed with the non-leaf nodes starting from the level just above the leaf nodes and color a node GREEN if both of its children are colored GREEN.

```
PureColoring(h, L_p):
01. Initialize a tree T with height h,
      where the nodes of the tree has no color.
02. for i ← 2^h − 1 to 2^{h+1} − 2      ▷ leaf nodes
03.     if (i, s) ∈ L_p
04.         color(i) ← GREEN, sign(i) ← s
05. for i ← 2^h − 1 to 0 of T      ▷ non-leaf nodes
06.     if color(leftChild(i)) = color(rightChild(i)) = GREEN
07.         color(i) ← GREEN, sign(i) ← sign(leftChild(i))
08. return T
```

Fig. 2. The coloring scheme for pure configurations

If we apply the algorithm $\mathsf{PureColoring}(h, L_p)$ on tree $\mathcal{T}$ with a pure configuration L_p, its nodes either get colored GREEN or they are without color.

Definition 5 (Top Node). *A colored node in a tree $\mathcal{T}$ is called a* top *node if the path from that node to the root does not contain any other colored node.*

We call the set of top nodes in a tree $\mathcal{T}$ with a pure configuration L_p as the cover of the configuration.

```
PureCoverGen(h, L_p):
01. T ← PureColoring(h, L_p)
02. for i ← 0 to 2^{h+1} − 2
03.     X_i ← ∅
04. for i = 2^h − 1 to 2^{h+1} − 2      ▷ leaf nodes
05.     if color(i) = GREEN
06.         X_i ← {(i, sign(i))}
07. for i ← 2^h − 2 to 0      ▷ every non-leaf node i
08.     if color(i) = GREEN
09.         X_i ← {(i, sign(i))}
10.     else
11.         X_i ← X_{leftChild(i)} ⋃ X_{rightChild(i)}
12. return X_0
```

Fig. 3. Algorithm for pure cover generation

With the above characterization of a cover of a pure configuration, we formulate an algorithm to construct one in Figure 3. Initially, the algorithm assigns an empty set X_i to each node i of the tree. For any leaf node j, if the leaf node is colored GREEN, then X_j is set to the singleton set $\{(j, \mathsf{sign}(j))\}$, otherwise X_j remains the empty set. For any non-leaf node i, if the node is colored GREEN, then X_i is set to $\{(i, \mathsf{sign}(i))\}$, else X_i is

set as $X_{\mathsf{leftChild}(i)} \bigcup X_{\mathsf{rightChild}(i)}$. Finally, the algorithm returns X_0, i.e., the set associated with the root node.

Proposition 1. $\mathsf{PureCoverGen}(h, L_p)$ *returns the top no-des of a tree of height* h *with configuration* L_p.

Proof. According to our coloring scheme and the definition of a top node, the following are true for any top node i.

1. Both children of i are GREEN. Our coloring scheme guarantees this.
2. The sibling of i is not GREEN, as otherwise, our coloring scheme will make the immediate ancestor of i also GREEN, which violates the condition that i is a top node.
3. No ancestor of i is GREEN, as i is a top node.

Based on the above observations, it follows that if i is a top node, then $X_i = \{(i, s)\}$ (see lines 6 and 9 of the Algorithm in Figure 3). Further, for any ancestor j of i, X_j contains (i, s) (see line 11 of Figure 3). As the root (i.e., node 0) is also an ancestor of i, hence X_0 contains (i, s). Thus, the set returned by $\mathsf{PureCoverGen}(h, L_p)$ contains all top nodes. Conversely, following the same arguments, it is easy to see that if $(i, s) \in X_0$, then i is a top node. □

We list some additional properties of covers generated by the cover generation algorithm PureCoverGen which are immediate from the algorithm.

Proposition 2. *Let* L_p *be a pure configuration of a tree* $\mathcal{T}$ *of height* h*, and let* $C_p = \mathsf{PureCoverGen}(h, L_p)$ *then the following are true.*

1. *If* $(i, s) \in C_p$ *and* i *is a leaf node then* $(i, s) \in L_p$.
2. *If* $(i, s) \in C_p$ *and* i *is not a leaf node then every leaf node of the subtree rooted at* i *occurs in* L_p *with sign* s.
3. *If* i *and* i' *be siblings in the tree* $\mathcal{T}$ *then both* (i, s), (i', s) *cannot be in* C_p.

```
PureReConstruct(h, C_p):
01. Initialize a tree T with height h,
    where the nodes of the tree has no color.
02. Initialize L_p ← ∅
03. for every node i = 0 to 2^h − 2     ▷ non-leaf nodes
04.    if (i, s) ∈ C_p
05.       for all j ∈ leaves(T(i))
06.          L_p ← L_p ∪ {(j, s)}
07. return L_p
```

Fig. 4. Algorithm for pure cover reconstruction

Now, given a pure cover C_p, corresponding to a pure configuration L_p, a reconstruction algorithm works as follows (see Figure 4). For every non-leaf node i in the cover, the reconstruction algorithm assigns the same sign to all the leaf nodes of the sub-tree rooted at node i and adds those leaf nodes along with their sign to the configuration L_p. And for every leaf node in the cover, it adds the node with its sign to the configuration.

The following proposition, which is easy to verify, asserts that the cover generation scheme is correct.

Proposition 3. *Let* L_p *be a pure configuration and* $C_p = \mathsf{PureCoverGen}(h, L_p)$ *and* $L'_p = \mathsf{PureReConstruct}(h, C_p)$*, then* $L_p = L'_p$.

Expected Cover Size of a Pure Cover In this section, we provide an analysis of the expected size of a cover returned by the Algorithm in Figure 3. Let $\mathcal{T}$ be a full binary tree of height h and thus $\mathcal{T}$ has a total of $n = 2^h$ leaf nodes. Let L_p be a pure configuration of $\mathcal{T}$ such that $|L_p| = r$, and without loss of generality, we assume that all nodes in L_p bear the sign $+$. There are $\binom{n}{r}$ such configurations possible, and we are interested in the average (expected value of) cover size over all these $\binom{n}{r}$ configurations.

Consider a sequence of $2n-1$ binary random variables $P_0, P_1, \ldots, P_{2n-2}$ corresponding to each node i of $\mathcal{T}$. Define

$$P_i = \begin{cases} 1; \text{ if } \mathsf{color}(i) = \text{GREEN} \\ 0; \text{ otherwise.} \end{cases}$$

That is, according to our coloring scheme, $P_i = 1$ denotes if the node i is colored GREEN or not. Consider any node i at the ℓ^{th} level of the tree $\mathcal{T}$, i.e., $i \in \left\{2^\ell - 1, \ldots, 2^{\ell+1} - 2\right\}$. Then, the subtree rooted at node i contains $2^{h-\ell}$ many leaf nodes. The event "$\{P_i = 1\}$" can then be viewed as choosing $2^{h-\ell}$ many BLACK balls from a bag containing a total of n balls, where r many are BLACK balls and the remaining $n-r$ are WHITE balls. Therefore,

$$\Pr[P_i = 1] = \begin{cases} \frac{\binom{r}{2^{h-\ell}}}{\binom{n}{2^{h-\ell}}} = \eta_{2^{h-\ell}}(n, r); \text{ if } 2^{h-\ell} \leq r \\ 0; \qquad\qquad\qquad\qquad \text{otherwise,} \end{cases}$$

where $\eta_\rho(\nu, \xi) = \frac{\binom{\xi}{\rho}}{\binom{\nu}{\rho}}$ denotes the probability of choosing ρ many BLACK balls from a bag containing ξ many are BLACK balls and the $\nu - \xi$ many WHITE balls. In order to avoid writing the boundary conditions every time, we extend the definition of $\eta_\rho(\nu, \xi)$ in the following way.

$$\eta_\rho(\nu, \xi) \triangleq \begin{cases} \frac{\binom{\xi}{\rho}}{\binom{\nu}{\rho}}; \text{ if } \nu, \xi, \rho \geq 0 \text{ and } \rho \leq \xi \\ 0; \quad \text{otherwise.} \end{cases}$$

Define another sequence of $2n-1$ binary random variables $X_0, X_1, \ldots, X_{2n-2}$ in the following manner.

$$X_i = \begin{cases} 1; \text{ if} \begin{cases} i = \text{ odd and } P_i = 1 \text{ and } P_{i+1} = 0 \\ i = \text{ even and } P_i = 1 \text{ and } P_{i-1} = 0 \end{cases} \\ 0 \text{ otherwise.} \end{cases}$$

That is, the random variable X_i corresponding to the node i contributes 1 to the size of the cover if i is a GREEN node, but its sibling node is not a GREEN node. Assume that i is odd and $2^\ell < i < 2^{\ell+1}$. Then,

$$\begin{aligned} \Pr[X_i = 1] &= \Pr[\{P_i = 1\} \cap \{P_{i+1} = 0\}] \\ &= \Pr[P_{i+1} = 0 | P_i = 1] \cdot \Pr[P_i = 1] \\ &= (1 - \Pr[P_{i+1} = 1 | P_i = 1]) \cdot \Pr[P_i = 1] \\ &= \left(1 - \frac{\binom{r-2^{h-\ell}}{2^{h-\ell}}}{\binom{n-2^{h-\ell}}{2^{h-\ell}}}\right) \cdot \eta_{2^{h-\ell}}(n, r) \\ &= \left(1 - \eta_{2^{h-\ell}}\left(n - 2^{h-\ell}, r - 2^{h-\ell}\right)\right) \cdot \\ &\quad \eta_{2^{h-\ell}}(n, r). \end{aligned}$$

Let the random variable X denote the cover size. Then, X can be expressed as

$$X = X_0 + X_1 + \cdots + X_{2^{h+1}-2}.$$

Assume that $2^\alpha \leq r < 2^{\alpha+1}$. This implies that $P_i = 0$ for all $0 \leq i \leq 2^{h-\alpha} - 2$ with the convention that if $2^{h-\alpha} - 2 < 0$ the such an i does not exist. Then the cover size X is given by,

$$\begin{aligned} X &= X_{2^{h-\alpha}-1} + X_{2^{h-\alpha}} + \cdots + X_{2^{h+1}-2} \\ &= \sum_{i=h-\alpha}^{h} \left(X_{2^i-1} + X_{2^i} + \cdots + X_{2^{i+1}-2}\right). \end{aligned}$$

Therefore, by linearity of expectation, the expected cover size is given by

$$\begin{aligned} E[X] &= \sum_{i=h-\alpha}^{h} \left(E[X_{2^i-1}] + E[X_{2^i}] + \cdots + E[X_{2^{i+1}-2}]\right) \\ &= \sum_{i=h-\alpha}^{h} \left\{2^i \cdot \left(1 - \eta_{2^{h-i}}\left(n - 2^{h-i}, r - 2^{h-i}\right)\right) \cdot \right. \\ &\quad \left. \eta_{2^{h-i}}\left(n, r\right)\right\}, \end{aligned} \tag{3}$$

where r denote the size of the pure configuration such that $2^\alpha \leq r < 2^{\alpha+1}$.

4.2 Mixed Cover: Generating a Smaller Sized Cover

In the previous section we discussed a scheme to generate covers assuming the configuration to be pure. In this section, we remove this restriction. Let L be a complete configuration for a tree $\mathcal{T}$ of height h. Hence, L contains all leaf nodes of $\mathcal{T}$ along with their sign, i.e.,

$$L = \left\{(i, \mathsf{sign}(i)) : i \in \mathsf{leaves}(\mathcal{T})\right\},$$

where $\mathsf{sign}(i) \in \{+, -\}$. Such a complete configuration can be naturally decomposed into pure configurations L_g and L_r where,

$$L_g = \left\{(i, +) : (i, +) \in L\right\}, \quad L_r = \left\{(i, -) : (i, -) \in L\right\}.$$

As L_g and L_r are pure configurations, we can use our cover generation algorithm for computing the covers of the pure configurations L_g and L_r as $C_g = \mathsf{PureCoverGen}\ (h, L_g)$, $C_r = \mathsf{PureCoverGen}\ (h, L_r)$. Note, that any one of C_g or C_r can be used as a cover of the complete configuration L. The reconstruction would be simple. We would reconstruct the pure configuration L_g (respectively L_r) from C_g (respectively C_r) and assign the opposite sign to all other leaf nodes of the tree. Thus, we can choose the one with smaller number of elements among C_g and C_r as the cover of L.

The above procedure would generate a correct cover for L, but we seek to find a more succinct cover. The above procedure yields a cover which contains nodes of the same sign, and hence we call such a cover as a *pure cover*. The procedure that we are about to describe will contain nodes with both signs and hence we name this algorithm a *mixed cover* algorithm.

As before, the heart of the algorithm is a coloring scheme $\mathsf{MixedColoring}(h, L_c)$ which is described in Figure 5. The algorithm takes in a complete configuration (L_c) for a tree $\mathcal{T}$ of height h, and colors the nodes of $\mathcal{T}$ as GREEN or RED. The algorithm examines the nodes level wise starting from the lowest level, i.e., the leaf nodes. All leaf nodes with sign $+$ are colored GREEN and leaf nodes with sign $-$ are colored RED. A non-leaf node gets the color GREEN if both its children are colored GREEN and get the color RED if both its children are RED. Other nodes remain uncolored.

```
MixedColoring(h, L_c):
01. Initialize a tree T with height h,
    where the nodes of the tree has no color.
02. for i ← 2^h − 1 to 2^{h+1} − 2
03.    if (i, +) ∈ L_c
04.       color(i) ← GREEN
05.    else
06.       color(i) ← RED
07. for i ← 2^h − 2 to 0
08.    if color(leftChild(i)) = color(rightChild(i)) = GREEN
09.       color(i) ← GREEN
10.    if color(leftChild(i)) = color(rightChild(i)) = RED
11.       color(i) ← RED
12. return T
```

Fig. 5. Coloring scheme for mixed covers

We discuss the main idea of our cover generation scheme next. It may be helpful to consider an example shown in Figure 6 all along. The number of leaf nodes in the tree $n = 16$. Consider that the nodes labeled + are $L_g = \{15, 16, 18, 19, 20, 23\}$, and the nodes labeled − are $L_r = \{17, 21, 22, 24, 25, 26, 27, 28, 29, 30\}$. According to our coloring algorithm of Figure 5, the nodes $\{15, 16, 18, 19, 20, 23\}$ will receive the color GREEN (denoted by shaded circles in the Figure 6) and the nodes $\{17, 21, 22, 24, 25, 26, 27, 28, 29, 30\}$ will be colored RED (shown by shaded squares in the figure).

Let $\mathcal{T}$ be a tree of height h with a complete configuration L_c and colored using MixedColoring(h, L_c). Recall that a node i is a top node of the colored tree $\mathcal{T}$ if i is colored and there is no colored node in the path from i to the root. In our example in Figure 6 the nodes 7, 17, 18, 9, 10, 23, 24, 12, and 6 are top nodes. Note that all leaf nodes of the subtree rooted at a top node i have the same color as that of i.

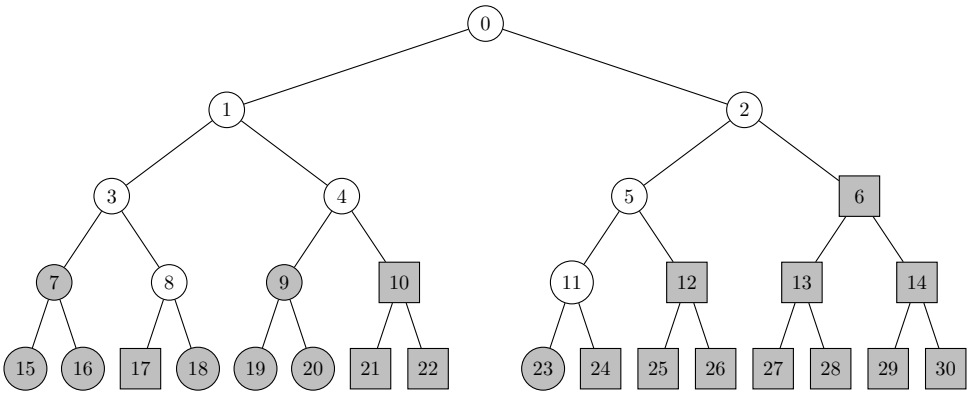


Fig. 6. An example of mixed cover: The leaf nodes represented by circles are assumed to bear the sign +, the leaf nodes represented by squares bear the sign −. The shaded circles represent nodes colored green and the shaded squares are nodes colored red.

The main intuition of our mixed cover generation algorithm is the following. Let i be a top node for a colored tree $\mathcal{T}$. Hence, the color of i is enough to determine the color of all the leaf nodes of the subtree rooted at i. In fact, in many cases a top node i along with its color can be used to uniquely color the leaf nodes of a tree much bigger than the subtree rooted at i. For example, in Figure 6, node 10 can be used to reconstruct the tree rooted at node 4: we will assign the sign of 10 to all of its leaf nodes and the opposite

sign to all the other leaf nodes of the subtree rooted at node 4. We want to make this intuition more concrete. We introduce some definitions for this purpose.

Definition 6 (Representable Top Nodes). *Let $\mathcal{T}$ be a tree of height h with complete configuration L_c and colored by* $\mathsf{MixedColoring}(h, L_c)$. *A set of top nodes S of $\mathcal{T}$ is called representable, if the following holds*

1. *All nodes in S are of same color $c \in \{\text{GREEN,RED}\}$.*
2. *There exists a subtree $\mathcal{T}'$ (of $\mathcal{T}$) containing all nodes in S, such that all leaf nodes of $\mathcal{T}'$ other than the leaf nodes of the trees rooted at nodes in S have a color different from c.*

If S is representable then any $\mathcal{T}'$ satisfying the property (2) above is said to represent S.

Definition 7 (Representative Tree). *Let $\mathcal{T}$ be a tree of height h with complete configuration L_c and colored by* $\mathsf{MixedColoring}(h, L_c)$. *$S$ be a representable set of top nodes of $\mathcal{T}$. The largest subtree of $\mathcal{T}$ which represents S is called the representative tree of S. By $\mathsf{rep}_{\mathcal{T}}(S)$ we will denote the root of the representative tree of S. If S is a singleton set containing only i, we will denote the root of the representative tree of S by $\mathsf{rep}_{\mathcal{T}}(i)$.*

From the definition it is immediate that every representable set of top nodes will have a representative tree. Moreover, every singleton set containing a top node is representable and thus has a representative tree.

In our example tree $\mathcal{T}$ in Figure 6 we have the set $\{17, 10\}$ is representable and $\mathsf{rep}_{\mathcal{T}}(\{17, 10\}) = 1$, whereas $\{17, 24\}$ is not representable. Moreover, we have $\mathsf{rep}_{\mathcal{T}}(7) = 7$, $\mathsf{rep}_{\mathcal{T}}(17) = 3$, $\mathsf{rep}_{\mathcal{T}}(6) = 6$, $\mathsf{rep}_{\mathcal{T}}(9) = \mathsf{rep}_{\mathcal{T}}(10) = 4$, $\mathsf{rep}_{\mathcal{T}}(23) = 2$, $\mathsf{rep}_{\mathcal{T}}(12) = 12$ and $\mathsf{rep}_{\mathcal{T}}(6) = 6$.

Definition 8 (Independent Nodes). *Let S_1, S_2 be two sets of representable top nodes of a colored tree $\mathcal{T}$ and let $\mathsf{rep}_{\mathcal{T}}(S_1) = i_1$ and $\mathsf{rep}_{\mathcal{T}}(S_2) = i_2$. We say S_1, S_2 are independent if $\mathcal{T}(i_1)$ and $\mathcal{T}(i_2)$ are disjoint.*

In our example, the sets {7}, {17} are not independent whereas {17}, {10} and {23}, {17} are independent.

Definition 9 (Span). *Let S be a set of top nodes of a tree $\mathcal{T}$ colored by* $\mathsf{MixedColoring}(h, L)$. *We say that S spans $\mathcal{T}$, if S can be decomposed into $S = S_1 \cup S_2 \cup \cdots \cup S_k$, for some $k \leq |S|$ such that the following holds*

1. *For all $i, j \in [k]$, $S_i \cap S_j = \emptyset$.*
2. *For all $i \in [k]$, S_i is representable.*
3. *For all $i, j \in [k]$, $i \neq j$, S_i and S_j are independent.*
4. *The union of the leaf nodes of the representative trees of the sets $S_1, \ldots, S_k$, gives the leaves of the tree $\mathcal{T}$, i.e.,*

$$\bigcup_{i \in [k]} \mathsf{leaves}\left(\mathcal{T}\left(\mathsf{rep}_{\mathcal{T}}(S_i)\right)\right) = \mathsf{leaves}(\mathcal{T}).$$

Observe that if X spans a colored tree $\mathcal{T}$, then the nodes in X along with their colors contain enough information to reconstruct the complete configuration of $\mathcal{T}$. Thus, for a given colored tree our goal is to find a set X of top nodes of minimum size which spans the tree. For the reconstruction, it is necessary to just decompose the span into representable sets and determine the representative tree for each such representable set.

Consider any node i of a colored tree $\mathcal{T}$. If i is colored and is of color $c \in \{\text{RED,GREEN}\}$, then i along with its color/sign spans the tree $\mathcal{T}(i)$. If i is un-colored, then the following statements hold and are easy to verify.

Proposition 4. *Let i be any un-colored node in a colored tree $\mathcal{T}$ and ρ and λ be the right child and left child of i respectively. Let X_ρ and X_λ span $\mathcal{T}(\rho)$ and $\mathcal{T}(\lambda)$ respectively. Then either both X_ρ and X_λ individually spans $\mathcal{T}(i)$ or $X_\lambda \cup X_\rho$ spans $\mathcal{T}(i)$.*

Proposition 5. *Let i be any un-colored node in a colored tree $\mathcal{T}$ and ρ and λ be the right child and left child of i respectively. Let X span $\mathcal{T}(i)$ and let $X = X_\lambda \cup X_\rho$, where $X_\lambda \subseteq \mathsf{nodes}(\mathcal{T}(\lambda))$ and $X_\rho \subseteq \mathsf{nodes}(\mathcal{T}(\rho))$. For $j \in \{\lambda, \rho\}$, if $X_j \neq \emptyset$, then X_j spans $\mathcal{T}(j)$. Moreover, if $X_\lambda = \emptyset$ then X_ρ contains nodes of the same color and all leaves of $\mathcal{T}(\lambda)$ are of color different from that of the nodes of X_ρ.*

The above observation is central to our algorithm $\mathsf{MixedCoverGen}(h, L_c)$ for generating covers described in Figure 7. The algorithm finds a cover for a tree $\mathcal{T}$ of height h with a complete configuration L_c. The algorithm assigns to each node i, three sets namely $X_{i,g}$, $X_{i,r}$, $X_{i,m}$. $X_{i,g}$ and $X_{i,r}$ contains the GREEN and RED top nodes of the subtree rooted at i. $X_{i,m}$ is empty if i is a leaf node, otherwise we set

$$Y \leftarrow \mathsf{MixedUnion}(i).$$

The procedure $\mathsf{MixedUnion}(i)$, when i is not a leaf node, is described in the right column of Figure 7. $\mathsf{MixedUnion}(i)$ computes the following collection of sets

$$\mathcal{C} = \Big\{ X_{\mathsf{leftChild}(i),x} \bigcup X_{\mathsf{rightChild}(i),y} : x, y \in \{r, g, m\}; \\ x \neq y; X_{\mathsf{leftChild}(i),x}, X_{\mathsf{rightChild}(i),y} \neq \emptyset \Big\},$$

and returns the set Y of minimum cardinality from $\mathcal{C}$. If $|Y|$ is smaller than both $|X_{i,g}|$ and $|X_{i,r}|$, then $X_{i,m}$ is set to Y otherwise it remains empty. Finally, the algorithm outputs $C = \mathsf{minCard}(X_{0,r}, X_{0,g}, X_{0,m})$.

Theorem 1. *Let $\mathcal{T}$ be a tree of height h, and L_c be a complete configuration of $\mathcal{T}$. Let $\mathsf{MixedCoverGen}(h, L_c)$ compute the sets $X_{i,g}, X_{i,r}, X_{i,m}$ for each node i of $\mathcal{T}$ as described in Figure 7. Let $C_i = \{X_{i,g}, X_{i,r}, X_{i,m}\}$. Then the following are true.*

1. *C_i contains at least one non-empty set.*
2. *Any non-empty set $X \in C_i$ spans $\mathcal{T}(i)$.*
3. *$\mathsf{minCard}(C_i)$ is the smallest set of nodes which spans $\mathcal{T}(i)$.*

The proof of Theorem 1 is presented in Appendix A. Theorem 1 asserts that the algorithm $\mathsf{MixedCoverGen}$ (h, L_c) outputs the smallest possible set X which spans a tree of height h with a complete configuration L_c. Now our goal is to regenerate the configuration of a tree of height h, given h and a set C which spans the tree and was produced by $\mathsf{MixedCoverGen}(h, L_c)$. The reconstruction algorithm is presented in Figure 8.

First, note that it may be the case that the algorithm $\mathsf{MixedCoverGen}(h, L_c)$ generates a pure cover, i.e., it generates C where all nodes in C are labeled with the same sign. In such a case either $C = X_{0,r}$ or $C = X_{0,g}$, i.e., C contains all top nodes of a single color. In this case, the reconstruction is simple, as the representative tree for C is the complete tree, i.e., $\mathsf{rep}_{\mathcal{T}}(C) = 0$. For reconstruction, the following procedure would suffice: For every $(i, \mathsf{sign}(i)) \in C$, the leaves of the sub-tree $\mathcal{T}(i)$ are labeled with $\mathsf{sign}(i)$ and the rest of the leaf nodes are labeled with the opposite sign. These steps are done in lines 5 to 8 of the algorithm described in Figure 8.

If C is a mixed cover, i.e., C contains nodes of both signs then $C = X_{0,m}$. In this case, the reconstruction procedure is a bit more involved. For convenience, we introduce some

MixedCoverGen(h, L_c):	MixedUnion(i):
01. $\mathcal{T} \leftarrow \mathsf{MixedColoring}(h, L_c)$	01. $\mathsf{cnt} \leftarrow 0$
02. **for** every node i of tree $\mathcal{T}$	02. **for** $x \in \{r, g, m\}$
03. $X_{i,g}, X_{i,r}, X_{i,m} \leftarrow \emptyset$	03. **for** $y \in \{r, g, m\}$
04. **for** $i \leftarrow 2^h - 1$ to 0	04. $\lambda \leftarrow \mathsf{leftChild}(i)$
05. **if** $\mathsf{color}(i) =$ GREEN	05. $\rho \leftarrow \mathsf{rightChild}(i)$
06. $X_{i,g} \leftarrow X_{i,g} \bigcup \{(i, +)\}$	06. **if** $(x \neq y) \bigwedge X_{\lambda,x} \neq \emptyset \bigwedge X_{\rho,y} \neq \emptyset$
07. **else-if** $\mathsf{color}(i) =$ RED	07. $Y_{\mathsf{cnt}} \leftarrow X_{\lambda,x} \bigcup X_{\rho,y}$
08. $X_{i,r} \leftarrow X_{i,r} \bigcup \{(i, -)\}$	08. $\mathsf{cnt} \leftarrow \mathsf{cnt} + 1$
09. **else**	09. $X \leftarrow \mathsf{minCard}(Y_0, Y_1, \ldots, Y_{\mathsf{cnt}})$
10. $X_{i,g} \leftarrow X_{\mathsf{leftChild}(i),g} \bigcup X_{\mathsf{rightChild}(i),g}$	10. **return** X
11. $X_{i,r} \leftarrow X_{\mathsf{leftChild}(i),r} \bigcup X_{\mathsf{rightChild}(i),r}$	
12. $Y \leftarrow \mathsf{MixedUnion}(i)$	
13. **if** $\lvert Y \rvert < \min\{\lvert X_{i,g} \rvert, \lvert X_{i,r} \rvert\}$	
14. $X_{i,m} \leftarrow Y$	
15. **return** $\mathsf{minCard}(X_{0,g}, X_{0,r}, X_{0,m})$	

Fig. 7. Algorithm for mixed cover generation

additional notations. For any node i in a tree $\mathcal{T}$, let $\mathsf{path}(i)$ denote the sequence of nodes in the unique path from i to the root of $\mathcal{T}$. For any node $j \neq i$ of $\mathsf{path}(i)$, $\mathsf{prev}_i(j)$ denotes the node preceding j in the sequence $\mathsf{path}(i)$.

Reconstructing the configuration, given a mixed cover C boils down to decomposing C into disjoint subsets such that each subset is representable. Once such representable subsets are obtained the leaves of the corresponding representative trees can be assigned signs (colors), and this assignment of signs would yield the desired complete configuration.

Let C be the output of $\mathsf{MixedCoverGen}$ (h, L_c) and let C be a mixed cover. Let $(j, s) \in C$, where $s \in \{+, -\}$ and let $\bar{s}$ be the sign opposite to s. Let, ℓ_j be the first node in $\mathsf{path}(j)$ which intersects with $\mathsf{path}(i)$ for some node i such that $(i, \bar{s}) \in C$. As C is a mixed cover, hence C contains at least two elements with different signs and thus for every $(j, s) \in C$, ℓ_j is well-defined.

Let λ and ρ be the left and right children of ℓ_j. Then, if i belongs to $\mathcal{T}(\lambda)$ then j belongs to $\mathcal{T}(\rho)$ and vice versa. Without loss of generality, let i belong to $\mathcal{T}(\lambda)$ also let J be those nodes in $\mathcal{T}(\rho)$ which belongs to C, i.e.,

$$J = \{(k, \mathsf{sign}(k)) : (k, \mathsf{sign}(k)) \in C, k \in \mathcal{T}(\rho)\}. \tag{4}$$

It is easy to see that all nodes in J bears the same sign as that of j, as otherwise ℓ_j cannot be the first node in $\mathsf{path}(j)$ which intersects with the path of another node in C with a sign different from the sign of j. We observe the following

Proposition 6. *The set J as defined in Eq. (4) is representable and $\rho = \mathsf{rep}_{\mathcal{T}}(J)$.*

Proof. As all nodes in J are top nodes and bear the same sign, then in the corresponding colored tree $\mathcal{T}$ all nodes in J have the same color, say GREEN. With reference to the algorithm $\mathsf{MixedCoverGen}((h, L_c))$, it is immediate that $J \subseteq X_{\rho,g}$. We claim that $J = X_{\rho,g}$, i.e., J contains all GREEN top nodes in $\mathcal{T}(\rho)$. As C is a cover of $\mathcal{T}$ and $J \subseteq C$, by Theorem 1, C spans $\mathcal{T}$, and the nodes in J are the only GREEN top nodes in $\mathcal{T}(\rho)$ which are in C. If there are GREEN top nodes in $\mathcal{T}(\rho)$ which are not in J then C cannot span $\mathcal{T}$. Thus, J is representable and $\mathcal{T}(\rho)$ represents J.

We are left to show that $\mathcal{T}(\rho)$ is the largest tree that represents J, i.e., $\mathsf{rep}_{\mathcal{T}}(J) = \rho$. The smallest tree larger than $\mathcal{T}(\rho)$ which contains $\mathcal{T}(\rho)$ is $\mathcal{T}(\ell_j)$. We will argue that $\ell_j \neq \mathsf{rep}_{\mathcal{T}}(J)$. Note that, $i \in C$ and i belongs to the left subtree $\mathcal{T}(\lambda)$ of $\mathcal{T}(\ell_j)$ and i has color different from j, i.e., i is colored RED. If $\ell_j = \mathsf{rep}_{\mathcal{T}}(J)$, then a set containing J and another set containing i cannot be independent, which implies that both nodes in J and the node i cannot be in C which by Theorem 1 spans $\mathcal{T}$. □

The above proposition is central to the reconstruction algorithm presented in Figure 8 when C is a mixed cover. Lines 10-14 of the algorithm presented in Figure 8 finds the node ℓ_j (as described above) for each j in C and thus decomposes C into representative sets J as asserted in Proposition 6. This process is iterated until all leaves of the tree are labeled.

```
MixedReConstruct(h, C):
01. Initialize a tree T with height h, where the nodes of the tree has no color.
02. L ← ∅
03. for all (i, s) ∈ C
04.    for all k ∈ leaves(T(i))
05.       L ← L ∪ {(k, s)}
05. if C is a pure cover with sign s ∈ {+, −}
06.    for all k' ∈ leaves(T)
07.       if (k', s) ∉ L
08.          L ← L ∪ {(k', s̄)}
09. else
10.    for each (i, s) ∈ C
11.       find the first ancestor node j of i in path(i) such that j ∈ path(l) for some (l, s̄) ∈ C
12.       for all the leaf nodes k of the sub-tree rooted at the node prev_i(j)
13.          if (k, s) ∉ L
14.             L ← L ∪ {(k, s̄)}
15. return L
```

Fig. 8. Algorithm for mixed cover reconstruction

4.3 Dynamic Tree Cover Scheme

Recall the setting of a dynamic tree cover scheme as discussed in Section 3.2. We have a sequence of trees $T_1, T_2, \ldots, T_\ell$, with their corresponding complete configurations $L_1, L_2, \ldots$, L_ℓ, and for $1 \leq i \leq \ell - 1$, L_i^δ is defined as in Equations (1) and (2). Note, each L_i^δ represents a configuration, not necessarily a complete one, of a tree of height h_i, where h_i is the height of the shortest complete binary tree which contains all nodes in L_i^δ. The dynamic cover generation algorithm takes L_i^δ and generates a cover C_i. The property which is required is that, knowing the complete configuration of T_1 and the sequence of covers $C_2, C_3, \ldots, C_\ell$ corresponding to the configurations $L_2^\delta, L_3^\delta, \ldots, L_{\ell-1}^\delta$, the cover reconstruction algorithm can generate the complete configuration of T_ℓ.

The dynamic cover generation algorithm just takes a configuration and produces its cover. If the input configuration L is a complete configuration, then the mixed cover generation algorithm is used to generate the cover. Otherwise, L is decomposed into two sets L_g and L_r, where L_g contains the nodes labeled $+$ and L_r contains the nodes labeled $-$, and the pure cover generation algorithm is used to generate covers C_g and C_r for the configurations L_g and L_r respectively. Finally, $C = C_g \cup C_r$ is produced as the output. The details are shown in Figure 9.

```
dCoverGen(h, L):
01. Initialize an empty tree T of height h
02. if |L| = 2^h
03.    C ← MixedCoverGen(h, L)
04. else
05.    C, L_g, L_r ← ∅
06.    for all (i, +) ∈ L
07.       L_g ← L_g ∪ {(i, +)}
08.    C_g ← PureCoverGen(h, L_g)
09.    for all (i, −) ∈ L
10.       L_r ← L_r ∪ {(i, −)}
11.    C_r ← PureCoverGen(h, L_r)
12.    C ← C_g ∪ C_r
13. return C
```

Fig. 9. Algorithm for dynamic cover generation

To explain the reconstruction procedure we introduce a new operation on configurations. Let L_α and L_β are two configurations for trees with heights h_α and h_β. Let $h_{\max} = \max\{h_\alpha, h_\beta\}$, $\tilde{L}_\alpha = \phi_{h_\alpha}(L_\alpha)$ and $\tilde{L}_\beta = \phi_{h_\beta}(L_\beta)$. We define a new configuration for a tree of height $h_{\max}$ as

$$L_\alpha \rhd_{(h_\alpha, h_\beta)} L_\beta = \phi^{-1}_{h_{\max}}(X \cup Y \cup Z), \tag{5}$$

where X, Y, Z are defined as follows:

$$\begin{aligned} X &= \{(i,s) \in \tilde{L}_\alpha : (i,s) \notin \tilde{L}_\beta \text{ and } (i,\bar{s}) \notin \tilde{L}_\beta\} \\ Y &= \{(i,s) \in \tilde{L}_\beta : (i,s) \notin \tilde{L}_\alpha, (i,\bar{s}) \notin \tilde{L}_\alpha\} \\ Z &= \{(i,s) \in \tilde{L}_\beta : (i,\bar{s}) \in \tilde{L}_\alpha\}. \end{aligned}$$

$L_\alpha \rhd_{(h_\alpha, h_\beta)} L_\beta$ is essentially the configuration obtained by overwriting L_α by L_β. Notice that the set X contains those labeled nodes in L_α which are not present in L_β. Similarly, Y contains those nodes in L_β which are not present in L_α, and Z contains those nodes in L_β which are present in L_α but with the opposite label.

For reconstruction, the sequence of covers, along with the corresponding tree heights, is used. Suppose C_1 be the cover of L_1 and for $2 \le i \le \ell - 1$, C_i be the cover of L_i^δ. The cover reconstruction algorithm first reconstructs the configuration L_i for each cover C_i and then outputs the configuration

$$L = ((((L_2 \rhd L_2) \rhd L_3) \rhd \cdots) \rhd L_{\ell-1}).$$

The details are in Figure 10.

5 Constructing Static SSE Using Tree Cover

In this section, we will show how to use the tree cover scheme to design a Static SSE scheme. A static database is where the number of documents in the database is a fixed natural number n. Let, $\mathcal{D} = \{\mathsf{d}_1, \ldots, \mathsf{d}_n\}$ be the set of documents and $\mathcal{I} = \{\mathsf{id}_1, \ldots, \mathsf{id}_n\}$ be the set of identifiers associated with those documents. We assume a natural ordering of the identifiers in the database, where id_i represents the i-th identifier of the database. Let $w \in \mathsf{W}$ be an arbitrary keyword and m_w be the largest integer such that $\mathsf{id}_{m_w} \in \mathsf{db}(w)$, and let h_w be the smallest integer such that $m_w \le 2^{h_w}$.

Let $\mathcal{T}_w$ be a complete binary tree of height h_w. We will represent $\mathsf{db}(w)$ by a configuration of the tree $\mathcal{T}_w$. We associate each identifier id_i, $i \le m_w$ with a leaf node of the tree

dCoverReConstruct($\{(h_i, C_i)\}_{i\in[0,\ell-1]}$):
01. $L_0 \leftarrow$ MixedReConstruct(h_0, C_0)
02. **for** $i = 1$ to $\ell - 1$
03. Initialize $X_i, C_p, C_r \leftarrow \emptyset$
04. **for** all $(j, +) \in C_i$
05. $C_p \leftarrow C_p \cup \{(j, +)\}$
06. $L_p \leftarrow$ PureReConstruct(h_i, C_p)
07. **for** all $(j, -) \in C_i$
08. $C_r \leftarrow C_r \cup \{(j, -)\}$
09. $L_r \leftarrow$ PureReConstruct(h_i, C_r)
10. $X_i \leftarrow L_p \cup L_r$
11. $h \leftarrow h_0$, $L \leftarrow L_0$
12. **for** $i = 1$ to $\ell - 1$,
13. $L \leftarrow L \rhd_{(h,h_i)} X_i$
14. $h \leftarrow \max\{h, h_i\}$
15. **return** L

Fig. 10. Algorithm for dynamic cover reconstruction

$\mathcal{T}_w$, though the injective map $\varphi^{-1} : \mathcal{I} \to \mathsf{nodes}(\mathcal{T}_w)$, where

$$\varphi^{-1}(\mathsf{id}_i) = \phi^{-1}(i) = i + 2^{h_w} - 2.$$

Note, the ϕ^{-1} function was introduced in Section 3. With the above specification, $\varphi^{-1}(\mathsf{id}_i)$ represents the i^{th} leaf node from the left of the tree $\mathcal{T}_w$.

We label the leaf nodes of $\mathcal{T}_w$ as follows. For every $i \le m_w$, the leaf node $\varphi^{-1}(\mathsf{id}_i)$ is labeled $+$ if $\mathsf{id}_i \in \mathsf{db}(w)$ and is labeled $-$ if $\mathsf{id}_i \notin \mathsf{db}(w)$. Moreover, if $m_w < 2^{h_w}$, then for all i, $m_w < i \le 2^{h_w}$, the leaf nodes $\varphi^{-1}(\mathsf{id}_i)$ are labeled $-$. This labelling of the leaves of $\mathcal{T}_w$ yields a complete configuration of $\mathcal{T}_w$, and we call this configuration L_w. Note that this configuration L_w uniquely represents the set $\mathsf{db}(w)$.

We now fix a tree cover scheme $\Psi = (\mathsf{CoverGen}, \mathsf{ReConstruct})$, and let $C_w \leftarrow \Psi.\mathsf{CoverGen}(h_w, L_w)$, where C_w is the cover of the configuration L_w of the tree $\mathcal{T}_w$ of height h_w. Thus, from C_w, the configuration L_w and further the set $\mathsf{db}(w)$ can be uniquely reconstructed. This interpretation of $\mathsf{db}(w)$ as a configuration of $\mathcal{T}_w$, which can be represented by a cover, will help us construct an efficient SSE scheme.

This is important to note that the height of a tree completely specifies it. So, it is not required by any of our schemes described later to store the tree explicitly.

5.1 Cover-based Representation of DB.

As a first step of constructing a cover-based SSE, we need to convert the given database DB to a different representation $\widetilde{\mathsf{DB}}$. We call this the *converted database*. This $\widetilde{\mathsf{DB}}$ then acts as the input to the existing SSE scheme. For each $w \in \mathsf{W}$, we generate a configuration L_w from the set $\mathsf{db}(w)$ as described before. Let $C_w \leftarrow \Psi.\mathsf{CoverGen}(L_w)$. We define

$$\widetilde{\mathsf{db}}(w) = \{(c, s, h_w) : (c, s) \in C_w\}. \tag{6}$$

The elements of $\widetilde{\mathsf{db}}(w)$ are the λ-bit encoding of the tuple (c, s, h_w). With this we define

$$\widetilde{\mathsf{DB}} = \bigcup_{w\in\mathsf{W}} \widetilde{\mathsf{db}}(w) \times \{w\}.$$

The conversion scheme from DB to $\widetilde{\mathsf{DB}}$ is summarized in $\mathsf{dbConversion}$ Algorithm in Figure 11.

$\mathsf{dbConversion}(\mathsf{DB})$	$\mathsf{Conversion}(\mathsf{db}(w))$
01. set $\widetilde{\mathsf{DB}} \leftarrow \emptyset$	01. set $\widetilde{\mathsf{db}}(w), L_w \leftarrow \emptyset$
02. **for** each keyword $w \in \mathsf{W}$	02. let m_w be the largest integer such that $\mathsf{id}_{m_w} \in \mathsf{db}(w)$
03. $\quad \widetilde{\mathsf{db}}(w) \leftarrow \mathsf{Conversion}(\mathsf{db}(w))$	03. let h_w be the smallest integer such that $m_w \leq 2^{h_w}$
04. $\quad$ **for** all $\widetilde{\mathsf{id}} \in \widetilde{\mathsf{db}}(w)$	04. initialize an empty full binary tree $\mathcal{T}_w$ of height h_w
05. $\quad\quad \widetilde{\mathsf{DB}} \leftarrow \widetilde{\mathsf{DB}} \cup \{(\widetilde{\mathsf{id}}, w)\}$	05. define $\mathcal{P} = \{\varphi^{-1}(\mathsf{id}_i) : \mathsf{id}_i \in \mathsf{db}(w)\}$
06. **return** $\widetilde{\mathsf{DB}}$	06. define $\mathcal{R} = \mathsf{leaves}(\mathcal{T}_w) \setminus \mathcal{P}$
	07. **for** all $p \in \mathcal{P}$
	08. $\quad L_w \leftarrow L_w \cup \{(p, +)\}$
	09. **for** all $r \in \mathcal{R}$
	10. $\quad L_w \leftarrow L_w \cup \{(r, -)\}$
	11. $C_w \leftarrow \Psi.\mathsf{CoverGen}(h_w, L_w)$
	12. **for** all $(c, s) \in C_w$
	13. $\quad \widetilde{\mathsf{db}}(w) \leftarrow \widetilde{\mathsf{db}}(w) \cup \{(c, s, h_w)\}$
	14. **return** $\widetilde{\mathsf{db}}(w)$

Fig. 11. Algorithm to convert DB to $\widetilde{\mathsf{DB}}$

5.2 Generic Static SSE Using Keyword Cover

Let us consider a database

$$\mathsf{DB} = \bigcup_{w \in \mathsf{W}} \{(\mathsf{id}, w) : \forall\ \mathsf{id} \in \mathsf{db}(w)\},$$

and any secure Static SSE scheme $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\ \mathsf{Search})$ as defined in Definition 1 (without the update protocol). Our goal is to covert Σ into a new SSE $\mathsf{s}\Sigma = (\mathsf{s}\Sigma.\mathsf{Setup}, \mathsf{s}\Sigma.\mathsf{Search})$. The procedures $\mathsf{s}\Sigma.\mathsf{Setup}$ and $\mathsf{s}\Sigma.\ \mathsf{Search}$ are described in Figure 12. We call the SSE Σ as the base SSE.

$\mathsf{s}\Sigma.\mathsf{Setup}$ takes in the database DB and a security parameter λ and outputs an encrypted database EDB, a key k and a client state σ_C. EDB is uploaded to the server and σ_C and k are retained with the client. $\mathsf{s}\Sigma.\mathsf{Setup}$ calls the routine $\mathsf{dbConversion}(\mathsf{DB})$, described in Figure 11 and converts DB to $\widetilde{\mathsf{DB}}$. Further $\widetilde{\mathsf{DB}}$ is sent as input to $\Sigma.\mathsf{Setup}$, the setup routine of the base Static SSE scheme.

Search for a keyword w is performed by running the $\mathsf{s}\Sigma.\mathsf{Search}$ protocol described in Figure 12. In the protocol, first the client's side $\mathsf{Search}_\mathsf{C}$ algorithm corresponding to the base SSE scheme Σ is executed on the inputs $(k, \sigma_\mathsf{C}, w)$. $\mathsf{Search}_\mathsf{C}$ $(k, \sigma_\mathsf{C}, w)$ returns a search token stoken_w for the keyword w and the updated client state σ_C. The search token stoken_w is sent to the server, which subsequently runs the $\Sigma.\mathsf{Search}_\mathsf{S}$ $(\mathsf{stoken}_w, \mathsf{EDB})$ and outputs res, the search result, which is sent to the client. The rest of the procedure, i.e., lines 15 to 23 of Figure 12 runs in the client side. Where the client obtains the set $\widetilde{\mathsf{db}}(w)$ from the search result sent by the server. Note, as described in Equation 6, $\widetilde{\mathsf{db}}(w)$ is a collection of tuples which encode a cover of a tree of height h_w. Using the cover reconstruction algorithm, we reconstruct the configuration of the tree representing the set $\mathsf{db}(w)$ and finally output it.

It is worth noting a few important characteristics of the scheme $\mathsf{s}\Sigma$.

Correctness. The correctness of $\mathsf{s}\Sigma$ directly follows from the correctness of the base Static SSE scheme Σ and the correctness of the cover generation scheme Ψ.

$\mathsf{s\Sigma.Setup}(1^\lambda, \mathsf{DB})$	$\mathsf{s\Sigma.Search}(k, \sigma_\mathsf{C}, w)$
01. $\widetilde{\mathsf{DB}} \leftarrow \mathsf{dbConversion}(\mathsf{DB})$	10. Generate $(\sigma_\mathsf{C}, \mathsf{stoken}_w) \leftarrow \Sigma.\mathsf{Search}_\mathsf{C}(k, \sigma_\mathsf{C}, w)$
02. $(k, \sigma_\mathsf{C}, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Setup}(1^\lambda, \widetilde{\mathsf{DB}})$	11. Send stoken_w to the server
03. Send EDB to server and retain (k, σ_C)	12. $\mathsf{res} \leftarrow \Sigma.\mathsf{Search}_\mathsf{S}(\mathsf{stoken}_w, \mathsf{EDB})$
	13. Send res to the client
	14. Decrypt and generate $\widetilde{\mathsf{db}}(w)$ from res
	15. Set $C_w, \mathsf{db}(w) \leftarrow \emptyset$
	16. **for** each $\widetilde{\mathsf{id}} \in \widetilde{\mathsf{db}}(w)$
	17. Parse $\widetilde{\mathsf{id}}$ as (c, s, h_w)
	18. $C_w \leftarrow C_w \cup \{(c, s)\}$
	19. Generate $L_w \leftarrow \Psi.\mathsf{ReConstruct}(h_w, C_w)$
	20. **for** each $(\ell, s) \in L_w$
	21. **if** $s = +$
	22. $\mathsf{db}(w) \leftarrow \mathsf{db}(w) \cup \{\varphi(\ell)\}$
	23. **return** $\mathsf{db}(w)$ to client

Fig. 12. Generic Static SSE using Tree-Cover scheme: Setup and Search operations

Benefits. Performance of any SSE schemes is measured by the search and update time and the amount of data communicated during these processes. All these parameters essentially depend on the databases under consideration. For a keyword w, let $n_w = |\mathsf{db}(w)|$ be the number of document identifiers containing the keyword w. In all the state-of-the-art SSE schemes, either static or dynamic, the search complexity is $\mathcal{O}(n_w)$ [16,30,17,11]. In many dynamic schemes, like in [23,9,6,7,15], the search complexity is in order of the number of updates for a keyword, which in the worst case may exceed $\mathcal{O}(n_w)$. Asymptotically, the worst case search complexity of our scheme is also $\mathcal{O}(n_w)$, but on average the exact number of tuples that need to be communicated is much less than n_w. To the best of our knowledge, in no existing scheme the search complexity is smaller than n_w.

Note that, this improvement of search time is obtained by our scheme because we use a compact but lossless representation of $\mathsf{db}(w)$. This compact representation results in a smaller index size which further results in more efficient search and communication. The exact savings obtained by pairing our scheme with existing SSEs are further discussed in Section 7. Concrete experimental data on real databases is presented in Section 9.

Extra overhead. Superficially, it may look that $\mathsf{s\Sigma}$ requires more computation than Σ. As $\mathsf{s\Sigma}$ requires the conversion of DB to $\widetilde{\mathsf{DB}}$ in the setup phase and the conversion of the cover to the configuration (lines 15-23 in Figure 12). These extra computations are negligible and take place on the client side. Conversion of db to $\widetilde{\mathsf{db}}$ for a database of decently large size only takes a few seconds whereas reconstruction of db from cover takes less than a second. More detailed discussion on this can be found in Section 9.3. The important savings that are achieved through $\mathsf{s\Sigma}$ is that, on average, the size of $\widetilde{\mathsf{DB}}$ is much smaller than DB, and this will significantly reduce the costs of the routines $\Sigma.\mathsf{Search}$. Moreover, this will lead to a smaller size of res which leads to a lower communication cost.

Security. The scheme $\mathsf{s\Sigma}$ just does some preprocessing of the input to the base scheme Σ and further does some post processing of the decrypted output. These pre and post processing takes place on the client side and does not require any computation involving the secret key(s). This implies that the security of the base scheme Σ implies the security of $\mathsf{s\Sigma}$.

$\mathsf{d\Sigma.Setup}\left(1^\lambda, \mathsf{DB}\right)$:
01. Set $\mathsf{B} \leftarrow \emptyset$
02. Initialize an empty map ts
03. $\widetilde{\mathsf{DB}} \leftarrow \mathsf{dbConversion}_d(\mathsf{ts}, \mathsf{DB})$
04. Run $(k, \sigma_\mathsf{C}, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Setup}(1^\lambda, \widetilde{\mathsf{DB}})$
05. Send EDB to server and retain $(k, \sigma_\mathsf{C}, \mathsf{B}, \mathsf{ts})$

$\mathsf{dbConversion}_d\,(\mathsf{ts}, \mathsf{DB})$
01. set $\widetilde{\mathsf{DB}} \leftarrow \emptyset$
02. **for** all $w \in \mathcal{W}$
03. $\quad \mathsf{ts}_w \leftarrow \perp$
02. **for** each keyword $w \in \mathsf{W}$
03. $\quad$ Set $\mathsf{ts}_w \leftarrow 0$
04. $\quad \mathsf{DB}^* \leftarrow \mathsf{dbConversion}\,(\mathsf{DB})$
05. $\quad$ **for** all $(\mathsf{id}, w) \in \mathsf{DB}^*$
06. $\quad\quad \widetilde{\mathsf{id}} \leftarrow \mathsf{id} \| \mathsf{ts}_w$
07. $\quad\quad \widetilde{\mathsf{DB}} \leftarrow \widetilde{\mathsf{DB}} \cup \{(\widetilde{\mathsf{id}}, w)\}$
08. **return** $\widetilde{\mathsf{DB}}$

Fig. 13. A generic Dynamic SSE using Tree-Cover Scheme: Setup phase and DB to $\widetilde{\mathsf{DB}}$ conversion algorithm

$\mathsf{BufferUpdate}((\mathsf{op}, \mathsf{in}), \mathsf{B})$:
01. **if** $(\mathsf{op} = \mathsf{add}) \wedge ((\mathsf{del}, \mathsf{in}) \in \mathsf{B})$
02. $\quad \mathsf{B} \leftarrow \mathsf{B} \setminus \{(\mathsf{del}, \mathsf{in})\}$
03. $\quad \mathsf{B} \leftarrow \mathsf{B} \cup \{(\mathsf{op}, \mathsf{in})\}$
04. **else-if** $(\mathsf{op} = \mathsf{del}) \wedge ((\mathsf{add}, \mathsf{in}) \in \mathsf{B})$
05. $\quad \mathsf{B} \leftarrow \mathsf{B} \setminus \{(\mathsf{add}, \mathsf{in})\}$
06. $\quad \mathsf{B} \leftarrow \mathsf{B} \cup \{(\mathsf{op}, \mathsf{in})\}$
07. **else**
08. $\quad \mathsf{B} \leftarrow \mathsf{B} \cup \{(\mathsf{op}, \mathsf{in})\}$
09. **return** B

Fig. 14. Buffer Update algorithm

We make this intuition more concrete in Section 8, where we present a reductionist security proof of our dynamic scheme $\mathsf{d\Sigma}$ (described in Section 6). The security result and the proof is also applicable for the static scheme.

Additional security advantage. Informally, an SSE scheme is called *volume hiding* if an adversary cannot guess the number of tuples that are related to a query by seeing the result of a query which is transmitted by a server. This additional security property in SSE schemes has been recently studied [27,22,3]. In general, making an SSE volume hiding makes it in-efficient in terms of communication and storage costs. In our scheme, the size of a search result for a keyword w is related to the size of the cover of the keyword w, and it does not directly reveal the number of tuples related to the keyword w. Thus, a constrained passive adversary, who only sees the communication between the server and the client, will not be able to accurately estimate the size of a query result from observing the response sizes. We believe that by using some additional randomness we can make our scheme to be volume hiding for more powerful adversaries.

6 Constructing Dynamic SSE Using Tree Cover

For constructing a Dynamic SSE, we need to support modifications in the database. Modification may take place in two ways. One being the addition of new documents in the database and the other being updating an existing document. In the context of SSE, a modification to the database is recorded by adding or removing the corresponding keyword identifier pair involved in the modification. In the context of the tree cover based SSE scheme, we will still represent $\mathsf{db}(w)$ as a tree, and thus adding a new document would result in adding an extra leaf node to the existing tree, which in some cases can only be done by increasing the height of the tree. Modification of an existing document would be achieved by assigning or altering the sign of the leaf node associated with the identifiers that were affected by the update operation.

$\mathsf{d\Sigma.Update}(k, \sigma_\mathsf{C}, \mathsf{ts}, \mathsf{B}, (\mathsf{op}, \mathsf{in}))$:

Client Side:

if $\mathsf{B} \neq \mathsf{FULL}$
  $\mathsf{B} \leftarrow \mathsf{BufferUpdate}(\mathsf{op}, \mathsf{in}, \mathsf{B})$
if $\mathsf{B} = \mathsf{FULL}$
  set $\mathsf{UList} \leftarrow \emptyset$
  for each w such that $(\mathsf{op}, (\mathsf{id}, w)) \in \mathsf{B}$
    set $L_w \leftarrow \emptyset$
    let m_w be the largest integer such that $(\mathsf{op}, (\mathsf{id}_{m_w}, w)) \in \mathsf{B}$
    let h_w bet the smallest integer such that $m_w \leq 2^{h_w}$
    initialize an empty full binary tree $\mathcal{T}_w$ of height h_w
    define $\mathcal{P} = \{\varphi^{-1}(\mathsf{id}) : (\mathsf{add}, (\mathsf{id}, w)) \in \mathsf{B}\}$
    define $\mathcal{R} = \{\varphi^{-1}(\mathsf{id}) : (\mathsf{del}, (\mathsf{id}, w)) \in \mathsf{B}\}$
    if $\mathsf{ts}_w = \perp$
      $\mathsf{ts}_w \leftarrow 0$
      for all $p \in \mathcal{P}$
        $L_w \leftarrow L_w \cup \{(p, +)\}$
      for all $r \in \mathsf{leaves}(\mathcal{T}_w) \setminus \mathcal{P}$
        $L_w \leftarrow L_w \cup \{(r, -)\}$
      $(h_w, C_w) \leftarrow \Psi_d.\mathsf{CoverGen}(L_w)$
    else
      $\mathsf{ts}_w \leftarrow \mathsf{ts}_w + 1$
      for all $p \in \mathcal{P}$
        $L_w \leftarrow L_w \cup \{(p, +)\}$
      for all $r \in \mathcal{R}$
      $L_w \leftarrow L_w \cup \{(r, -)\}$
    $C_w \leftarrow \Psi_d.\mathsf{CoverGen}(h_w, L_w)$
    for each $(c, s) \in C_w$
      $\widetilde{\mathsf{id}} = (c, s, h_w, \mathsf{ts}_w)$
      $(\sigma_\mathsf{C}, \mathsf{utoken}) \leftarrow \Sigma.\mathsf{Update}_\mathsf{C}\left(k, \sigma_\mathsf{C}, \mathsf{add}, (\widetilde{\mathsf{id}}, w)\right)$
      $\mathsf{UList} \leftarrow \mathsf{UList} \cup \{\mathsf{utoken}\}$
  send UList to server

Server Side:

for all $\mathsf{utoken} \in \mathsf{UList}$
  $\mathsf{EDB} \leftarrow \Sigma.\mathsf{Update}_\mathsf{S}(\mathsf{utoken}, \mathsf{EDB})$

Fig. 15. A generic Dynamic SSE using Tree-Cover scheme: Update protocol

As before we take a secure Dynamic SSE scheme $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\mathsf{Search}, \Sigma.\mathsf{Update})$ as our base scheme and convert it into a tree cover based Dynamic SSE $\mathsf{d\Sigma} = (\mathsf{d\Sigma.Setup}, \mathsf{d\Sigma.Search}, \mathsf{d\Sigma.Update})$. The procedures for $\mathsf{d\Sigma.Setup}$ are shown in Figures 13. $\mathsf{d\Sigma.Update}$ is shown in Figures 14 and 15. $\mathsf{d\Sigma.Search}$ is shown in Figure 16.

$\mathsf{d\Sigma.Setup}$ described in Figure 13 is very similar to $\mathsf{s\Sigma.Setup}$. In $\mathsf{s\Sigma.Setup}$ a given database DB is converted to $\widetilde{\mathsf{DB}}$, where $\widetilde{\mathsf{DB}}$ consists of tuples of the form (w, c, h_w) where c is an element of the cover corresponding to the keyword w and h_w is the height of the corresponding tree representing the set $\mathsf{db}(w)$. In the case of $\mathsf{d\Sigma.Setup}$ the initial database DB is converted into $\widetilde{\mathsf{DB}}$, but in this case, the $\widetilde{\mathsf{DB}}$ consists of tuples of the form $(w, c, h_w, \mathsf{ts}_w)$, where ts_w is a new variable associated with each keyword, which keeps information about the time at which some identifiers related to w have been updated. Further, we'll call ts_w as the time stamp for the keyword w. In the setup phase, for each keyword w, ts_w is set to zero, signifying that no update has taken place yet. The role of this variable ts_w will be more clear from the update operation which we describe next.

Updates, in our case, take place by adding or deleting keyword-identifier pairs. Our main strategy for an update is a *lazy update* model. We assume a buffer memory B of restricted size (which may be user defined) at the client's side to store intermediate updates. The client uses this buffer B to store a "few" keyword-identifier pairs in unencrypted form along with the corresponding operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, where add and del represents addition and deletion respectively. Once the buffer is full, the client uploads the contents of the buffer to the server and resets the buffer to empty. The update procedure consists of procedures to update the buffer and procedures to upload the contents of the buffer to the server.

The update procedure shown in Figure 15 takes as input $(\mathsf{op}, \mathsf{in})$, where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ and in is a keyword-identifier pair (id, w). On receiving the input the client first checks if $(\overline{\mathsf{op}}, \mathsf{in}) \in \mathsf{B}$ or not, where $\overline{\mathsf{op}} = \mathsf{del}$ if $\mathsf{op} = \mathsf{add}$ and vice versa. If $(\overline{\mathsf{op}}, \mathsf{in}) \in \mathsf{B}$, then the client deletes $(\overline{\mathsf{op}}, \mathsf{in})$ from B and add $(\mathsf{op}, \mathsf{in})$ to the buffer B. Otherwise, it only adds $(\mathsf{op}, \mathsf{in})$ to the buffer B. This procedure is summarized in the procedure $\mathsf{BufferUpdate}$ shown in Figure 14. It is important to note that if $(\overline{\mathsf{op}}, \mathsf{in})$ is present in the buffer, then only deleting $(\overline{\mathsf{op}}, \mathsf{in})$ from the buffer does not suffice. It is also necessary to add $(\mathsf{op}, \mathsf{in})$ to the buffer B as the client has no knowledge of the current contents of the server, in particular, it is not possible for the client to know during the update process if $(\overline{\mathsf{op}}, \mathsf{in})$ is currently present in the server or not. For example, assume that for an identifier id the client wants to delete a keyword w, i.e., $\mathsf{op} = \mathsf{del}$. Also assume that $(\mathsf{add}, (\mathsf{id}, w)) \in \mathsf{B}$. If (id, w) currently resides in the server, then deleting $(\mathsf{add}, (\mathsf{id}, w))$ from buffer and not adding $(\mathsf{del}, (\mathsf{id}, w))$ to B would lead to a wrong configuration for the keyword w.

After the buffer B becomes full the client retrieves all the entries $(\mathsf{op}, \mathsf{in}) \in \mathsf{B}$ that correspond to each keyword w. Let $\mathsf{Udt}(w) = \{(\mathsf{op}, \mathsf{in}) \in \mathsf{B} : \mathsf{in} = (\mathsf{id}, w)\}$. The set $\mathsf{Udt}(w)$ gives all identifiers corresponding to w which were updated in the current phase. The client creates a tree $\mathcal{T}$ whose leaf nodes based on the identifiers present in $\mathsf{Udt}(w)$, the identifiers associated with the operation add are labeled $+$ and the ones associated with del are labeled with $-$. This labeling gives a configuration L_w of the tree $\mathcal{T}$. This configuration along with the height of the tree is fed to a dynamic cover generation algorithm $\Psi_d.\mathsf{dCoverGen}$, which yields a cover C_w for the configuration L_w. C_w consists of pairs (i, s) where i is a node of the three $\mathcal{T}$ and $s \in \{+, -\}$. The update operation uploads a λ bit representation of the tuple $(i, s, h_w, \mathsf{ts}_w)$ to the server by creating an update token for the string through the client side update procedure of the base SSE, i.e., through $\Sigma.\mathsf{Update}_\mathsf{C}$. The details are depicted in the Algorithm shown in Figure 15.

The search protocol is summarised in Figure 16. To perform a search query on w, the client uses the $\Sigma.\mathsf{Search}_\mathsf{C}$ protocol of the base SSE to search for the keyword w and obtain the search token stoken_w. This token is then sent to the server. Upon receiving the search token, the server returns the encrypted search result $\mathsf{res}_\mathsf{S} \leftarrow \Sigma.\mathsf{Search}_\mathsf{S}(\mathsf{stoken}, \mathsf{EDB})$, which the client decrypts. Each element of the decrypted search result res is an encoding of (c, s, h, ts_w). The client generates a sequence of covers $\{(h_t, C_t)\}_{t \in [0, \mathsf{ts}_w - 1]}$, where

$$C_t = \{(c, s) : (c, s, h, \mathsf{ts}_w) \in \mathsf{res} \text{ and } , \mathsf{ts}_w = t\},$$

for all $t \in [0, \mathsf{ts}_w - 1]$. It is important to note that for every update (that is when the buffer is full and offloaded to the server), the timestamp and corresponding height h related to all the updates of a particular keyword are the same. The client then feeds this sequence of covers to the dynamic cover reconstruction algorithm $\Psi_d.\mathsf{dReConstruct}$ (Section 4.3). The final configuration produced by the $\Psi_d.\mathsf{dReConstruct}$ algorithm is used to construct $\mathsf{db}(w)$. For computing the final search result the client looks for w in the buffer B and denotes the search result as res_C. The entries in B have the format $(\mathsf{op}, (\mathsf{id}, w))$. If $(\mathsf{del}, (\mathsf{id}, w)) \in \mathsf{res}_\mathsf{C}$ then client discards the id from $\mathsf{db}(w)$. Otherwise, if $(\mathsf{add}, (\mathsf{id}, w)) \in \mathsf{res}_\mathsf{C}$ then the client

$\mathsf{d\Sigma.Search}\,(k, \sigma_C, \mathsf{ts}, \mathsf{B}, w)$:

Round 1:

01. Generate $(\sigma_C, \mathsf{stoken}_w) \leftarrow \Sigma.\mathsf{Search}_C\,(k, \sigma_C, w)$
02. Send stoken_w to server
03. Server returns $\mathsf{res}_S \leftarrow \Sigma.\mathsf{Search}_S\,(\mathsf{stoken}, \mathsf{EDB})$
04. Parse all $r \in \mathsf{res}_S$ as (c, s, h, i)
05. **for** $j \in [0, \mathsf{ts}_w - 1]$
06. Initialize $C_j \leftarrow \emptyset$
07. $C_j \leftarrow C_j \cup \{(c, s)\}$, and $h_j \leftarrow h$ for all $(c, s, h, j) \in \mathsf{res}_S$
08. Generate the final configuration $L \leftarrow \Psi_d.\mathsf{dReConstruct}\left(\{h_j, C_j\}_{j \in [0, \mathsf{ts}_w - 1]}\right)$
09. Generate $\mathsf{db}(w)$ from L using φ
10. Search in B with keyword w and create the set res_C from the search result
11. **for** all $(\mathsf{op}, (\mathsf{id}, w)) \in \mathsf{res}_C$
12. **if** $\mathsf{op} = \mathsf{add}$
13. $\mathsf{db}(w) \leftarrow \mathsf{db}(w) \cup \{\mathsf{id}\}$
14. **else**
15. $\mathsf{db}(w) \leftarrow \mathsf{db}(w) \setminus \{\mathsf{id}\}$

Round 2:

Client Side:

16. Set $\mathsf{UList} \leftarrow \emptyset$
17. Set $\mathsf{ts}_w \leftarrow 0$
18. $\widetilde{\mathsf{db}}(w) \leftarrow \mathsf{Conversion}(\mathsf{db}(w))$
19. **for** all $\mathsf{id} \in \widetilde{\mathsf{db}}(w)$
20. $\widetilde{\mathsf{id}} \leftarrow (\mathsf{id}, \mathsf{ts}_w)$
21. $(\sigma_C, \mathsf{utoken}) \leftarrow \Sigma.\mathsf{Update}_C\left(k, \sigma_C, \mathsf{add}, (\widetilde{\mathsf{id}}, w)\right)$
22. $\mathsf{UList} \leftarrow \mathsf{UList} \cup \{\mathsf{utoken}\}$
23. Send UList to server

Server Side:

24. **for** all $\mathsf{utoken} \in \mathsf{UList}$
25. $\mathsf{EDB} \leftarrow \Sigma.\mathsf{Update}_S(\mathsf{utoken}, \mathsf{EDB})$

Fig. 16. A generic Dynamic SSE using Tree-Cover scheme: Search protocol

adds id to $\mathsf{db}(w)$. Subsequently, it resets ts_w to 0. The client then re-uploads the search result for w, in a manner similar to the update phase discussed earlier.

7 Discussions

In this section we discuss some existing SSEs and the consequences of pairing our scheme with them.

Consider a Dynamic SSE scheme applied to an initially empty database. At a certain instance of time, let i_w be the number of additions, and d_w be the number of deletions that has taken place for a keyword w. Thus, the total number of updates u_w for the keyword w is given by $u_w = i_w + d_w$, and $n_w = i_w - d_w$ denote the number of identifier pairs currently matching keyword w. In addition, let N be the total number of document identifier pairs in the database at that instance. In Table 1 we summarize the characteristics of some widely studied recent SSE schemes. The list does not pretend to be a complete one but is a good representative of the existing SSE schemes. The Table has two major columns named **Computation Cost** and **Communication Cost**. The two sub-columns under Computation Cost report the asymptotic computation cost for search and update, respectively. The three sub-columns under Communication Cost list the size of the result

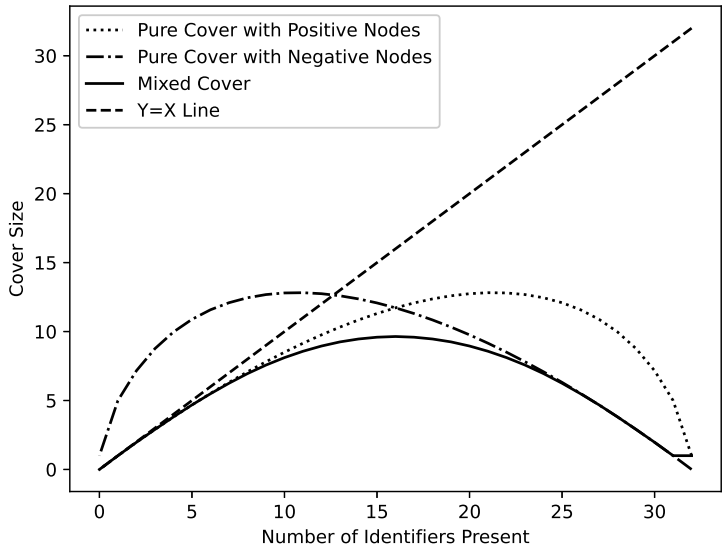


Fig. 17. Number of identifiers vs average cover size

of a single keyword search, the size of an update token for a single update and the number of round-trips (RT), i.e., the number of communication rounds necessary between the server and client for a search.

The schemes reported in Table 1 can be naturally grouped into two groups as follows:

Group-1: The schemes whose search cost is $\mathcal{O}(u_w)$. The first four entries of Table 1, i.e., Fides, Mitra, Π_{BP} and $\mathsf{Diana}_{\mathsf{del}}$ falls under this category.
Group-2: The schemes whose search cost is not $\mathcal{O}(u_w)$ but is dominated by n_w. The last five entries of Table 1 fall under this group.

SSE protocols that achieve a $\mathcal{O}(n_w)$ time complexity for search are called optimal search protocols. The *Group-2* schemes are near optimal, as the leading term in the search cost of these schemes is dominated by n_w.

The *Group-1* schemes fail to achieve the optimal search complexity as they treat deletion also as an insertion with a specific tag and thus the search time depends on the number of updates and not on the number of documents currently in the database that contains w. But the *Group-2* schemes achieve near optimal search time at an increased cost for updates. All *Group-2* schemes except Janus and LLSE have an update cost of $\mathcal{O}(\log N)$ whereas most *Group-1* schemes have a constant update cost.

A similar pattern is observed in the case of communication costs also. The size of the search results of all schemes in *Group-2* is dominated by n_w, but this is achieved with an increased number of communication rounds. Most *Group-1* schemes require a small constant number of communication rounds, but the response size for a search query is $\mathcal{O}(u_w)$.

Effect of our Preprocessing on Efficiency: Our proposed scheme is just a preprocessing step which can be applied to all existing SSEs. For a concrete understanding, we can consider the effect of our pre-processing step on the schemes listed in Table 1. Firstly, if the tree cover scheme is paired with any of the listed schemes, the asymptotic complexity of the schemes does not change. For each w, our scheme deals with $\widetilde{\mathsf{db}}(w)$ instead of $\mathsf{db}(w)$, hence the parameters of interest on which the complexity is measured in all the listed schemes will change. In particular, with our scheme the parameter $n_w = |\mathsf{db}(w)|$

should be replaced by $\widetilde{n_w} = |\widetilde{\mathsf{db}}(w)|$ and $N = |\mathsf{DB}|$ should be replaced by $\widetilde{N} = |\widetilde{\mathsf{DB}}|$. We have already amply argued that on average for any database we will have $\widetilde{n_w} < n_w$ and $\widetilde{N} < N$. With this, it is easy to see that on average each of the *Group-2* schemes will have a concrete reduction of both computation and communication costs.

The effect of our scheme in the case of the *Group-1* schemes is similar. In these schemes, the search cost grows linearly with the number of updates u_w. These schemes consider each update, either insertion or deletion, as a new keyword document pair and these are stored in the database. Thus, the number of keyword document pairs currently in the database is u_w, and this leads to the linear dependence of the search time with the number of updates. But, if paired with the tree cover scheme, the effective number of updates that are to be stored will get reduced on average as instead of the keyword identifier pairs the cover of the configuration related to those pairs will be stored and this would incur lesser cost. Based on the same argument, there would be a concrete reduction of the response sizes on average. Moreover, the constant update cost and the constant update token sizes, as achieved by these schemes, would be retained if paired with the tree cover scheme.

Scheme	**Computation Cost**		**Communication Cost**		
	Search	Update	Search	Update	RT
Fides [7]	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	2
Mitra [12]	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	1
Π_{BP} [15]	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	2
$\mathsf{Diana}_{\mathsf{del}}$ [7]	$\mathcal{O}(u_w)$	$\mathcal{O}(\log u_w)$	$\mathcal{O}(n_w + d_w \log u_w)$	$\mathcal{O}(1)$	2
Janus [7]	$\mathcal{O}(n_w \cdot d_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	1
QOS [17]	$\mathcal{O}(n_w \log i_w + \log^2 \lvert\mathcal{W}\rvert)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^3 N)$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$
Orion [12]	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$
Horus [12]	$\mathcal{O}(n_w \log(d_w) \log N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$
LLSE [11]	$\mathcal{O}((n_w + \log i_w) \cdot \log \log N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w)$	$\mathcal{O}(\log^2 N)$	1

Table 1. Characteristics of some existing SSE schemes: N is the number of (id, w) pairs, $|\mathcal{W}|$ is the number of distinct keywords. For each keyword w, i_w and d_w are the number of insertions and deletions, and $u_w = i_w + d_w$ is the total number of updates, and $n_w = i_w - d_w$ is the number of keyword-identifier pairs currently matching the keyword w. RT is the number of roundtrips for a search query.

8 Security of $\mathsf{d}\Sigma$

As already stated, our scheme $\mathsf{d}\Sigma$ acts as key-less pre-processing step on a base SSE scheme Σ. Thus, if our scheme is used as a preprocessing over an SSE Σ, the resulting scheme would inherit the security of Σ. In this section, we formalize this intuition.

The security of a Dynamic SSE Σ is determined by a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Update}})$. $\mathcal{L}$ denote the information that the adversary learns from the setup process and each execution of the search and update protocols. The security of SSE schemes is generally argued by showing that an adversary can not distinguish between a real-world execution and an ideal-world execution (simulated using the leakage function) of the scheme [16,23,9].

Definition 10 (Adaptive security of DSSE). *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a Dynamic SSE scheme with security parameter* λ *and leakage profile* $\mathcal{L}$*. Then, for any probabilistic polynomial time* (ppt) *adversary* $\mathcal{A}$ *and a simulator* $\mathcal{S}$ *with access to* $\mathcal{L}$*, we define the experiments* $\mathsf{SSEReal}^{\Sigma}_{\mathcal{A}}(\lambda)$*, and* $\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda)$ *as follows.*

$\underline{\mathsf{SSEReal}^{\Sigma}_{\mathcal{A}}(\lambda):}$ *The adversary* $\mathcal{A}(1^{\lambda})$ *chooses a* DB *and the challenger runs* $(\sigma_{\mathsf{C}},\ \mathsf{EDB}) \leftarrow \overline{\mathsf{Setup}_{\mathsf{C}}\left(1^{\lambda}, \mathsf{DB}\right)}$ *and returns* EDB *to* $\mathcal{A}$*. Then for subsequent search or update queries, the challenger runs the real protocols, i.e.,* $(\sigma_{\mathsf{C}}, \mathsf{stoken}) \leftarrow \mathsf{Search}_{\mathsf{C}}(\sigma_{\mathsf{C}}, q)$ *or* $(\sigma_{\mathsf{C}}, \mathsf{utoken}) \leftarrow \mathsf{Update}_{\mathsf{C}}(\sigma_{\mathsf{C}},\ \mathsf{op}, (\mathsf{id}, w))$ *respectively and provides the adversary with* stoken *or* utoken*. The adversary* $\mathcal{A}$ *stops by outputting a bit, which is the output of the experiment.*

$\underline{\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda):}$ *The adversary* $\mathcal{A}\left(1^{\lambda}\right)$ *chooses a* DB *and the challenger runs* $\mathsf{EDB} \leftarrow \overline{\mathcal{S}\left(\mathcal{L}_{\mathsf{Setup}}\right)}$ *and returns* EDB *to* $\mathcal{A}$*. Then for subsequent search or update queries, the challenger runs* $\mathsf{stoken} \leftarrow \mathcal{S}\left(\mathcal{L}_{\mathsf{Search}}\right)$ *or* $\mathsf{utoken} \leftarrow \mathcal{S}\left(\mathcal{L}_{\mathsf{Update}}\right)$ *respectively and provides the adversary with* stoken *or* utoken*. The adversary* $\mathcal{A}$ *stops by outputting a bit, which is the output of the experiment.*

A Dynamic SSE scheme Σ *is said to be* $\mathcal{L}$*-adaptively secure if for all adversaries* $\mathcal{A}$*, the exists a simulator* $\mathcal{S}$ *such that, the SSE advantage* $\left(\mathsf{Adv}^{\Sigma}_{\mathcal{A},\mathcal{S}}\right)$ *of* $\mathcal{A}$ *(defined below) is a negligible function in* λ*.*

$$\mathsf{Adv}^{\mathsf{Priv},\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda) = \left|\Pr\left[\mathsf{SSEReal}^{\Sigma}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

We reduce the security of our dynamic scheme $\mathsf{d}\Sigma$ to the security of the base SSE scheme Σ.

Theorem 2. *Let* $\mathsf{d}\Sigma = (\mathsf{d}\Sigma.\mathsf{Setup}, \mathsf{d}\Sigma.\mathsf{Search}, \mathsf{d}\Sigma.\mathsf{Update})$ *be a Dynamic SSE scheme as described in Figures 13, 15 and 16.* $\mathsf{d}\Sigma$ *is instantiated with a fixed but arbitrary tree cover scheme* $\Psi = (\Psi.\mathsf{CoverGen}, \Psi.\ \mathsf{ReConstruct})$ *and a base dynamic SSE scheme* Σ *with leakage profile* $\mathcal{L}$*. If* Σ *is* $\mathcal{L}$*-adaptive secure, then* $\mathsf{d}\Sigma$ *is also* $\mathcal{L}$*-adaptive secure.*

We provide the proof of the above theorem in the Appendix B. A few more points regarding the security of $\mathsf{d}\Sigma$ are to be noted:

1. Our preprocessing step does not add to the security of the base scheme. The tree cover scheme is meant to enhance the efficiency of the scheme while restoring the security of the base scheme.
2. Theorem 2 only asserts a basic security guarantee of our scheme. We assumed the base scheme to be $\mathcal{L}$-adaptive secure which is a well accepted model of security. But security of SSE schemes is still an active area of research and is not fully understood. It has been claimed that SSE schemes proven secure in the $\mathcal{L}$-adaptive model may still succumb to attacks that the security model fails to incorporate. For example, a class of attacks called file injection attacks [34] or a recent generalization in [2] may still be applicable to provably secure schemes. Such weaknesses of the base SSE scheme Σ may affect the security of $\mathsf{d}\Sigma$.
3. A similar security Theorem holds for our static scheme $\mathsf{s}\Sigma$.

9 Experimental Results

9.1 Comparing Cover Sizes

Consider a database with n documents, where $n = 2^k$ for some k and let for a keyword w, $|\mathsf{db}(w)| = r$. A tree representing this keyword w will contain n leaves, of which r leaves would be labeled $+$ and $n - r$ leaves would be labeled $-$. We are interested in finding the cover size of such a keyword. Note that if we fix the database size to n and $|\mathsf{db}(w)|$ to r, then the tree for w may have $\binom{n}{r}$ different configurations, and each configuration will give rise to a cover of a different size. In our first experiment, we keep n fixed to 32 and for each $0 \leq r \leq 32$ we generate $\binom{n}{r}$ configurations and compute the cover of all these configurations. For each configuration, we generate three types of covers:

1. Pure cover with the nodes labeled +, by using the algorithm $\mathsf{PureCoverGen}(\cdot,\cdot)$ described in Figure 3.
2. Pure cover with the nodes labeled −, also using algorithm $\mathsf{PureCoverGen}(\cdot,\cdot)$ of Figure 3.
3. Mixed cover using Algorithm $\mathsf{MixedCoverGen}(\cdot,\cdot)$ described in Figure 7.

For each r, we compute the average cover size over all the configurations. These results are summarized in Figure 17. The X-axis of the graph shown in Figure 17 represents the value $r = |\mathsf{db}(w)|$, i.e., the number of identifiers which are present corresponding to the keyword w. The Y-axis represents the average cover size of the configurations. The size of the three types of covers is represented by three different types of lines. In addition, we have also plotted the line $Y = X$ for the sake of reference. Note that the line $Y = X$ shows how the input size corresponding to a normal SSE (which is just the number of identifiers present in the database corresponding to w, i.e., $|\mathsf{db}(w)|$) will grow with the increase in the number of identifiers r.

The following can be immediately observed from the results demonstrated in Figure 17:

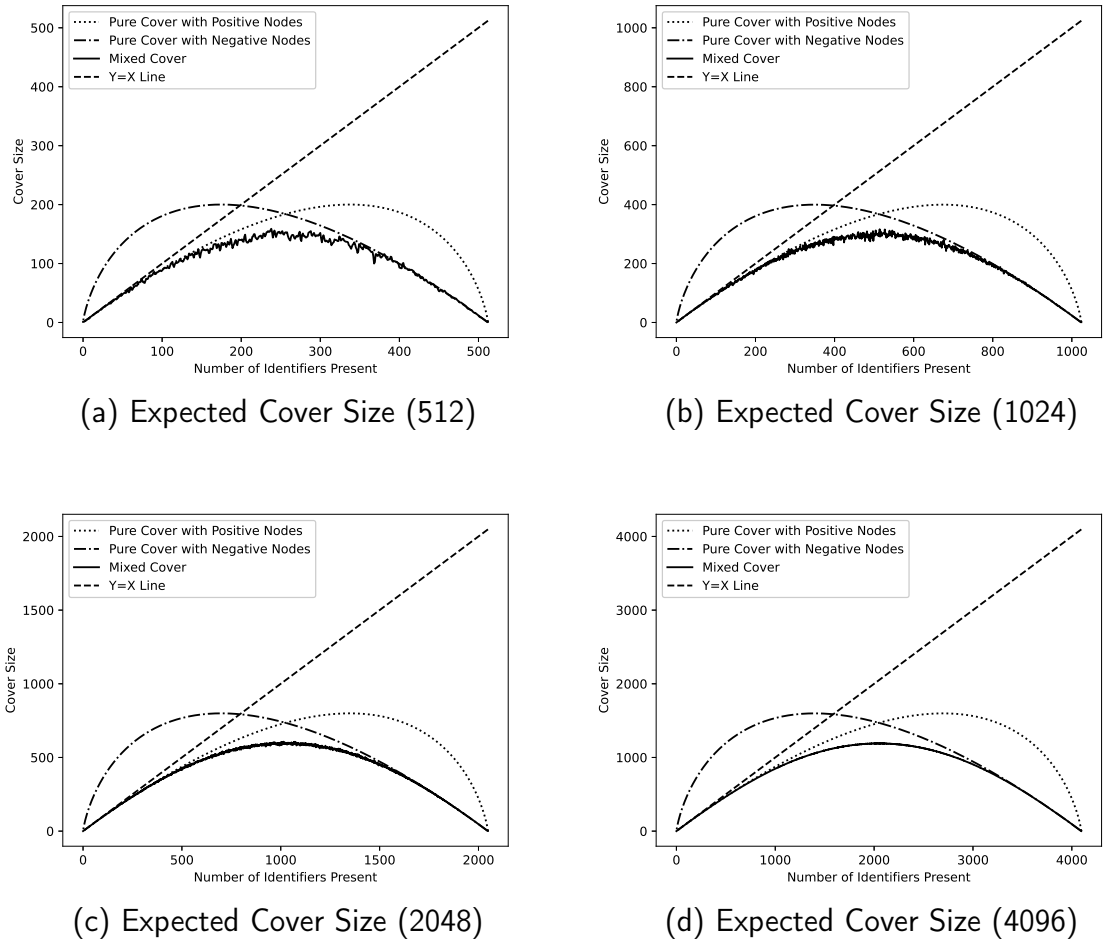
1. For all three types of covers, the average size of the covers increases with r, and then it decreases.
2. For all values of r, the size of the mixed cover is smaller than the size of the pure covers. The difference is significant when the number of identifiers present is around half of the total number of identifiers.
3. The curves representing the average size of pure cover with nodes labeled + and the mixed cover always lies below the line $Y = X$, signifying that on average, using covers as input to an SSE would give rise to savings compared to using just the set $\mathsf{db}(w) \times \{w\}$.

We have theoretically calculated the expected cover size for pure configuration (see Section 4.1), given by Equation (3). To validate our theoretical result (Equation (3)), we ran the pure cover generation algorithm for all possible configurations of the set $\mathsf{db}(w)$ in a tree with 32 leaf nodes. The experimental results for the expected cover size match very closely with the values predicted by the expression (Equation 3), confirming the tightness of our expression.

Next, we repeat the same experiment with different database sizes for $n = 512, 1024$, 2048, and 4096. In this setting, it would be computationally prohibitive to compute the cover size of all configurations corresponding to a value of r, $0 \leq r \leq n$. Hence, for each r, we generate 5000 uniform random configurations (with replacement), then compute their mixed cover and the average cover size over these 5000 configurations. The results of this experiment are depicted in Figure 18, which shows the variation of the average cover size with r for different values of n. In each figure, the line $Y = X$ is also drawn for reference. For pure covers, the curves were drawn using Equation (3). The results in Figure 18 show the same pattern as in Figure 17.

9.2 Performance on a Real Data

To validate our proposed scheme in practical databases, we conducted an experiment using the Enron Email Dataset [18], which has about 500,000 documents and 200,000 keywords. We began by extracting keywords from the data set and construction of DB consisting of the keyword identifier pairs. Subsequently, using the algorithm in Figure 11 we converted DB to $\widetilde{\mathsf{DB}}$. We experimented on different sizes of the database by randomly selecting subsets of the database and for each case we compared the sizes of the original and converted versions. These results are shown in Table 2, and for easy visual comparison



(a) Expected Cover Size (512)

(b) Expected Cover Size (1024)

(c) Expected Cover Size (2048)

(d) Expected Cover Size (4096)

Fig. 18. Expected cover size comparison

a pictorial representation of the data in Table 2 is shown in Figure 19. The last column of Table 2 computes $\frac{(|\mathsf{DB}|-|\widetilde{\mathsf{DB}}|)\times 100}{|\mathsf{DB}|}$. It is evident from Table 2 that our protocol results in substantial reductions in database size, ranging from 60% to 35%, even when dealing with reasonably large databases. This demonstrates the effectiveness of our approach in practical databases.

$\lvert\mathsf{DB}\rvert$	$\lvert\widetilde{\mathsf{DB}}\rvert$	Advantage (%)
2^{20}	5,26,492	52.4
$2^{20.5}$	6,08,347	60.2
2^{21}	9,24,455	54.8
$2^{21.5}$	15,50,264	49.8
2^{22}	29,74,256	43.8
$2^{22.5}$	44,05,133	41.1
2^{23}	64,47,481	36.4
$2^{23.5}$	75,17,379	35.6

Table 2. The size of original database DB and the converted database $\widetilde{\mathsf{DB}}$ in case of Enron data

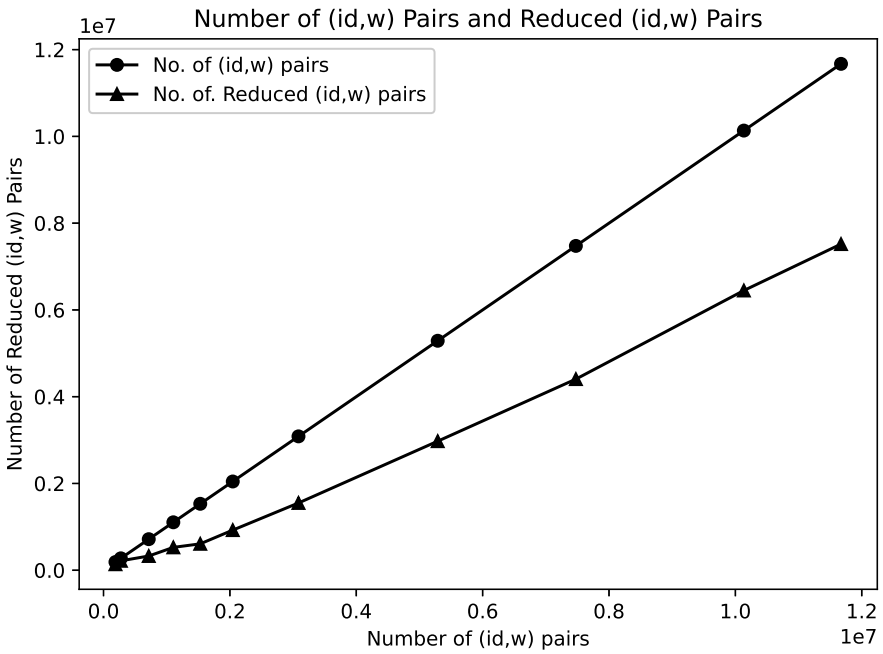


Fig. 19. A pictorial depiction of the data in Table 2 showing the number of tuples in the original database (represented by filled circles) and the number of tuples in the converted database (represented by filled triangles) for different database sizes

9.3 Extra Overheads

Our scheme builds over an existing SSE and the extra overhead over the original SSE is the time required for computing covers and cover reconstruction. We experimentally compute the time required for cover generation and computation. We used the following configuration for our computation:

CPU: Intel Core i5-1135G7 @ 2.40GHz × 8 processor (3.1GHz).
RAM: 16 GB
OS: Ubuntu 22.04.3 LTS, 64 bits.
Programming Language: Python

Table 3 shows the time required for cover generation and reconstruction for databases of different sizes. We considered database sizes of $n = 2^{17}, 2^{18}, 2^{19}$. For each of these databases, we considered different sizes of $|\mathsf{db}(w)|$ as shown in the rows of the Table. For each $|\mathsf{db}(w)|$ we generated 100 random configurations and the times reported for the cover generation and re-construction are the average time required for generation and reconstruction for these 100 configurations. As expected, Table 3 clearly shows that the cover generation times increase with both the increase in n and $|\mathsf{db}(w)|$. The reconstruction times reported for $|\mathsf{db}(w)| < 10^4$, are negligible. If $|\mathsf{db}(w)|$ is small, it is expected that a pure cover is generated, and reconstruction of a pure cover is immediate.

Dynamic Scenario: Now, we test the performance of our scheme for updates. We consider a database containing 10^9 keyword-identifier pairs, and a client buffer of size 0.01% of the actual database size. We consider updates for a single keyword w following the protocol described below.

We consider an initial set of documents $\mathcal{D}_0$ and a keyword w^* where w^* is initially present in 12.5% of the documents in $\mathcal{D}_0$. Next, we perform a series of update operations involving w^*. The updates are made in three phases P_1, P_2, P_3. The updates in phase P_i are applied to the documents in $\mathcal{D}_{i-1}$, and the updates result in a new set of

$\lvert \mathsf{db}(w) \rvert$	Time (sec) for $n = 2^{17}$		Time (sec) for $n = 2^{18}$		Time (sec) for $n = 2^{19}$	
	Gen	Re-con	Gen	Re-con	Gen	Re-con
10^2	0.22	10^{-5}	0.42	10^{-5}	0.94	10^{-5}
10^3	0.27	10^{-4}	0.49	10^{-4}	1.01	10^{-4}
10^4	0.51	10^{-3}	0.88	10^{-3}	1.51	10^{-3}
2^{16}	0.87	0.17	1.64	0.25	2.97	0.35
2^{17}	-	-	1.99	0.37	3.69	0.57
2^{18}	-	-	-	-	4.14	0.85

Table 3. Cover generation and reconstruction times

documents $\mathcal{D}_i$. The update operations are designed in such a way that w^* is present in 25% of the documents in $\mathcal{D}_1$, 50% of the documents in $\mathcal{D}_2$ and 60% documents in $\mathcal{D}_3$. In each phase, multiple update operations are performed and 10% of all operations in each phase are delete operations. At the end of each phase of updates, a search operation is performed involving w^*. Note, that the search operation forces the transmission of all tuples corresponding to w^* from the buffer to the server.

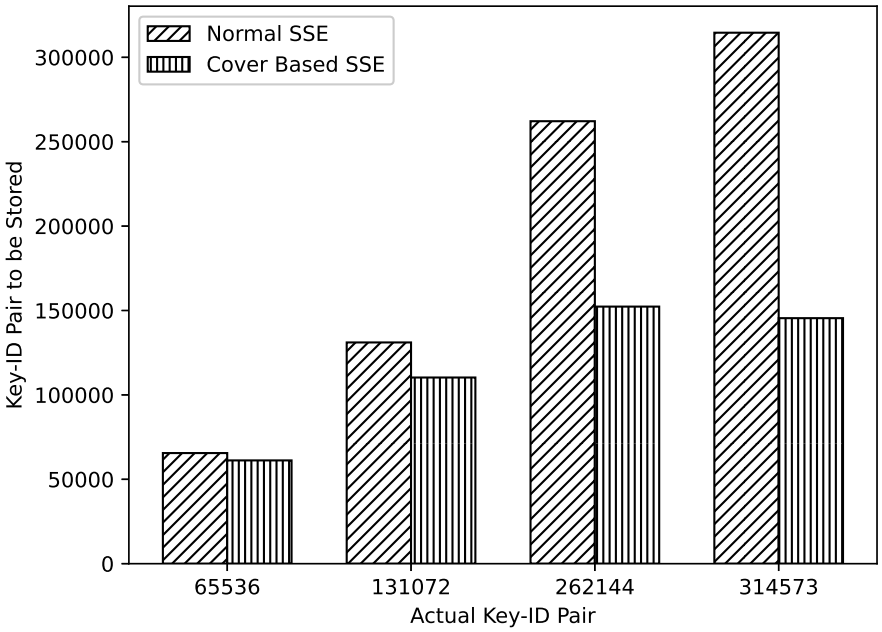


Fig. 20. Database sizes in dynamic scenario.

We repeated the above protocol 2000 times with random updates following the rules stated above using our scheme $\mathsf{d}\Sigma$. After each execution, we recorded the size of the database after the search operations in each phase. In Figure 20 we report the average size of the database across the 2000 executions in a bar diagram. In Figure 20 the database sizes for our scheme and the base SSE are shown. The four pairs of bars correspond to the sizes of the databases after each phase just after the search operation. The first pair correspond to the database for the initial set of documents $\mathcal{D}_0$, and the second pair is for $\mathcal{D}_1$ etc. The figure clearly shows that our scheme leads to considerable savings in the dynamic scenario also.

10 Conclusion

We describe a novel way to make SSE schemes space efficient. Our scheme converts a given database into a different representation which on average results in a significant amount of space savings. The smaller representation also results in reduced search time and response sizes. Our scheme depends on representing the set $\mathsf{db}(w)$ using binary trees where the leaves are labeled. This representation has interesting combinatorial properties which we explore in detail. Our experiments show that our representation results in smaller index sizes with very low extra overhead. Our scheme can be used with any secure SSE, and our scheme being just a pre-processing step retains the security of the base SSE.

We list some aspects of the tree cover scheme as applied to SSE which we plan to explore in the near future:

1. Our current study involves only single keyword select queries. It seems that with some small modifications, it may be possible to equip the scheme to handle range queries.
2. We compute an estimate of the average size of a cover in the case of pure covers. Our estimate is quite accurate as demonstrated by the experiments, but we failed to obtain a semantically useful analytical expression of the estimate. It would be nice to try to estimate the average size in some alternative way to obtain a more meaningful expression.
3. We would like to see the performance of our scheme in real environment which would require a careful implementation of our scheme paired with a base SSE in a real cloud environment.
4. An attractive property of the scheme is that given the response of a query produced by our scheme it is not possible to directly know the number of documents that match the query. This hints that our scheme is volume hiding to some extent. But, a thorough investigation of this property is required. We think that by using some additional randomization, it may be possible to convert our scheme into a volume hiding scheme against a large class of powerful adversaries.

A Proof of Theorem 1

The proof of (1) is immediate, as at least one of $X_{i,r}$ or $X_{i,g}$ must be non-empty as L_c is a complete configuration of $\mathcal{T}$ and hence all the leaves of $\mathcal{T}$ are colored.

(2) Directly follows from the description of the algorithm and Proposition 4.

We prove (3) by induction on the height of a node. Let i be of height 0, i.e., a leaf node and without loss of generality let i be colored GREEN, thus $X_{i,g} = \{(i, +)\}$ and $X_{i,r} = X_{i,m} = \emptyset$, and $X_{i,g}$ is the smallest set which spans $\mathcal{T}(i)$ as $|X_{i,g}| = 1$. This serves as the base case. As induction hypothesis consider that (3) is true for all nodes at height less than ℓ. Consider, a node i at height ℓ. As $\ell > 0$, i have two children say λ and ρ. By our induction hypothesis, $X_\lambda = \mathsf{minCard}(X_{\lambda,r}, X_{\lambda,g}, X_{\lambda,m})$ and $X_\rho = \mathsf{minCard}(X_{\rho,r}, X_{\rho,g}, X_{\rho,m})$ are the smallest sets which spans $\mathcal{T}(\lambda)$ and $\mathcal{T}(\rho)$ respectively. Now we have a few cases to consider:

Case 1: i is colored. Without loss of generality let $\mathsf{color}(i) =$ GREEN, then $X_{i,g} = \{(i, +)\}$ and $X_{i,r}$ and $X_{i,m}$ are empty. Moreover, as $X_{i,g} = \{(i, +)\}$ contains a single node which is the root of $\mathcal{T}(i)$, thus $X_{i,g}$ is the smallest set which spans $\mathcal{T}(i)$.
Case 2: i is not colored. Let $X = \mathsf{minCard}(C_i)$, and for the sake of contradiction let Y span $\mathcal{T}(i)$ and let $|Y| < |X|$. Let $Y = Y_\lambda \cup Y_\rho$ where Y_j contains nodes in $\mathcal{T}(j)$ for $j \in \{\lambda, \rho\}$. We consider two subcases:

(a) Both Y_λ and Y_ρ are non-empty. By Proposition 5, Y_λ and Y_ρ spans $\mathcal{T}(\lambda)$ and $\mathcal{T}(\rho)$ respectively. Now, we claim that either $|Y_\lambda| < |X_\lambda|$ or $|Y_\rho| < |X_\rho|$, and this contradicts our induction hypothesis. Finally, to see why our claim is correct, notice that by our algorithm $X_\lambda \cup X_\rho \in C_i$, and as $|Y| < |X|$ and $X = \mathsf{minCard}(C_i)$ it cannot be that $|Y_\lambda| \geq |X_\lambda|$ and $|Y_\rho| \geq |X_\rho|$.

(b) One of Y_λ and Y_ρ is empty. Without loss of generality, let $Y_\rho = \emptyset$, i.e., $Y = Y_\lambda$. Then by Proposition 5, Y_λ must contain nodes of the same color, say GREEN, and all leaves of $\mathcal{T}(\rho)$ must be RED. Hence, node ρ must be RED and all nodes in $\mathsf{leaves}(\mathcal{T}(\lambda)) \setminus Y_\lambda$ must also be RED. In this configuration, the set $X_{i,g}$ computed by MixedCoverGen, will contain the same nodes as in Y_λ. Hence, we have $Y = Y_\lambda \in C_i$, which contradicts $|Y| < |X|$, as $X = \mathsf{minCard}(C_i)$.

□

B Proof of Theorem 2

We say that $\mathsf{Sim}_\mathcal{B}$ is a *compatible simulator* for an adversary $\mathcal{B}$ attacking Σ if

$$\left|\Pr\left[\mathsf{SSEReal}^{\Sigma}_{\mathcal{B}}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{B},\mathsf{Sim}_\mathcal{B}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

As Σ is $\mathcal{L}$-adaptive secure hence by Definition 10, a compatible simulator for $\mathcal{B}$ always exists.

Let $\mathcal{A}$ be an arbitrary adversary for $\mathsf{d}\Sigma$ instantiated with a fixed but arbitrary tree cover scheme Ψ and a base dynamic SSE Σ. Based on the protocols involved in $\mathsf{d}\Sigma$ (see Figures 13, 15, 16) it is important to note down the correct interface between the adversary $\mathcal{A}$ and its challenger:

1. On a setup query $(1^\lambda, \mathsf{DB})$ for a database DB of $\mathcal{A}$'s choice its challenger returns EDB.
2. On an update query $(\mathsf{op}, \mathsf{in})$, its challenger returns UList a set of update tokens (see line 30 of Figure 15).
3. For a search query w, the challenger returns a search token stoken_w and a set of update tokens UList (see lines 02 and 23 of Figure 16).

Given an adversary $\mathcal{A}$ for $\mathsf{d}\Sigma$, we construct an adversary $\mathcal{B}$ for the base protocol Σ, which acts as a challenger for $\mathcal{A}$. $\mathcal{B}$ being an adversary for Σ has a challenger which we denote as $\mathcal{C}$. $\mathcal{B}$ provides responses to the queries of $\mathcal{A}$ with the help of the responses it receives from its challenger $\mathcal{C}$. Note the tree cover scheme Ψ is key-less and is thus accessible to $\mathcal{B}$.

Now we describe the interaction of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ in a sequence of games G0, G1, G2.
Game G0: We assume that $\mathcal{B}$'s challenger $\mathcal{C}$ follows the real protocols (Σ.Setup, Σ.Search, Σ.Update) to answer queries of $\mathcal{B}$. $\mathcal{B}$ acts as a challenger to $\mathcal{A}$ as follows:

1. When a setup request $\mathsf{Setup}(1^\lambda, \mathsf{DB})$ is issued by $\mathcal{A}$ then $\mathcal{B}$ runs lines 01 to 03 of the procedure $\mathsf{d}\Sigma.\mathsf{Setup}(1^\lambda, \mathsf{DB})$ described in Figure 13 and obtains $\widetilde{\mathsf{DB}}$. $\mathcal{B}$ then issues the setup request $(1^\lambda, \widetilde{\mathsf{DB}})$ to its challenger $\mathcal{C}$ and receives EDB as a response which it transmits to $\mathcal{A}$. Note, in this process $\mathcal{B}$ has created a map ts and a buffer B which it retains with it and updates during the subsequent queries of $\mathcal{A}$.
2. On receiving an update query $(\mathsf{op}, \mathsf{in})$ from $\mathcal{A}$, $\mathcal{B}$ runs lines 01 to 30 of the procedure $\mathsf{d}\Sigma.\mathsf{Update}$ described in Figure 15. In lieu of line 28, it issues an update query $(\widetilde{\mathsf{id}}, w)$ to its challenger $\mathcal{C}$ and receives utoken in response. It populates the set UList using the responses received from $\mathcal{C}$, then it executes lines 31 and 32 of the procedure $\mathsf{d}\Sigma.\mathsf{Update}$ and finally sends UList to $\mathcal{A}$.

3. On receiving a search query w from $\mathcal{A}$, $\mathcal{B}$ queries $\mathcal{C}$ with w and receives as a response stoken_w. Then it executes lines 03 to 25 of the procedure $\mathsf{d\Sigma.Search}$ as described in Figure 16. Instead of executing line 21, it queries $\mathcal{C}$ with an update query $(\mathsf{add}, (\widetilde{\mathsf{id}}, w))$ and receives utoken as response. Finally, it sends UList as constructed in line 23 to $\mathcal{A}$.

After $\mathcal{A}$ stops querying, $\mathcal{A}$ outputs a bit $b \in \{0,1\}$, $\mathcal{B}$ and $\mathsf{G0}$ also outputs b.

We can view $\mathsf{G0}$ as an interaction between $\mathcal{A}$ and its challenger $\mathcal{B}$, where $\mathcal{A}$ gets a perfect interface for the real scheme $\mathsf{d\Sigma}$, thus

$$\Pr\left[\mathsf{SSEReal}^{\mathsf{d\Sigma}}_{\mathcal{A}}(\lambda) = 1\right] = \Pr[\mathsf{G0} = 1]. \tag{7}$$

We can also view adversaries $\mathcal{A}$ and $\mathcal{B}$ together as a single adversary $\mathcal{B}'$ who interacts with $\mathcal{C}$ (the challenger of $\mathcal{B}$), which provides the perfect interface for Σ. Thus, we have

$$\Pr\left[\mathsf{SSEReal}^{\Sigma}_{\mathcal{B}'}(\lambda) = 1\right] = \Pr[\mathsf{G0} = 1]. \tag{8}$$

Game $\mathsf{G1}$: We make some small changes in Game $\mathsf{G0}$ to obtain game $\mathsf{G1}$. First, observe that for any adversary $\mathcal{A}$ for $\mathsf{d\Sigma}$, Game $\mathsf{G0}$ gives a concrete description for an adversary $\mathcal{B}$, and thus of the combined adversary $\mathcal{B}'$ which attacks Σ. As Σ is $\mathcal{L}$-adaptive secure hence a compatible simulator $\mathsf{Sim}_{\mathcal{B}'}$ for $\mathcal{B}'$ exists. In Game $\mathsf{G1}$ the challenger responds to the queries of $\mathcal{B}'$ using the simulator $\mathsf{Sim}_{\mathcal{B}'}$ instead of the real protocols of Σ. Thus,

$$\Pr\left[\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{B}',\mathsf{Sim}_{\mathcal{B}'}} = 1\right] = \Pr[\mathsf{G1} = 1]. \tag{9}$$

As Σ is $\mathcal{L}$-adaptive secure hence, we have

$$\left|\Pr\left[\mathsf{SSEReal}^{\Sigma}_{\mathcal{B}'}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{B}',\mathsf{Sim}_{\mathcal{B}'}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda). \tag{10}$$

Thus using Equations (8), (9), (10), we have

$$|\Pr[\mathsf{G0} = 1] - \Pr[\mathsf{G1} = 1]| \leq \mathsf{negl}(\lambda). \tag{11}$$

Game $\mathsf{G2}$: In this game we view the game $\mathsf{G1}$ a bit differently. We see the description of $\mathcal{B}$ along with the simulator $\mathsf{Sim}'_{\mathcal{B}}$ together and call it as $\mathsf{Sim}_{\mathcal{A}}$. Note, with this view, we see $\mathcal{A}$ interacting with a single piece of code $\mathsf{Sim}_{\mathcal{A}}$. Thus, we have

$$\Pr[\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{A},\mathsf{Sim}_{\mathcal{A}}}(\lambda)] = \Pr[\mathsf{G2} = 1], \tag{12}$$

and as $\mathsf{G1}$ and $\mathsf{G2}$ are essentially same we have

$$\Pr[\mathsf{G1} = 1] = \Pr[\mathsf{G2} = 1]. \tag{13}$$

Now, using Equations (11) and (13) we have

$$|\Pr[\mathsf{G0} = 1] - \Pr[\mathsf{G2} = 1]| \leq \mathsf{negl}(\lambda). \tag{14}$$

Finally, using Equations (7), (12) and (14), we have

$$\left|\Pr\left[\mathsf{SSEReal}^{\Sigma}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Sigma}_{\mathcal{A},\mathsf{Sim}_{\mathcal{A}}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda),$$

as desired. □

Acknowledgments. A major portion of this work was done when Subhabrata Samajder and Avishek Majumder were in the Computer Science and Engineering department of Indraprastha Institute of Information Technology Delhi (IIIT-Delhi). They thank IIIT-Delhi for the support.

References

1. Tree Cover SSE. https://github.com/sochoavi/IJIS_Code
2. Amjad, G., Kamara, S., Moataz, T.: Injection-secure structured and searchable symmetric encryption. In: Guo, J., Steinfeld, R. (eds.) Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part VI. Lecture Notes in Computer Science, vol. 14443, pp. 232–262. Springer (2023). https://doi.org/10.1007/978-981-99-8736-8_8, https://doi.org/10.1007/978-981-99-8736-8_8
3. Amjad, G., Patel, S., Persiano, G., Yeo, K., Yung, M.: Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. Proc. Priv. Enhancing Technol. **2023**(1), 417–436 (2023). https://doi.org/10.56553/popets-2023-0025, https://doi.org/10.56553/popets-2023-0025
4. Bhattacherjee, S., Sarkar, P.: Complete tree subset difference broadcast encryption scheme and its analysis. Designs, codes and cryptography **66**(1), 335–362 (2013)
5. Bhattacherjee, S., Sarkar, P.: Concrete Analysis and Trade-Offs for the (Complete Tree) Layered Subset Difference Broadcast Encryption Scheme. IEEE Transactions on Computers **63**(7), 1709–1722 (2014), DOI: 10.1109/TC.2013.68
6. Bost, R.: $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1143–1154. ACM (2016). https://doi.org/10.1145/2976749.2978303, https://doi.org/10.1145/2976749.2978303
7. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1465–1482. ACM (2017). https://doi.org/10.1145/3133956.3133980, https://doi.org/10.1145/3133956.3133980
8. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. pp. 668–679 (2015)
9. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS. vol. 14, pp. 23–26. Citeseer (2014)
10. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. pp. 353–373. Springer (2013)
11. Chamani, J.G., Papadopoulos, D., Karbasforushan, M., Demertzis, I.: Dynamic searchable encryption with optimal search in the presence of deletions. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. pp. 2425–2442. USENIX Association (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/chamani
12. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 1038–1055. ACM (2018)
13. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010. pp. 577–594. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
14. Chatterjee, S., Kesarwani, M., Modi, J., Mukherjee, S., Puria, S.K.P., Shah, A.: Secure and efficient wildcard search over encrypted data. Int. J. Inf. Sec. **20**(2), 199–244 (2021). https://doi.org/10.1007/S10207-020-00492-W, https://doi.org/10.1007/s10207-020-00492-w
15. Chatterjee, S., Puria, S.K.P., Shah, A.: Efficient backward private searchable encryption. J. Comput. Secur. **28**(2), 229–267 (2020). https://doi.org/10.3233/JCS-191322, https://doi.org/10.3233/JCS-191322
16. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. p. 79–88. CCS '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1180405.1180417, https://doi.org/10.1145/1180405.1180417
17. Demertzis, I., Chamani, J.G., Papadopoulos, D., Papamanthou, C.: Dynamic searchable encryption with small client storage. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020), https://www.ndss-symposium.org/ndss-paper/dynamic-searchable-encryption-with-small-client-storage/
18. Enron Corp and Cohen, W. W.: Enron email dataset. United States Federal Energy Regulatory Commissioniler, comp (2015), https://www.loc.gov/item/2018487913/
19. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. Proc. Priv. Enhancing Technol. **2018**(1), 5–20 (2018). https://doi.org/10.1515/POPETS-2018-0002, https://doi.org/10.1515/popets-2018-0002

20. Hu, C., Han, L.: Efficient wildcard search over encrypted data. Int. J. Inf. Secur. **15**(5), 539–547 (oct 2016). https://doi.org/10.1007/s10207-015-0302-0, https://doi.org/10.1007/s10207-015-0302-0
21. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012. The Internet Society (2012), https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation
22. Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 183–213. Springer (2019)
23. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. p. 965–976. CCS '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2382196.2382298, https://doi.org/10.1145/2382196.2382298
24. Lai, S., Patranabis, S., Sakzad, A., Liu, J.K., Mukhopadhyay, D., Steinfeld, R., Sun, S.F., Liu, D., Zuo, C.: Result pattern hiding searchable encryption for conjunctive queries. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. pp. 745–762 (2018)
25. Molla, E., Rizomiliotis, P., Gritzalis, S.: Efficient searchable symmetric encryption supporting range queries. Int. J. Inf. Sec. **22**(4), 785–798 (2023). https://doi.org/10.1007/S10207-023-00667-1, https://doi.org/10.1007/s10207-023-00667-1
26. Naor, D., Naor, M., Lotspiech, J.: Revocation and tracing schemes for stateless receivers. In: Kilian, J. (ed.) Advances in Cryptology — CRYPTO 2001. pp. 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
27. Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 79–93 (2019)
28. Patranabis, S., Mukhopadhyay, D.: Forward and backward private conjunctive searchable symmetric encryption. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society (2021), https://www.ndss-symposium.org/ndss-paper/forward-and-backward-private-conjunctive-searchable-symmetric-encryption/
29. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy. p. 44. SP '00, IEEE Computer Society, USA (2000)
30. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014), https://www.ndss-symposium.org/ndss2014/practical-dynamic-searchable-encryption-small-leakage
31. Sun, S., Steinfeld, R., Lai, S., Yuan, X., Sakzad, A., Liu, J.K., Nepal, S., Gu, D.: Practical non-interactive searchable encryption with forward and backward privacy. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society (2021), https://www.ndss-symposium.org/ndss-paper/practical-non-interactive-searchable-encryption-with-forward-and-backward-privacy/
32. Wu, Z., Li, K.: Vbtree: forward secure conjunctive queries over encrypted data for cloud computing. The VLDB journal **28**(1), 25–46 (2019)
33. Yuan, D., Zuo, C., Cui, S., Russello, G.: Result-pattern-hiding conjunctive searchable symmetric encryption with forward and backward privacy. Proc. Priv. Enhancing Technol. **2023**(2), 40–58 (2023). https://doi.org/10.56553/POPETS-2023-0040, https://doi.org/10.56553/popets-2023-0040
34. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: Proceedings of the 25th USENIX Conference on Security Symposium. p. 707–720. SEC'16, USENIX Association, USA (2016)
35. Zuo, C., Sun, S.F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23. pp. 228–246. Springer (2018)
36. Zuo, C., Sun, S.F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption with forward and stronger backward privacy. In: Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II. p. 283–303. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29962-0_14, https://doi.org/10.1007/978-3-030-29962-0_14