Proceedings on Privacy Enhancing Technologies ; 2022 (1):28–48

Jiafan Wang and Sherman S. M. Chow*

Forward and Backward-Secure Range-Searchable Symmetric Encryption

Abstract: Dynamic searchable symmetric encryption (DSSE) allows a client to query or update an outsourced encrypted database. Range queries are commonly needed. Previous range-searchable schemes either do not support updates natively (SIGMOD'16) or use file indexes of many long bit-vectors for distinct keywords, which only support toggling updates via homomorphically flipping the presence bit. (ESORICS'18).
We propose a *generic upgrade* of any (inverted-index) DSSE to support range queries (a.k.a. range DSSE), without homomorphic encryption, and a specific instantiation with *a new trade-off reducing client-side storage.* Our schemes achieve forward security, an important property that mitigates file injection attacks. Moreover, we identify *a variant of injection attacks* against the first somewhat dynamic scheme (ESORICS'18). We also extend the definition of backward security to range DSSE and show that our schemes are compatible with a generic upgrade of backward security (CCS'17).
We comprehensively analyze the computation and communication overheads, including implementation details of *client-side* index-related operations *omitted* by prior schemes. We show high empirical efficiency for million-scale databases over a million-scale keyword space.

Keywords: dynamic symmetric searchable encryption, range, generic construction, saving client storage, attack

DOI 10.2478/popets-2022-0003
Received 2021-05-31; revised 2021-09-15; accepted 2021-09-16.

1 Introduction

Searchable symmetric encryption (SSE) allows a client to issue (keyword) search queries over the encrypted data outsourced to an untrusted server. SSE schemes with different trade-offs of efficiency, security, and functionality have been proposed. Major milestones include dynamic SSE (DSSE) by Kamara *et al.* [19], enabling insertion/deletion over the encrypted database.

Forward/Backward Security. The dynamism inevitably introduces new attack surfaces and thus motivates new security notions [31] – *Forward security* [31] requires newly updated keyword-file pairs remain private against the server that previously got any search token of those keywords. Leakages of non-forward-secure schemes can be exploited, as shown by Zhang *et al.* [37]. Bost [3] formally defined forward security, and Lai and Chow [26] gave a generalized definition. Quite a few forward-secure DSSE schemes have been proposed using advanced cryptographic primitives [3, 4] or novel data structures [26]. Security under adaptive attacks is established by formally confining the leakage.

Also regarding updates, *backward security* [31] ensures that keyword searches of any deleted keyword-file pair ideally reveal nothing about the deleted file, *e.g.*, not even any hint whether it has never been added or has just been deleted. Bost *et al.* [4] defined three levels of backward security for single-keyword DSSE.

Range-Searchable Encryption. Searching a range of values is a common query that also constitutes more expressive ones such as skylining [35]. It should be more efficient than searching each value in the range one by one. Handling encrypted range queries without order-preserving encryption [2] is posed as an important problem towards SQL on encrypted databases [15].

Demertzis *et al.* [10] and Faber *et al.* [11] are the first in building range-searchable encryption upon SSE with range-covering techniques of a tree-like index. However, their frameworks apply only to *static* SSE, while updates are suggested to be handled in batch by setting up a new instance [10]. *Native dynamism* that processes even just a single update in real-time is more natural.

Systemization-of-Knowledge (SoK). We start with a mini SoK study on prior works. Demertzis *et al.* [10] provided a set of range-covering techniques. To construct dynamic SSE for range queries, or *range* DSSE, later works [38, 39] proposed *ad hoc* designs, namely, a similar tree-like index over a specific DSSE scheme [3] or file indexes of long bit-vectors, requiring local tree *rebuilding* for search/update. These designs also lead

Jiafan Wang: Dept. of Information Engineering, The Chinese University of Hong Kong, E-mail: wj016@ie.cuhk.edu.hk
***Corresponding Author: Sherman S. M. Chow:** Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong, E-mail: [firstname]@ie.cuhk.edu.hk

Table 1. Property of (Range) SSE/DSSE Schemes ($\rightarrow$/$\leftarrow$: Forward/Backward Security, #: Non-predefined (Large) Number of Files)

Scheme	Range Query	(Native) Dynamism	$\rightarrow$	$\leftarrow$	#	No False Positive	Symmetric-key Building Block	Inverted Index	Special Feature
$\sum$oφoς [3]	✗	✓	✓	✗	✓	✓	✗	✓	Maliciously Secure
Cascaded Triangle [26]	✗	✓	✓	✗	✓	✓	✓	✓	Parallelizable
FASTIO [30]	✗	✓	✓	✗	✓	✓	✓	✓	Optimized I/O
Logarithmic-SRC/SRC-i [10]	✓	✗	✓	✗	✓	✗	✓	✓	$\mathcal{O}(1)$ Token Size
Logarithmic-BRC/URC [10]	✓	✗	✓	✗	✓	✓	✓	✓	Logarithmic Search
SEAL [9]	✓	✗	✗	✗	✓	✓	✓	✓	Adjustable Leakage
Scheme-A [38]	✓	✓	✓	✗	✓	✓	✗	✓	n/a
Scheme-B [38]	✓	✓	✗	✓	✗	✓	✗	✗	n/a
FBDSSE-RQ [39]	✓	✓	✓	✓	✗	✓	✓	✗	n/a
MoRSE(Π)	✓	✓	✓	✓	✓	✓	✓	✓	Inherited from Π
(RS)^2	✓	✓	✓	✓	✓	✓	✓	✓	Reduced Storage

Table 2. Efficiency of Range SSE/DSSE Schemes with a Notation Summary for Typical Parameters of Interests (*)

Scheme	Server Computation		Client Computation		Communication	
	Search	Update	Search	Update	Search	Update
Log.-SRC [10]	$\mathcal{O}(N)$				$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert + \epsilon)$	
Log.-SRC-i [10]	$\mathcal{O}(w_q + \lvert\mathtt{DB}(q)\rvert)$	n/a	See Sec. 7.1	n/a	$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert + \epsilon')$	n/a
Log.-BRC/URC [10]	$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$				$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$	
Scheme-A [38]	$\mathcal{O}(n_q)$	$\mathcal{O}(\log W)$ or $\mathcal{O}(N)^\dagger$	$\mathcal{O}(W_\mathrm{x} + \log w_q)$	$\mathcal{O}(W_\mathrm{x})$ or $\mathcal{O}(N)^\dagger$	$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$	$\mathcal{O}(\log W)$ or $\mathcal{O}(N)^\dagger$
Scheme-B [38]	$\mathcal{O}(\log w_q)^\ddagger$	$\mathcal{O}(\log W)$	$\mathcal{O}(W_\mathrm{x} + \log w_q)$	$\mathcal{O}(W_\mathrm{x})$	$\mathcal{O}(\log w_q)^\ddagger$	$\mathcal{O}(\log W)$
FBDSSE-RQ [39]	$\mathcal{O}(\overline{n_q})$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$	$\mathcal{O}(\lvert\mathcal{W}\rvert)$	$\mathcal{O}(\lvert\mathcal{W}\rvert)$	$\mathcal{O}(\log w_q)^\ddagger$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$
MoRSE(Π)	$\mathcal{O}(n_q)$	$\log(\lvert\mathcal{W}\rvert)T_s^{\mathsf{Upd}}$	$\log(w_q)T_c^{\mathsf{Srch}}$	$\log(\lvert\mathcal{W}\rvert)T_c^{\mathsf{Upd}}$	$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$
(RS)^2	$\mathcal{O}(\overline{n_q})$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$	$\mathcal{O}(w_q)$	$\mathcal{O}(W + \log \lvert\mathcal{W}\rvert)$	$\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$

* W is the number of distinct keywords in the database. W_x is the largest value of existing keywords. $\mathcal{W}$ is the keyword space.
w_q is the number of keywords within a range query. Special notes for complexity marked with $^{\dagger/\ddagger}$ can be found in Section 7.
n_q is the total number of updates (add + del) that contain the keywords in a range query q since initialization.
$\overline{n_q}(< n_q)$ is the total number of updates that contain the keywords in a query q since the last search of them.
$\lvert\mathtt{DB}(q)\rvert$ ($= n_q$ if no deletion) is the number of files matching a range query q. ϵ/ϵ' refers to the false positive, where $\epsilon' < \epsilon$.
N is the database size (largest here), *i.e.*, the total number of updates (add + del) since initialization.
$T_c^{\mathsf{Srch}}/T_c^{\mathsf{Upd}}$ (resp. $T_s^{\mathsf{Srch}}/T_s^{\mathsf{Upd}}$) are the client (resp. server) search/update complexity of Π. All except T_s^{Srch} are typically $\mathcal{O}(1)$.

to inefficiency at different levels – handling many long bit-vectors at the data-structure level or using homomorphic techniques to enable (forward-secure) searches and updates at the cryptography level. Studying these major designs unveils neglected complexities (detailed in Section 7, which also explains Tables 1-3) and sheds light on how to tackle their shortcomings.

Overview of Our Contributions. In response, we put forward a range DSSE framework adopting the range-covering techniques [22, 29]. We propose two range DSSE constructions with subtle modifications that avoid drawbacks in prior works. Particularly, they offer newer functionalities (*e.g.*, supporting native updates without asymmetric or homomorphic primitives) and better performance (*e.g.*, further reducing client-side storage). Tables 1 and 2 summarize our advantages in security, efficiency, and features. We also extend the backward security definitions [4] for range DSSE.

Despite the cryptographic nature of our study, we also emphasize practical aspects. Notably, we give a detailed performance analysis to illustrate the practical advantages brought by the desired properties. We also consider theoretical notions in practice and uncover vulnerabilities [38]. Below we highlight our contributions.

New Constructions. We start with a range DSSE framework, abstracting existing works and distilling the core idea of reducing a range search to multiple single-keyword searches while handling dynamism natively. We propose two schemes based on the framework.

Table 3. Client Storage of (Forward-Secure) Range SSE/DSSE

Scheme	Client Storage
Logarithmic-SRC [10]	$3W - \log W - 2$
Logarithmic-SRC-i [10]	$3W - \log W + 3D - \log D - 4$
Logarithmic-BRC/URC [10]	$2W - 1$
Scheme-A [38]	$2W - 1$
FBDSSE-RQ [39]	$2W + \log \lvert\mathcal{W}\rvert - \log W$
MoRSE(Π)	$2W + \log \lvert\mathcal{W}\rvert - \log W$
(RS)^2	$W + s$

We follow notations in Table 2. D is the number of distinct files. s is the number of distinct (possibly overlapping) sub-ranges in all queries (See Section 5.3). We assume W and D are integral multiples of 2, and existing keywords are contiguous. Π stores a state per keyword as typical forward-secure DSSE [3, 26, 30].

Our first scheme MoRSE is a modular approach that upgrades any typical forward-secure DSSE (generically obtainable from DSSE [26]). While Logarithmic-BRC [10], scheme-A [38], and MoRSE all build upon the range-covering technique called best range cover (Section 3.3), they have very different properties in multiple dimensions. In short, MoRSE captures the essence of prior works but avoids their design-specific problems [38, 39] (Section 7). It performs better on many criteria, notably client search complexity (Table 2), by avoiding indexes rebuilding at a little client-storage cost.

Our second scheme (RS)^2 aims at reducing client storage (see Table 3) with most features remaining competitive. Namely, the storage is always less than that of other round-optimal forward-secure constructions before all possible ranges have been queried.

Besides the favorable properties in Table 1, our modular design enables i) physical deletion [1, 30] for reclaiming the space for deleted data and ii) search results archives [14], which cache the file identifiers in prior authorized searches and save the future search time.

In a sense, Demertzis *et al.* [10] studied six different range-covering realizations for (static) SSE based on the same skeleton (which we extract in Section 4.1), some with extra structures (*e.g.*, two indexes in Logarithmic-SRC-i [10]) to tackle some shortcomings. We instead focus on getting the best out of the best range cover, but follow a similar spirit in proposing techniques to tackle specific shortcomings (*e.g.*, reducing client storage).

New File Injection Attack. We identify a variant of the adaptive file injection attack determining the range to query (vs. a single keyword) over non-forward-secure range DSSE. This attack breaks the forward security claim of scheme-B [38]. Our result carries independent interests in studying range DSSE security.

Refined Definition of Backward Security. We illustrate the subtlety of backward security in range DSSE. The only formal definition [39] (subsequent to the preprint version of this work [32]) solely concerns a specific case. Our three levels of backward security, extended from DSSE [4], are more generic. Following the definition, we upgrade our schemes with a two-round transformation [4] for backward security.

Theoretical and Experimental Evaluations. We provide complete theoretical analyses of the proposed constructions, followed by experimental evaluations. In particular, we make clear the use of the range-covering technique (with a suggested implementation) and include its overhead in Table 2. Unfortunately, these were either not mentioned explicitly [10] or previously omitted in the client-side overhead analysis [38, 39].

While recent works provide no [38] or limited evaluations [39], we implement ours and FBDSSE-RQ [39] for comprehensive analyses of practical performance for a >1 million keyword space and ∼5 million records. We measure the client-side storage omitted in other works.

Organization. The next two sections overview the related works and the preliminaries (*e.g.*, basic notations). Section 4 sketches a range DSSE framework and proposes our modular range DSSE construction. In Section 5, we propose a reduced-client-storage range DSSE scheme with forward security. Section 6 discusses backward security and potential reconstruction attacks. Section 7 thoroughly reviews prior range SSE schemes and shows an attack breaking a wrong claim of forward security. (Also see Appendix D.) Section 8 presents our prototypes and experiments, followed by a conclusion.

2 Related Work

Faber *et al.* [11] first studied how to support range queries in SSE by utilizing a binary tree index. The tree nodes covering the queried range are searched with an SSE called OXT [6] (oblivious cross-tags protocol) for disjunctive queries. Demertzis *et al.* [10] built six range SSE schemes with different range-covering techniques of tree indexes and can be deemed as more generic than the range SSE of Faber *et al.* [11]. One of the schemes [10] is later adapted for adjustable leakage [9].

Two range DSSE schemes are then proposed [38]. Both require tree construction before each query, while one of them is indeed not forward-secure. Note that a fix [39] appeared after we had publicized our designs [32] and the attack against the non-forward-secure scheme [38]. The fix [39] also has its own drawbacks, *e.g.*, an even worse client-side computation complexity [38].

3 Preliminary

Let $\mathsf{negl}(\lambda)$ be a negligible function in λ. PPT stands for probabilistic polynomial-time. For a set X, $x \leftarrow_{\$} X$ uniformly samples an x from X. In a two-party protocol $(c_{\text{out}}; s_{\text{out}}) \leftarrow P(c_{\text{in}}; s_{\text{in}})$ between a client and a server, c_{in} and c_{out} (resp. s_{in} and s_{out}) are inputs and outputs of the client (resp. server). $||$ means concatenation.

3.1 Dynamic SSE

Definition 3.1 (DSSE [3])**.** A DSSE scheme is a tuple of PPT algorithms/protocols (Setup, Search, Update).

$\underline{(K, \mathsf{EDB}, st) \leftarrow \mathsf{Setup}(1^\lambda)}$ is a PPT algorithm taking a security parameter λ. It outputs a secret key K, an (initially empty) encrypted database EDB, and state st.

$\underline{((st', \mathcal{R}); (\mathsf{EDB}', \mathcal{R})) \leftarrow \mathsf{Search}(K, st, q; \mathsf{EDB})}$ is a protocol between the client (with a secret key K, a state st, and a query q), and the server (with an encrypted database EDB). The client outputs a new state st', while the server outputs a (possibly) updated database EDB'. Both parties output a sequence of responses $\mathcal{R}$, which essentially consists of the (identifiers of) files matching q.

$\underline{(st'; \mathsf{EDB}') \leftarrow \mathsf{Update}(K, st, op, up; \mathsf{EDB})}$ is a protocol between the client (with a secret key K, a state st, an operation $op \in \{\mathsf{add}, \mathsf{del}\}$, and an update input up parsed as a keyword-file pair (w, id)), and the server (with an encrypted database EDB). The client outputs a new state st' and the server updates EDB to EDB'.

In a round-optimal DSSE scheme, for Search (resp. Update), the client generates a search token t_q (resp. update token t_u) for the server to process EDB locally. No interaction needed before the output of responses.

A DSSE scheme is correct if for all security parameters λ, all $(K, \mathsf{EDB}, st) \in \mathsf{Setup}(1^\lambda)$, and all sequences of Search and Update operations, the search protocol $\mathsf{Search}(K, st, q; \mathsf{EDB})$ returns correct responses (*i.e.*, the identifiers of files matching the query q) except with negligible probability. See [5] for a formal definition.

Adaptive security is captured under the real/ideal simulation paradigm with a stateful leakage function set $\mathcal{L}$ for simulation. Each component of $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}\}$ corresponds to the leakage during setup, search, and update operations, respectively.

Definition 3.2. We say DSSE = (Setup, Search, Update) is $\mathcal{L}$-adaptively-secure, where $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}\}$ is a set of stateful leakage functions, if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that:

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

$\mathbf{Real}_{\mathcal{A}}(1^\lambda)$: The challenger executes $\mathsf{Setup}(1^\lambda)$ and sends (initially empty) EDB to $\mathcal{A}$. Then $\mathcal{A}$ adaptively makes a polynomial number of search queries q and update queries (op, up). The challenger returns the transcripts generated by running Search or Update protocol on q or (op, up), respectively. Finally, $\mathcal{A}$ returns a bit b.

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$: $\mathcal{S}$ generates (initially empty) EDB using $\mathcal{L}^{\mathsf{Stp}}$ and sends it to $\mathcal{A}$. $\mathcal{A}$ adaptively makes a polynomial number of search queries with input q and update queries with input (op, up). For a query q, $\mathcal{S}$ returns the transcripts generated with $\mathcal{L}^{\mathsf{Srch}}(q)$. For an update (op, up), $\mathcal{S}$ returns the transcripts generated with $\mathcal{L}^{\mathsf{Upd}}(op, up)$. Eventually, $\mathcal{A}$ returns a bit b.

Both probabilistic experiments output the bit b.

Search Pattern and Update History. Definition 3.2 captures the leakage of DSSE with the leakage function set $\mathcal{L}$. $\mathcal{L}$ maintains an *operation list* Q to record all operations issued so far. Assume u is the timestamp when an operation happens, Q records (u, w) for a search on keyword w, or (u, op, w, id) for an update with $(op, up = (w, id))$. Each individual leakage function $(\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}})$ implicitly takes Q as input, whose last record is the last operation before evaluating the leakage. This pinpoints the leakage incurred due to the last operation while considering all historical operations.

Using Q, we define the repetition of queried keywords as the *search pattern* $\mathsf{sp}(w) = \{u | (u, w) \in Q\}$, which is the information leaked in typical SSE schemes. We use $\mathsf{hist}(w) = \{(u, op, id) | (u, op, w, id) \in Q\}$ to record the *update history* for every keyword since initialization.

3.2 File Injection Attack

Zhang *et al.* [37] demonstrated file injection attacks against DSSE. In their setting, the server could send files of its choice to the client, who then encrypts and uploads them as normal updates in DSSE schemes. Let $\mathcal{W}$ be the keyword space. The server would inject $\lceil \log |\mathcal{W}| \rceil$ different files, each containing exactly half of the keywords from $\mathcal{W}$. By observing the set of injected files returned for a search token, the adversary could tell which keyword is contained in the token. This is a non-adaptive

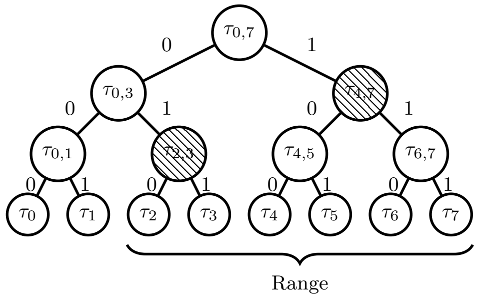

Fig. 1. Example for the Best Range Cover of $[2, 7] = [010, 111]$

attack and can be mitigated by limiting the number of keywords in a single file by a threshold T.

An adaptive attack against the above threshold-based countermeasure divides the keyword space into $\lceil|\mathcal{W}|/T\rceil$ subsets. The server first injects $(\lceil|\mathcal{W}|/T\rceil - 1)$ files, each containing T keywords, to determine which subset the keyword of the search token lies in. For the target subset, the server only needs to inject $\mathcal{O}(\log T)$ files to figure out the exact keyword like the above non-adaptive attack. This adaptive attack efficiently breaks the threshold-based countermeasure. Zhang *et al.* [37] showed that, after the target subset is known, only 8 new files have been injected when $T = 200$. Whereas this attack requires the knowledge of the relation between newly injected files and the keyword in the target search token. Such leakage is hidden by forward security to be formally defined in Section 4.2.

3.3 Best Range-Covering Technique

We review the *best range cover* (BRC) technique used by prior schemes [10, 38] and ours. For simplicity, we assume a perfect binary tree with 2^m leaves and a numerical domain $\mathcal{D} = \{0, \ldots, 2^m - 1\}$. Starting from the left, the i-th leaf node is denoted by τ_i and assigned with the numerical value i; thus $\{\tau_i\}_{i\in\mathcal{D}}$ covers $\mathcal{D}$. Each non-leaf node is denoted by $\tau_{i,j}$, where i, j are the values assigned to the left-most and right-most leaf descendants of the node, respectively, and $i < j$. That is, a node $\tau_{i,j}$ is assigned with the range of values $\{i, \ldots, j\} \subset \mathcal{D}$.

Given a range (of contiguous values) over the domain $\mathcal{D}$, BRC selects the minimum set of tree nodes that exactly covers the range. For the example in Figure 1, values of domain $\{0, \ldots, 7\}$ are assigned to leaf nodes accordingly. BRC of range $[2, 7]$ contains $\tau_{2,3}$ and $\tau_{4,7}$ (shown as patterned nodes). The number of nodes in BRC is always no more than the number of values in the range. We define the GetBRC algorithm as follows.

```
GetBRC_m(a, b)
1 :  BRC ← ∅, i := 0
2 :  $a_0 \cdots a_m := [a]_{bin}, b_0 \cdots b_m := [b]_{bin}$
3 :  while $a_0 \cdots a_{m-i} < b_0 \cdots b_{m-i}$ do
4 :    if $a_{m-i} = 1$ then
5 :      BRC := BRC ∪ {$a_0 \cdots a_{m-i}$}
6 :    if $b_{m-i} = 0$ then
7 :      BRC := BRC ∪ {$b_0 \cdots b_{m-i}$}
8 :    $a_0 \cdots a_{m-i} := a_0 \cdots a_{m-i} + [1]_{bin}$
9 :    $b_0 \cdots b_{m-i} := b_0 \cdots b_{m-i} - [1]_{bin}$
10 :   $i := i + 1$
11 : endwhile
12 : if $a_0 \cdots a_{m-i} = b_0 \cdots b_{m-i}$ then
13 :   BRC := BRC ∪ {$a_0 \cdots a_{m-i}$}
14 : return BRC
```

Fig. 2. Suggested Implementation of Best Range Cover (GetBRC)

BRC ← GetBRC$_m(a, b)$ is a deterministic algorithm that takes as input the height m of a binary tree and two numerical values a, b representing a range $[a, b]$ over domain $\{0, \ldots, 2^m - 1\}$. It outputs BRC of range $[a, b]$.

Kiayias *et al.* [22] introduced how to get BRC: given a range $[a, b]$, it starts by determining the last common ancestor of a and b from the root. By checking specific bits of a and b in their binary forms, it judges whether a node belongs to BRC and moves down towards leaves.

We suggest an implementation the other way round: it ascends from the leaves of a and b towards their first common ancestor, avoiding (probably long) bit checking from the root. Both implementations can be viewed as extensions of the minimal dyadic cover [29], serving as references for instantiating related structures [10, 38] with the asymptotic complexity of $\mathcal{O}(\log(b - a))$.

A Suggested Implementation. We represent the $(k+1)$-bit binary form $[a]_{\mathsf{bin}}$ of a numerical value a by $\mathsf{a}_0 \cdots \mathsf{a}_k$ (prepending 0s if needed) with $\mathsf{a}_0 =$" ", a null string denoting the root. Such an $(m + 1)$-bit binary string can represent the value of every leaf node in a binary tree associated with a domain $\mathcal{D} = \{0, \ldots, 2^m - 1\}$. The binary form for any parent is that for its child with its last bit truncated. For example, in Figure 1, the binary form of node τ_4 is " 100" while its parent $(\tau_{4,5})$ is " 10". With the knowledge of $\mathcal{D}$, we can follow the node representation without explicitly building the tree.

We detail the procedure in Figure 2. For any given range, we first transform its upper and lower bounds to their binary form (line 2). If the last bit of the lower bound equals 1 (resp. the last bit of the upper bound equals 0), the value of the lower (resp. upper) bound will be inserted into BRC (line 4–7). This means the tree node for the lower (resp. upper) bound is a right (resp. left) child of its parent, and none of its ancestors covers solely the values in the range.

The value of the lower (resp. upper) bound will be increased (resp. decreased) by 1 afterward (line 8–9). Then, we truncate the last bit of both bounds by moving the bit position (line 10) and setting the truncated bit-strings as the new bounds. We repeat the above operations until the lower bound is not smaller than the upper bound. If they are equal eventually, we also insert the value of the bound into BRC (line 12–13).

Consider BRC of $[2, 7]$ in Figure 1:
- Transform $2, 7$ into " 010" and " 111" respectively.
- As the last bit of the lower bound is not 1, increase it by 1 (*i.e.*, " 011") and get the new lower bound by truncating the last bit of the result (*i.e.*, " 01").
- As the last bit of the upper bound is not 0, decrease it by 1 (*i.e.*, " 110") and get the new upper bound by truncating the last bit of the result (*i.e.*, " 11").
- As the last bit of the lower bound is 1, insert " 01" into BRC and update the bound likewise (*i.e.*, " 1").
- As the last bit of the upper bound is not 0, update the upper bound as above (*i.e.*, " 1").
- Insert the equal bounds " 1" into BRC.
- BRC of $[2, 7]$ consists of " 01" ($\tau_{2,3}$) and " 1" ($\tau_{4,7}$).

4 A Framework of Range DSSE

4.1 A Skeleton Framework of Range DSSE

Our goal is to build a server-side index for the server to answer range queries from the client, *i.e.*, retrieving the identifiers of files containing any keyword in the range. Let $\mathcal{W}$ be a set of contiguous positive integers. (It is possible to convert a floating-point domain, *e.g.*, temperatures, to $\mathcal{W}$ with scaling and transformation.) We consider one-dimensional queries over $\mathcal{W}$. Each file owned by the client contains at least one keyword in $\mathcal{W}$.

A core idea in range SSE [10] is to reduce a range search to multiple single-keyword searches. This subsection is meant to distill the details from existing works, despite their many differences in security, features, and performance. Our framework also considers native (vs. batched [10]) dynamism. The following will use "keyword" to refer to an artifact denoting a *sub-range* for search and update of the underlying DSSE mechanism, which is different from real keywords in range DSSE.

Setup.
1. Break $\mathcal{W}$ into a set of (potentially overlapping) sub-ranges, and associate a unique "keyword" of the underlying DSSE to every range.
2. Run DSSE.Setup to initialize an encrypted database.

Search.
1. Break the queried range into sub-ranges, and get the "keyword" associated with each sub-range.
2. Search for each "keyword" via DSSE.Search.

Update.
1. For operation $op \in \{\mathsf{add}, \mathsf{del}\}$, parse an update input up into a keyword and a file identifier.
2. Map the keyword to the set of sub-ranges depending the covering technique.
3. Execute DSSE.Update associating the "keywords" for each sub-range to the file identifier for op.

Security. The $\mathcal{L}$-adaptive security for range DSSE can be defined as that of DSSE in Definition 3.2. The security of range DSSE obtained via the above abstract framework is highly related to that of the underlying DSSE. Its leakage can be obtained by augmenting the leakage functions $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}\}$ of the underlying DSSE scheme to capture the extra leakage stemming from the keyword mapping and index structure. See [10, 11] for discussions on leakage caused by different keyword-mapping strategies and efficiency trade-offs.

4.2 Forward and Backward Security

Forward security requires that Update reveals nothing about which keywords are involved in the keyword-file pairs to be updated. Its definition for single-keyword DSSE is readily extensible to range DSSE.

Definition 4.1 (Forward Security [7, 26])**.** We say an $\mathcal{L}$-adaptively-secure (range) DSSE scheme is forward-secure if the update leakage can be written as: $\mathcal{L}^{\mathsf{Upd}}(op, up) = \mathcal{L}'(op, id)$, where $op \in \{\mathsf{add}, \mathsf{del}\}$ is an operation, up is an update input parsed as a keyword-file pair (w, id), and $\mathcal{L}'$ is stateless and does not take w.

Backward security requires that whenever a keyword-file pair (w, id) has been added then deleted, searching over w reveals nothing about file id. Bost *et al.* [4] formalized backward security via three leakage functions over the operation list Q. $\mathsf{TimeDB}(w)$ contains files currently matching w and when (u) they are added. $\mathsf{UpdTime}(w)$

is the timestamp u when all the updates on w happened. Finally, $\mathsf{DelHist}(w)$ lists the timestamp-pairs of deletions and corresponding insertions on w.

$$\begin{aligned}\mathsf{TimeDB}(w) &= \{(u, id) \mid (u, \mathsf{add}, w, id) \in Q \\ &\qquad \wedge\ \forall u', (u', \mathsf{del}, w, id) \notin Q\}, \\ \mathsf{UpdTime}(w) &= \{u \mid (u, \mathsf{add}, w, id)\ \vee\ (u, \mathsf{del}, w, id) \in Q\}, \\ \mathsf{DelHist}(w) &= \{(u^{\mathsf{add}}, u^{\mathsf{del}}) \mid \exists id \text{ s.t. } (u^{\mathsf{del}}, \mathsf{del}, w, id) \in Q \\ &\qquad \wedge\ (u^{\mathsf{add}}, \mathsf{add}, w, id) \in Q\}.\end{aligned}$$

With the above leakage functions, we extend three levels of backward security [4] from single-keyword DSSE to range DSSE. We define them by considering a range query q for Search. Suppose $KSet_q$ is the "keyword" set associated with the sub-ranges of q as illustrated in Section 4.1. We parse an update up as a keyword-file pair (w, id). Also, we omit leakages unrelated to forward/backward security, *e.g.*, the keyword-space size.

Definition 4.2 (Backward Security)**.** We say an $\mathcal{L}$-adaptively-secure range DSSE is
- insertion-pattern revealing backward-secure if
  - $\mathcal{L}^{\mathsf{Upd}}(op, w, id) = \mathcal{L}'(op)$,
  - $\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}''((\mathsf{TimeDB}(k), a_k)_{k \in KSet_q})$,

  where a_k is the total number of updates on k.
- update-pattern revealing backward-secure if
  - $\mathcal{L}^{\mathsf{Upd}}(op, w, id) = \mathcal{L}'(op, w)$,
  - $\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}''((\mathsf{TimeDB}(k), \mathsf{UpdTime}(k))_{k \in KSet_q})$.
- weakly backward-secure if
  - $\mathcal{L}^{\mathsf{Upd}}(op, w, id) = \mathcal{L}'(op, w)$,
  - $\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}''((\mathsf{TimeDB}(k), \mathsf{DelHist}(k))_{k \in KSet_q})$.

The earlier definition is stronger. $\mathcal{L}'$ and $\mathcal{L}''$ are stateless, *i.e.*, their outputs solely depend on the inputs.

Note that a combination of forward security and weak backward security is enough to limit the update leakage to the type of involved operations [4].

Any search query on keyword w, happening between the insertion and the deletion of keyword-file pair (w, id) for the same keyword w, will expose the file identifier id associated with the deletion. Bost *et al.* [4] excluded this case for backward security for single-keyword DSSE.

For range DSSE, we cannot reuse the definition verbatim. Between the insertion and the deletion of keyword-file pair (w, id), we cannot just exclude any search query on w. We need to exclude any search query on (any "keyword" associated with) any range covering the updated keyword w. For example, a file f, added for keyword 6, will also be added for ("keyword" associated with) $[4, 7]$. If a search query on $[4, 7]$ is issued before the deletion of the keyword-file pair $(6, f)$, the adversary can link file f with the deleted file of 6 by simply observing the insertion and deletion time revealed by a subsequent query on 6. It suggests that a trivial extension cannot guarantee the privacy of the deleted file.

```
Setup(1^λ)
1 : $(K, \mathsf{EDB}, st) \leftarrow \Pi.\mathsf{Setup}(1^\lambda)$
2 : return $(K, \mathsf{EDB}, st)$

Update(K, st, op, up; EDB)
1 : Parse $up$ as $(w, id)$
2 : $\mathsf{w}_0 \cdots \mathsf{w}_m := [w]_{\mathsf{bin}}$
3 : for $i := 0$ to $m$ do
4 :     $up_i := (\mathsf{w}_0 \cdots \mathsf{w}_i, id)$
5 :     $(st'; \mathsf{EDB}') \leftarrow \Pi.\mathsf{Update}(K, st, op, up_i; \mathsf{EDB})$
6 :     $st := st', \mathsf{EDB} := \mathsf{EDB}'$
7 : endfor
8 : return $(st'; \mathsf{EDB}')$

Search(K, st, q; EDB)
1 : Parse $q$ as $[a, b]$
2 : $CSet \leftarrow \mathsf{GetBRC}_m(a, b)$
3 : $RSet \leftarrow \emptyset$
4 : for $\tau \in CSet$ do
5 :     $((st', \mathcal{R}); (\mathsf{EDB}', \mathcal{R})) \leftarrow \Pi.\mathsf{Search}(K, st, \tau; \mathsf{EDB})$
6 :     $RSet := RSet \cup \mathcal{R}; st := st'; \mathsf{EDB} := \mathsf{EDB}'$
7 : endfor
8 : return $((st', RSet); (\mathsf{EDB}', RSet))$
```

Fig. 3. MoRSE: Modular Range DSSE building upon DSSE Π

4.3 Our Secure Modular Construction

Our modular range DSSE construction MoRSE (Figure 3) is based on a binary-tree-like index and any typical DSSE scheme Π. The resulting scheme achieves forward security, provided Π is forward-secure. Section 6.1 will show how to upgrade it with backward security.

Description. Let $CSet$ be the covering-node set that helps the client record elements in BRC of a given range, and $RSet$ be the set recording the search result during the search operation. Without loss of generality, we assume the size of keyword space $\mathcal{W}$ is 2^m.

Our scheme uses the binary representation of the keyword to implicitly maintain a binary tree with depth logarithmic in the keyword space size. More specifically, for each keyword w from $\mathcal{W}$ (*i.e.*, the leaf node), its

$(w{+}1)$-bit binary form is $\mathsf{w}_0 \cdots \mathsf{w}_m := [w]_{\mathsf{bin}}$ (prepending 0s if needed) with $\mathsf{w}_0 =$" ". Then, we can represent the ancestors for the leaf node of w by $\mathsf{w}_0, \mathsf{w}_0\mathsf{w}_1, \mathsf{w}_0\mathsf{w}_1\mathsf{w}_2, \ldots,$ and $\mathsf{w}_0 \cdots \mathsf{w}_{m-1}$. Among these ancestors, w_0 always represents the root with representation " ".

For a range query, the client derives its best range cover via GetBRC (Figure 2) and stores it as *CSet* (line 2 of Search). For each element of *CSet*, the client searches for it using the underlying DSSE (line 4–7). The server returns the union of all results as *RSet*.

When adding/deleting a keyword-file pair in the index, the client represents the keyword in its binary form (line 2 of Update). Besides the keyword itself, every ancestor of it will be treated as a "keyword" (line 3–4) to be updated with the underlying DSSE scheme accordingly (line 5). The client does not need to rebuild the binary tree (*cf.* [38, 39]) since the ancestors' locations can be found with the knowledge of the (binary form of) keyword value. The client simply records the height of the implicit tree (and the bit length of the keyword value) to process this binary-tree-like index correctly.

Efficiency. Let T_c^{Srch} and T_c^{Upd} (resp. T_s^{Srch} and T_s^{Upd}) be the client-side (resp. server-side) search and update complexity for Π. Typically [3], $T_c^{\mathsf{Srch}} = T_c^{\mathsf{Upd}} = T_s^{\mathsf{Upd}} = \mathcal{O}(1)$ and $T_s^{\mathsf{Srch}} = \mathcal{O}(n_q)$, which is parallelizable in some schemes [26, 30], where n_q is the number of matches.

The update of MoRSE(Π) takes the client (resp. server) an overhead of $\log|\mathcal{W}| \cdot T_c^{\mathsf{Upd}}$ (resp. $\log|\mathcal{W}| \cdot T_s^{\mathsf{Upd}}$), since Π.Update is executed for $\log|\mathcal{W}|$ times. For the search of MoRSE(Π), the client runs GetBRC once and executes Π.Search for each element in BRC of query q, leading to $\log w_q \cdot (T_c^{\mathsf{Srch}} + 1)$ client-side computation.

The server-side computation is upper-bounded by $\log w_q \cdot T_s^{\mathsf{Srch}}$. Yet, note that the server in MoRSE(Π) just executes Search of Π for each part of the range, taking the sum of search complexity T_s^{Srch} for each sub-query q_i. When T_s^{Srch} is linear in the number of matches n_{q_i} for each sub-query q_i (and $\log w_q < n_q$), the sum is linear in the number of matches for the range query, *i.e.*, $\mathcal{O}(n_q)$. With a similar analysis, since BRC covers exactly the range, the search communication adds up to $|\mathtt{DB}(q)|$ (which is n_q if there was no deletion).

MoRSE(Π) has a $\log|\mathcal{W}|$ blow-up over Π for update communication. Typical forward-secure DSSE, *e.g.*, [3, 26, 30] features constant-size update communication.

MoRSE(Π) treats every accessed tree node as a keyword, while forward-secure DSSE typically requires the client to store a state per keyword. To baseline our reduced-storage scheme (RS)^2 in the next section, we consider the optimal case where keywords are contiguous (mostly belonging to the same subtree). The client storage overhead is then the number of existing keywords and the non-leaf nodes on their way to the root, *i.e.*, around $(2W + \log|\mathcal{W}| - \log W)$ in Table 3.

To illustrate, suppose existing files only have 0 or 1 as their keywords, *i.e.*, $W = 2$, for the keyword space in Figure 1, the stored states are: τ_0, τ_1, $\tau_{0,1}$ (*i.e.*, around $2W$), plus $\tau_0, \tau_{0,1}, \tau_{0,3}, \tau_{0,7}$ (*i.e.*, $\log(|\mathcal{W}|)+1$ nodes along the leaf-to-root path), minus the double-counted $\tau_0, \tau_{0,1}$ (*i.e.*, $\log(W) + 1$ nodes, the subtree height).

MoRSE(Π) naturally benefits from the development of DSSE. For example, when instantiated with forward-secure FASTIO [30] (reviewed in Appendix A), our scheme inherits the advantages of physical deletion and caching the previous search results. The server search complexity is reduced to $\mathcal{O}(\overline{n_q})$, where $\overline{n_q}$ is the number of updates that contain the keywords in the queried range since the last search of them, boosting efficiency.

Security. The search protocol incurs the search leakage of Π over *CSet*. Since the update protocol reveals nothing regarding the keyword beyond the leakage of Π, our modular construction is forward-secure when Π satisfies forward security. We formally provide the leakage functions and analyze the security in Appendix B.

5 Saving Client Storage

To our knowledge, the best client storage for round-optimal single-keyword *forward-secure* DSSE (*e.g.*, [3, 26]) is linear in W, the number of distinct keywords. Table 3 suggests that existing range DSSE schemes require client storage of at least twice as much as that of single-keyword forward-secure DSSE, even if no range query has been issued. Scheme-B [38] tries to solve this problem, but it is not forward secure.

This extra client-side overhead may dampen users' enthusiasm in outsourcing. Some clients perform range queries but may not be as frequent as single-keyword queries. Persistently paying a large overhead for an infrequent operation is uneconomical.

We thus propose a new scheme (RS)^2 with reduced client storage. If no range query happens, the client storage is just W. It increases when any range is queried for the first time. Only when all possible ranges have been queried will the client storage grow as prior range DSSE.

5.1 Overview

In a forward-secure DSSE database, the client needs to store a tuple for each existing keyword to record related information (*e.g.*, a key and a counter). Current range DSSE designs (scheme-A [38], FBDSSE-RQ [39], and our MoRSE) also uses a binary-tree-like structure for storing the same type of tuple for each node, making the client storage at least twice the number of existing keywords.

Storing such tuples for the leaves is hard to avoid as they are the "basis" required by existing round-optimal forward-secure DSSE schemes. Yet, we note that some information about the non-leaf nodes (*e.g.*, the update number) can be computed from the tuples of related leaf nodes. Also, the key is useless before any search over the node. Thus, we record the key only when necessary (after the first search happens over the node corresponding to the key). This is how $(\mathsf{RS})^2$ saves client storage.

For forward security, whenever the previous key of a tree node is revealed, we let the client generate and store a new random key for locating subsequent updates.

5.2 Description

Figure 4 lists the pseudocode of $(\mathsf{RS})^2$. We still assume the size of keyword space $|\mathcal{W}| = 2^m$.

Besides the covering-node set $CSet$ and the result set $RSet$, we use a state set $SSet$ to record *the state of elements to be updated*. t_q and t_u are *collections of the range-query tokens and the update tokens*, respectively.

Setup. H_1 and H_2 are cryptographic hash functions with appropriate domains and output lengths. The client outputs two λ-bit keys K_1 and K_2 for the *pseudo-random function* (PRF) F, together with three empty maps $\mathbf{W}$, $\mathbf{T}_\mathsf{e}$, and $\mathbf{T}_\mathsf{c}$. The client keeps $K = (K_1, K_2)$ and $st = \mathbf{W}$ secretly, while $\mathsf{EDB} = (\mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c})$ is sent to the server. The purpose of each map is explained as follows.

- $\mathbf{T}_\mathsf{e}$ is used to store the encrypted index.
- $\mathbf{T}_\mathsf{c}$ is used to store the last search results.
- For each keyword w (*i.e.*, leaf node) and each internal node that has been searched in the implicit binary tree, $\mathbf{W}$ stores a $(k_\tau, c^\mathsf{h}, c^\mathsf{u})$ tuple:
  - a key k_τ that is either a PRF output when the node τ is added for the first time or a random string after each search over the node,
  - a history counter c^h for the number of historical updates over the node since the initialization, and
  - an update counter c^u to indicate the number of updates over the node after the last search.

We first define histUptCnt, a function for the client to compute c^h_τ of a non-leaf node τ (without storing it) from handy information stored at its descendant nodes. This count is required in Search and Update.

$\underline{c^\mathsf{h}_\tau \leftarrow \mathsf{histUptCnt}(\mathbf{W}, m, \tau)}$ is a deterministic algorithm that takes the map $\mathbf{W}$, the depth of the implicit binary tree m, and a node τ in the implicit binary tree. It outputs c^h_τ, which is essentially the sum of c^h over the existing leaf descendants of τ. See Section 5.3 for details.

Search. For a query q of range $[a, b]$, it takes $K_1, K_2, \mathbf{W}$ from the client and $\mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c}$ from the server.

1. (Line 1) Via GetBRC (Figure 2), the client figures out BRC of the queried range, sets it as the covering node set $CSet$, and resets the query token collection t_q.
2. For each τ in $CSet$:
   (Line 2–8) If $\mathbf{W}[\tau]$ does not exist, *i.e.*, the node τ has not been retrieved before, the client uses histUptCnt to get the historical updates c^h of τ. If $c^\mathsf{h} \neq 0$, node τ will be searched for the first time. The client sets k_τ as $F(K_1, \tau)$ and c^u to be just c^h for node τ.
   (Line 9–11) The client generates token (nym_w, k_w, c):
   - nym_w is a pseudonym for locating previous records related to τ from $\mathbf{T}_\mathsf{c}$,
   - k_w is set as k_τ retrieved from $\mathbf{W}[\tau]$ if $c^\mathsf{u} \neq 0$, and
   - c is a counter indicating how many updates have been performed on τ after the last search for it or since the initialization if not searched.

   (Line 12) $\mathbf{W}[\tau]$ now stores a random key k_τ, the history counter c^h, and update counter $c^\mathsf{u} = 0$.
   (Line 13–15) Sends the collected search tokens t_q.
3. (Line 17–18) For every search token in t_q, the server accesses the last search results in $\mathbf{T}_\mathsf{c}$ with nym_w and puts them into the set $RSet$.
   (Line 19–28) If $k_w \neq \perp$, the server gets its updates since the last search from $\mathbf{T}_\mathsf{e}$ with locations generated by (k_w, c). If $op = \mathsf{del}$, remove id from $RSet$; otherwise, insert id into $RSet$. Slots in $\mathbf{T}_\mathsf{e}$ accessed by the current search can be physically deleted afterward. The search result $RSet$ is archived into $\mathbf{T}_\mathsf{c}$.
4. (Line 29–31) The server outputs a response $\mathcal{R}$, a union set of the result sets for all search tokens in t_q.

Update. Update works similarly to our modular construction, using the same implicit tree-like index but with special housekeeping for saving client storage.

For an update tuple $(op, up = (w, id))$, the client uses $(K_1, \mathbf{W})$ to generate for the server the update token collection t_u, which contains update tokens for nodes corresponding to keyword w and all of its ancestors.

```
$\mathsf{Setup}(1^\lambda)$
1 :  $K_1, K_2 \leftarrow\$ \{0,1\}^\lambda; \mathbf{W}, \mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c} \leftarrow \emptyset$
2 :  return $((K_1, K_2), (\mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c}), \mathbf{W})$

$\mathsf{Update}(K_1, \mathbf{W}, op, up; \mathbf{T}_\mathsf{e})$
Client:
1 :   Parse $up$ as $(w, id)$
2 :   $SSet, t_\mathsf{u} \leftarrow \emptyset; cnt := 0$
3 :   $\mathsf{w}_0 \cdots \mathsf{w}_m := [w]_\mathsf{bin}$
4 :   $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) \leftarrow \mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_m]$
5 :   if $\mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_m] = \perp$ then
6 :     $k_\tau := F(K_1, \mathsf{w}_0 \cdots \mathsf{w}_m)$
7 :     $c^\mathsf{h} := c^\mathsf{u} := 0$
8 :   endif
9 :   $SSet := SSet \cup \{(k_\tau, c^\mathsf{u})\}; cnt := c^\mathsf{h}$
10 :  $\mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_m] := (k_\tau, c^\mathsf{h} + 1, c^\mathsf{u} + 1)$
11 :  for $i := m - 1$ to $0$ do
12 :    $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) \leftarrow \mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_i]$
13 :    if $\mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_i] = \perp$ then
14 :      $cnt := cnt + \mathsf{histUptCnt}(\mathbf{W}, m, \mathsf{w}_0 \cdots \mathsf{w}_i \bar{\mathsf{w}}_{i+1})$
15 :      $k_\tau := F(K_1, \mathsf{w}_0 \cdots \mathsf{w}_i); c^\mathsf{h} := cnt$
16 :      $SSet := SSet \cup \{(k_\tau, c^\mathsf{h})\}$
17 :    else
18 :      $SSet := SSet \cup \{(k_\tau, c^\mathsf{u})\}; cnt := c^\mathsf{h}$
19 :    endif
20 :  endfor
21 :  for $(k_\tau, c) \in SSet$ do
22 :    $addr := H_1(k_\tau || (c + 1))$
23 :    $val := (id || op) \oplus H_2(k_\tau || (c + 1))$
24 :    $t_\mathsf{u} := t_\mathsf{u} \cup \{(addr, val)\}$
25 :  endfor
26 :  send $t_\mathsf{u}$ to server
Server:
27 :  for $(addr, val) \in t_\mathsf{u}$ do $\mathbf{T}_\mathsf{e}[addr] := val$
```

```
$\mathsf{Search}((K_1, K_2), \mathbf{W}, [a, b]; \mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c})$
Client:
1 :   $t_\mathsf{q} \leftarrow \emptyset; CSet \leftarrow \mathsf{GetBRC}_m(a, b)$
2 :   for $\tau \in CSet$ do
3 :     $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) \leftarrow \mathbf{W}[\tau]$
4 :     if $\mathbf{W}[\tau] = \perp$ then
5 :       $c^\mathsf{h} := \mathsf{histUptCnt}(\mathbf{W}, m, \tau)$
6 :       if $c^\mathsf{h} = 0$ then continue
7 :       $k_\tau := F(K_1, \tau); c^\mathsf{u} := c^\mathsf{h}$
8 :     endif
9 :     if $c^\mathsf{u} \neq 0$ then $k_w := k_\tau; k_\tau \leftarrow\$ \{0,1\}^\lambda; c := c^\mathsf{u}$
10 :    else $k_w := \perp; c := 0$
11 :    $nym_w := F(K_2, \tau)$
12 :    $\mathbf{W}[\tau] := (k_\tau, c^\mathsf{h}, 0)$
13 :    $t_\mathsf{q} := t_\mathsf{q} \cup \{(nym_w, k_w, c)\}$
14 :  endfor
15 :  send $t_\mathsf{q}$ to server
Server:
16 :  $\mathcal{R} \leftarrow \emptyset$
17 :  for $(nym_w, k_w, c) \in t_\mathsf{q}$ do
18 :    $RSet := \mathbf{T}_\mathsf{c}[nym_w]$
19 :    if $k_w \neq \perp$ then
20 :      for $i := 1$ to $c$ do
21 :        $addr := H_1(k_w || i)$
22 :        $(id, op) := \mathbf{T}_\mathsf{e}[addr] \oplus H_2(k_w || i)$
23 :        if $op = \mathsf{del}$ then $RSet := RSet \setminus \{id\}$
24 :        else $RSet := RSet \cup \{id\}$
25 :        delete $\mathbf{T}_\mathsf{e}[addr]$
26 :      endfor
27 :    endif
28 :    $\mathbf{T}_\mathsf{c}[nym_w] := RSet$
29 :    $\mathcal{R} := \mathcal{R} \cup RSet$
30 :  endfor
31 :  send $\mathcal{R}$ to client
```

Fig. 4. (RS)^2: Reduced-Storage Forward-Secure Range DSSE

1. (Line 3–10) If $\mathbf{W}[[w]_\mathsf{bin}]$ is not set (returning $\perp$), the client sets k_τ as $F(K_1, [w]_\mathsf{bin})$ and initializes c^h and c^u to 0 for w. The tuple (k_τ, c^u), possibly from $\mathbf{W}[[w]_\mathsf{bin}]$, is put to $SSet$. c^u and c^h in $\mathbf{W}[[w]_\mathsf{bin}]$ will increase by 1.
2. (Line 11–20) For every ancestor of w, the client gets $(k_\tau, c^\mathsf{h}, c^\mathsf{u})$ from $\mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_i]$. If it is not $\perp$, *i.e.*, the node has been retrieved, the client inserts (k_τ, c^u) into $SSet$; otherwise, the client gets the number of historical updates c^h with $\mathsf{histUptCnt}$, assigns k_τ to be the PRF output, and inserts (k_τ, c^h) into $SSet$.
   Note that the client is not required to store these values currently. To avoid repetitive access to $\mathbf{W}$ by $\mathsf{histUptCnt}$, we use a temporary counter cnt to record the sum of historical updates in previous nodes.
3. (Line 21–26) For each (k_τ, c) in $SSet$, the client generates update token $(addr, val)$, where $addr$ is a location based on the hash of k_τ and the counter c, and val

is both id and $op \in \{\mathsf{add}, \mathsf{del}\}$ encrypted by another hash of $k_{\mathcal{T}}$ and c. The update tokens are sent as t_{u}.

4. (Line 27) For each $(addr, val)$ in t_{u}, the server stores val for insertion/deletion of a keyword-file pair at the location $addr$ of the encrypted index $\mathbf{T}_{\mathsf{e}}$.

5.3 Analysis

Efficiency. The client storage overhead is $\mathcal{O}(W + s)$, where s is *the number of distinct (possibly overlapping) sub-ranges in all issued queries*, *i.e.*, the size of the union set of all *CSet*s (BRC's of range queries) since initialization, which only increases when a query accesses any implicit node that has never been retrieved. Consider Figure 1, if the first range query is $[2, 7]$, $s = 2$ since $\tau_{2,3}$ and $\tau_{4,7}$ are retrieved for the first time. Afterward, s will not increase until querying a range beyond $[2, 7]$, $[2, 3]$, and $[4, 7]$. If $[4, 5]$ is queried next, it overlaps with $[4, 7]$ but s will still be incremented (by 1).

Information of any node besides the existing leaf nodes will not be stored until it is retrieved for the first time. We trade the update complexity and obtain the following advantages that are hardly found in other range DSSE constructions. (RS)^2 has the same client storage as single-keyword forward-secure DSSE if no range query happens. For a "complete" database (with all possible keywords), our client storage is always less than the existing range DSSE schemes before every possible non-overlapping range has been queried, *i.e.*, all nodes in the index have been retrieved.

The client computation overhead for Update is $\mathcal{O}(W + \log|\mathcal{W}|)$, *i.e.*, an extra $\mathcal{O}(W)$ is needed, which is the worst case for histUptCnt where the client accesses all existing leaf nodes and sums the counters of them accordingly to get the number of historical updates over the target. However, we can compute histUptCnt in $\mathcal{O}(1)$ time when the number of queried ranges increases. As the number of historical updates over a non-leaf node is exactly the sum of historical updates of its children, we exploit the information for the adjacent nodes of the target that exists in the client storage. For example, if the parent and the sibling of the target have been retrieved, histUptCnt returns the difference of their history counters; if both children of the target have been retrieved, histUptCnt returns the sum of their history counters.

The number of accessed nodes by histUptCnt is at most w_q, so the client-side search complexity is $\mathcal{O}(w_q)$.

On the server side, the computation overheads for Search and Update are the same as MoRSE(FASTIO). Particularly, the server does not carry out any operation over the part of results that was archived into $\mathbf{T}_{\mathsf{c}}$ beyond putting it in the result set. (RS)^2 keeps the optimal $\mathcal{O}(|\mathtt{DB}(q)|)$ search communication overhead.

Security. Search reveals the search pattern and the update history regarding the covering nodes of the queried range, which is typical for round-optimal schemes. The updated tuple will not be revealed to the server until a related search is issued, hence attaining forward security. Appendix C provides a formal security analysis.

6 Additional Security Concerns

6.1 Backward-Secure Range DSSE

We upgrade both our range DSSE constructions with the two-roundtrip transformation of Bost *et al.* [4]. It provides at least update-pattern-revealing backward security, originally for single-keyword DSSE. To carry out update operation op for the keyword-file pair (w, id), the transformation asks the client to operate with $(w, \mathsf{Enc}_{K_w}(id, op))$ instead, where Enc is an encryption algorithm taking K_w as the symmetric key, derived using another PRF key applying on input w. Thus, the server can only see ciphertexts of the file identifiers and the kind of updates during Search. The client then needs to decrypt all retrieved results, removes the deleted file identifiers locally to get the actual search results, takes an additional round to get the matching files, and re-encrypts the ciphertexts of non-deleted identifiers.

Our first construction MoRSE is generic. We omit the repetitive description of the instantiation but highlight the important aspects. For Update, since MoRSE adopts an implicit binary-tree-like index, the client operates along the path from the leaf, corresponding to the updated keyword, to the root of the tree. Following the transformation, any update over a node along this path is encrypted under a node-specific key. During Search, as the server has no knowledge of the keys, the identifiers and operations of related files will not be exposed before the client removes the deleted ones. After each Search, the client re-encrypts and uploads the non-deleted files to the server with Update. The transformed scheme satisfies update-pattern-revealing backward-security in Definition 4.2, as it only leaks the timestamps of related tuples and the identifiers of files that currently contain any keyword of the queried range.

For our reduced-storage construction (RS)^2, recall that the server keeps the archives for the searched keyword (*i.e.*, $\mathbf{T}_{\mathsf{c}}$); one may worry that the adversary can violate backward security, say, by analyzing the searched results in different stages to gain information of deleted files. However, the archives are only refreshed when a new search query is processed, *i.e.*, any potential leakage involved with the archive can only manifest itself after a new search. As in Section 4.2, we exclude the backward-security coverage when a search query happens between the insertion and the corresponding deletion.

Concretely, for an update over a keyword, the transformation over $(\mathsf{RS})^2$ requires the client to insert the ciphertext encrypting the operation and the file identifier. This will leak the size of *SSet* (*i.e.*, the size of keyword space), yet it is not relevant to backward privacy. For the search of range q, the client decrypts the retrieved ciphertexts corresponding to the best range covering set of q, then removes the deleted file identifiers since the last search of it, which guarantees backward privacy. The server is informed of the identifiers, which should be removed from the archives of each corresponding tree node, and returns the matching files accordingly.

Despite an additional round, the transformed construction maintains the same complexities, and most importantly, preserves our reduced client storage.

The leakage functions $\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}$ of our backward-secure range DSSE constructions are as follows, where $\mathsf{DB}(\tau)$ is the files matching a query over τ.

– $\mathcal{L}^{\mathsf{Stp}} = \perp$; $\mathcal{L}^{\mathsf{Upd}}(op, up) = |\mathcal{W}|$.
– $\mathcal{L}^{\mathsf{Srch}}(q) = ((\mathsf{sp}(\tau), \mathsf{DB}(\tau), \mathsf{UpdTime}(\tau))_{\tau \in \mathsf{BRC} \text{ of } q})$.

Some backward-secure (range) DSSE leak the search pattern $\mathsf{sp}(\cdot)$ too, *e.g.*, Janus [4] and scheme-B [38].

6.2 Discussion on Reconstruction Attack

Reconstruction attacks [12, 20] exploit the *fixed* (volumes of) search results of sufficiently many queries. Some attacks thus simply assume the database is static. Insertion and deletion in range DSSE will change the volume of the responses observed by the adversary, making it less affected than SSE. It is true that the response volume given by range DSSE may also stay unchanged, say, when the next update is yet to happen. Even so, the state-of-the-art attack [12] requires the observation of $\mathcal{O}(|\mathcal{W}|^2 \log |\mathcal{W}|)$ uniformly distributed range queries, which translates to $\sim 20 \cdot 2^{40}$ of them if $|\mathcal{W}| = 2^{20}$ (the setting in Section 8). It may be too many in practice.

Two remarks are in order. First, the attack complexity is exponential when the size of the underlying space is. Second, any such query requires the symmetric key. Similar to the reduction of other cryptographic primitives, the tightness bound often requires the tolerated number of keyed queries to be lower than unkeyed ones.

Moreover, the client can choose to rebuild the encrypted database to reduce the risk when needed. The client storage in $(\mathsf{RS})^2$, which increases with the number of distinct range queries, can indicate when to rebuild. Another way to further foil the leakage attacks is to instantiate our modular construction with advanced (*e.g.*, volume-hiding [16, 33]) SSE or apply generic upgrades (*e.g.*, leakage suppression [17]).

Lacharité *et al.* [23] showed attacks that can reconstruct the original data fully or at least within a constant ratio of error. The accuracy depends on the number of observed queries and whether the adversary has the knowledge of an auxiliary distribution. The attacks require the target dataset to be dense. Update recovery attack of Grubbs *et al.* [12] assumes the adversary has executed the reconstruction attack or a one-time compromise of the database. The assumptions can be difficult to realize in practice. Meanwhile, mitigation strategies are also emerging, *e.g.*, [27].

7 Closer Look at the Prior Art

We divide range SSE schemes [10, 38, 39] into two classes depending on the index type and do a comprehensive review. Particularly, we explain in detail the asymptotic complexities in Table 2 and the client storage overheads in Table 3. We point out a common omission in their analysis. We also note that the forward security claim of scheme-B [38] is wrong and design an attack against it.

7.1 Inverted Index

An inverted index stores a map from a keyword to a set of files. It is commonly used in SSE and DSSE schemes.

7.1.1 Logarithmic-SRC/SRC-i/BRC/URC [10]

Demertzis *et al.* [10] designed tree indexes whose leaf nodes are associated with a numerical keyword domain. Based on different strategies to represent the queried range with tree nodes (for generating tokens), they proposed six range SSE schemes with different security/efficiency trade-offs. Among the adaptively secure schemes (*i.e.*, excluding those using delegatable

PRFs [22]), Logarithmic-SRC (SRC stands for Single Range Cover) covers any queried range with a single tree node, which inevitably incurs *false positive*. Its server computation could be undesirably linear in the database size due to the data skew. Logarithmic-SRC-i reduces the false positive and server computation by involving one more round trip and an extra index for files to double-check the search results. In terms of communication and server computation, Logarithmic-BRC/URC (BRC/URC stands for Best/Uniform Range Cover) provides better performance without false positives.

Notably, all these constructions [10] are static. Demertzis *et al.* suggested a batched update approach, *i.e.*, creating a new instance for the next batch of updates and periodically merge different indexes. This method is not flexible enough when updates happen frequently. Worse still, even for the same keyword, the client needs different tokens for different instances before the instances are merged, which increases the search overhead. Nevertheless, the combination of static SSE and batched updates makes the consideration of forward security a non-issue, as search tokens for existing instances fail to identify any updates in the new instances.

Analysis. Both Logarithmic-SRC and Logarithmic-SRC-i incur false positive in communication (ϵ and ϵ' in Table 2 with $\epsilon' < \epsilon$). Due to the data skew, the false positive leads to the worst-case server search computation of $\mathcal{O}(N)$ in Logarithmic-SRC, which is reduced to $\mathcal{O}(w_q + |\mathtt{DB}(q)|)$ by Logarithmic-SRC-i assuming extra interactions and indexes. Logarithmic-BRC/URC realizes $\mathcal{O}(|\mathtt{DB}(q)|)$ of server computation and communication without false positive by issuing accurate tokens.

Different realizations of the index provide diverse trade-offs between the *search computation overhead* and the client-side storage overhead, *e.g.*, the client needs to reconstruct the index before the search [38]. Table 2 thus does not provide search complexities for the schemes of Demertzis *et al.* since we think that the algorithmic details are underspecified. That said, since outsourcing the unencrypted tree structure to the server inevitably violates security, we assume that the client stores it locally. We could thus estimate the client storage as follows.

Let W and D be the number of distinct keywords and files (for one instance), respectively. For simplicity, W and D are assumed as integral multiples of 2.

– Log.-BRC/URC: It is essentially a binary tree with leaf nodes representing keywords, *i.e.*, $(2W - 1)$ nodes.
– Log.-SRC: This is the same as Logarithmic-BRC/URC except that a common node exists between every two neighboring nodes at the same level. More precisely, the number of common nodes is $(2W - 1)$ nodes for a perfect binary tree, minus the number of leaf nodes W (as the common nodes at this level have been built by the tree), and minus $\log W$ (as the number of common nodes is less than the regular nodes at the same level by one). Thus, $(3W - \log W - 2)$ nodes in total.
– Log.-SRC-i: In addition to a tree for keywords like the one in Logarithmic-SRC, the same structure is used for existing files, *i.e.*, extra $(3D - \log D - 2)$ nodes are needed and $(3W - \log W + 3D - \log D - 4)$ in total.

Table 3 provides the concrete number of stored tuples of these schemes [10]. This evaluation is purely for the index. There may be some other storage overhead omitted, *e.g.*, the storage needed by the underlying SSE.

7.1.2 Scheme-A [38]

Scheme-A [38] uses a similar design as Logarithmic-BRC/URC [10] with $\sum o\varphi o\varsigma$ [3] as its underlying DSSE. Before searching/updating, the client reconstructs a binary tree (whose number of leaf nodes equals W_{x}, the largest value of existing keywords) to get the relations between tree nodes. It also incurs heavy update overheads whenever the update for a new keyword meets a perfect binary tree. As admitted [38], the client needs to get back *all* historical updates of the old root and re-encrypt them for the new root, which is undesirable, especially for large databases. Scheme-A inherits forward security of $\sum o\varphi o\varsigma$ [3] using (public-key) trapdoor permutations without considering backward security.

Analysis. The client storage is $(2W - 1)$ since $\sum o\varphi o\varsigma$ requires the client to store a state per keyword. For the search and update complexities, the reconstruction incurs a client-side overhead of $\mathcal{O}(W_{\mathrm{x}})$, on top of an $\mathcal{O}(\log w_q)$ overhead for using BRC during each search.

The worst case of updates (marked with † in Table 2) returns and re-encrypts all historical updates, leading to $\mathcal{O}(N)$ in all update complexities, where N is the database size. It can be easily illustrated when keywords are inserted in order; the tree will reach the full capacity as a perfect binary tree (of size $2, 4, 8, \ldots$) for $\log |\mathcal{W}|$ times. The reconstruction requires the server to add all file indexes of the old tree to the new one, keep processing everything added so far again and again.

7.2 A Remark on Client-Side Computation

The *client computation overhead* for operating the tree-like index before *searching* or *updating* seems to be not seriously considered in existing works [10, 38, 39], no matter those reviewed in Section 7.1 or to be reviewed in Section 7.3. Demertzis *et al.* [10] omitted details for this in their instantiations. It is later explained in the subsequent schemes [38], which require rebuilding the index whenever there is a search or update. Yet, this overhead is still not included in comparison (Table 1 in [38]). The search/update efficiency may not be as good when it is implemented. Even worse, its follow-up [39] simply did not provide any complexity analysis[1].

7.3 Homomorphically-Encrypted Index

A file index refers to bit-vectors of length being the maximum number of files. The i-th bit of a file index for a keyword w denotes whether the i-th file contains w. Scheme-B [38] and FBDSSE-RQ [39] exploit the homomorphically-encrypted file-index (HE file-index). As no file-index bits are revealed before client-side decryption, HE file-index prevents the server from knowing the deleted files, so it is naturally backward secure.

7.3.1 Scheme-B [38]

Scheme-B was proposed to save the client storage and avoid the worst-case update overhead of scheme-A. It is still a tree-based approach using BRC. Instead of an inverted index, the server stores one HE file-index for every node. Any query still needs reconstructing the tree for the largest existing keyword as scheme-A. The server then accesses the locations according to BRC and returns those ciphertexts. The client decrypts them to get bit-vectors denoting the identifiers of matching files and retrieves the files with another communication round.

Updates also need tree reconstruction. The client finds the path from the leaf node of the keyword to the root of the tree and encrypts a bit-vector for toggling between presence/absence. The server homomorphically adds it to the ciphertext of the indicator-vector at each node along the path to *flip* every presence bit.

1 A reviewer's remark: "over the past few years, cases of hand-wavy treatment of efficiency analysis and unsubstantiated claims of optimality and/or low client storage in STE and SSE papers have increased. We should not let it become the new norm."

Analysis. Due to the reconstruction and BRC, the client-side search complexity remains the same as scheme-A, *i.e.*, $\mathcal{O}(W_\mathrm{x} + \log w_q)$. Both communication and server-side computation overheads (marked with ‡ in Table 2) for the search protocol are $\mathcal{O}(\log w_q)$ as reported [38], which is the number of HE file-indexes accessed and returned. We stress that this cost does not include any processing on (the identifiers of) matching files, which is typically present in the analysis, even for the complexity of $\mathcal{O}(|\mathtt{DB}(q)|)$ in their scheme-A [38].

Update overheads are $\mathcal{O}(W_\mathrm{x})$ for the client for the reconstruction and $\mathcal{O}(\log W)$ for the server for homomorphically adding $\mathcal{O}(\log W)$ ciphertexts (putting aside executing Search for confirming the bit before flipping).

7.3.2 FBDSSE-RQ [39]

FBDSSE-RQ [39] improves the efficiency of scheme-B by encrypting the file-index using symmetric encryption with homomorphic addition. Since initialization, the number of leaf nodes in the tree index is linear in the *keyword space size*, and the client reconstructs such a tree before any search or update. The search and update procedures of FBDSSE-RQ are similar to scheme-B but with the forward security vulnerability fixed.

Analysis. The tree reconstruction of FBDSSE-RQ requires $\mathcal{O}(|\mathcal{W}|)$ client computation, which is even worse than the tree reconstruction of scheme-A/B. Same as scheme-B, the reported search communication is $\mathcal{O}(\log w_q)$, without considering any processing on (the identifiers of) matching files. The server-side search overhead becomes $\mathcal{O}(\overline{n_q})$ as they also employ the trick of archiving previous search results. FBDSSE-RQ also requires the client to store a state per node, making the client storage the same as our MoRSE, *i.e.*, $(2W + \log|\mathcal{W}| - \log W)$, which we analyzed in Section 4.3.

7.3.3 Drawbacks of HE File-Index

Both schemes possess inherent drawbacks of HE index:

- The server cannot retrieve any files until the client *decrypts* the query results and *sends them back.*
- The element size of (a)symmetric cryptographic primitives should be large enough to cover the *predefined* maximum number of files, which in turn impacts the dynamism. No file can be added when it is reached.
- The predefined maximum affects the time complexity of all operations and the storage complexity of

every object in the encrypted index. For example, each query and index node will bear the cost incurred by many empty slots when the predefined maximum should be large to accommodate unforeseen usage.
- Updates are *restrictive* since they crucially rely on the additive homomorphism and can only *flip* the presence bit of a file over the file (vs. inverted) index to indicate whether it contains a keyword (vs. pointing to all the files containing a particular keyword). Without another search (or keeping a local copy of the database, which nullifies outsourcing), the client can hardly confirm any insertion/deletion operation.

They are not necessary for range DSSE (*e.g.*, MoRSE and $(\mathsf{RS})^2$). We remark that other techniques, such as puncturable encryption, can achieve backward privacy without another communication round [4].

7.4 Attack on Scheme-B [38]

We have the following observations of scheme-B [38]. If a range covering a specific keyword was searched before, the update for this keyword (*e.g.*, add a file containing this keyword) would let the server operate over at least one location accessed in the previous search. In other words, scheme-B leaks to the server whether the keyword in a later update is contained in any previous search, which obviously violates Definition 4.1. The construction is thus not forward-secure as claimed. (The update leakage $\mathcal{L}^{\mathsf{Upd}}$ of scheme-B is claimed [38] to be the number of updates made to the keyword w and when the update happened. As we show, it leaks more.) We then concretely show how an adversary with the power of file injection would exploit this information to efficiently determine the queried range.

Consider the threshold setting [37] in Section 3.2. The server first injects $\lceil |\mathcal{W}|/T \rceil - 1$ files, each contains T contiguous keywords in the keyword space $\mathcal{W}$, and records the accessed locations for each injected file. For a target search query from the client, the server selects the previous injected files associated with at least one identical location accessed in this search phase. The server generates a union set of contiguous keywords in the selected files, whose size is an integral multiple of T. Specifically, if the set size is nT and w_q is the number of keywords within the target range, the parameters satisfy the relation that $w_q < nT < w_q + 2T$.

For the first and last T contiguous keywords in the selected set, the server adaptively injects files to accurately determine the target range. Concretely, for the first T keywords, the server injects a file containing the first half of these keywords. If the update does not access any locations overlapped with those of the search, the server injects a file containing the first $T/4$ keywords of the second half; otherwise, it injects a file containing the first $T/4$ keywords of the first half. The operation repeats until the server rules out all false keywords whose updates do not access any location in the target search, *i.e.*, these keywords are out of the searched range. Above operations are also executed over the last T contiguous keywords in the selected set. Finally, we can figure out the range for the previous search.

This variant of attack against scheme-B exploits its lack of forward security. The server adaptively injects only $\mathcal{O}(\log T)$ files after fixing a rough range of size nT.

8 Experimental Evaluation

We implement our modular construction MoRSE and reduced-storage construction $(\mathsf{RS})^2$ in C++11. We instantiate the underlying DSSE of MoRSE with an adaptation of FASTIO, a recent forward-secure single-keyword DSSE scheme [30]. For cryptographic functions, we instantiate our PRFs and hash functions with AES-128 and SHA-256 from crypto++ library, respectively. For non-cryptographic parts, we store the maps with RocksDB and build the communication between the client and the server with gRPC. The server is equipped with a single Intel Core i7-4790 3.60GHz CPU and 16GB of RAM, and the client with a single Intel Core i5-6500 3.20GHz CPU and 8GB of RAM in a LAN setting.

Our experiments consider the keyword domain to be $\mathcal{W} = \{0, \ldots, 1048575\}$ of size 2^{20}. For comparison, we also implement FBDSSE-RQ [39] with 10^6 as its file-index length, which upper-bounds the number of allowed files and governs its time for an atomic operation.

8.1 Update and Search Evaluation

Table 4 shows the time needed to update keyword-file pairs. We measure the average time of updating a single keyword-file pair over $\{10^3, 10^4, 10^5, 10^6\}$ update operations (with $9 \cdot \{10^2, 10^3, 10^4, 10^5\}$ distinct keywords, respectively). MoRSE and $(\mathsf{RS})^2$ outperform FBDSSE-RQ [39] as they do not need to rebuild indexes. $(\mathsf{RS})^2$ is a bit slower than MoRSE for its reduced client storage.

We calculate how many search tokens our schemes (and FBDSSE-RQ) can reduce from single-keyword

Table 4. Update Performance

Range DSSE Scheme	Time for updating n pairs (s)				Avg. time
	10^3	10^4	10^5	10^6	for $n = 1$ (s)
FBDSSE-RQ	3.09	29.12	316.75	3029.34	3.04×10^{-3}
MoRSE	1.43	14.15	141.41	1483.32	1.47×10^{-3}
(RS)^2	3.38	22.80	223.42	2225.14	2.23×10^{-3}

Table 5. Average Number of Search Tokens

Scheme/Queried Range	5×10^2	5×10^3	5×10^4
Single-keyword DSSE	5×10^2	5×10^3	5×10^4
MoRSE/(RS)^2	8.17	9.67	13.16

Table 6. Search Performance with or without (w/o) Cache

Range DSSE Scheme	Time for searching a range of ℓ (ms)					
	$\ell = 5 \times 10^2$		$\ell = 5 \times 10^3$		$\ell = 5 \times 10^4$	
	w/o	Cache	w/o	Cache	w/o	Cache
FBDSSE-RQ	157.2	46.8	207.8	60.7	355.2	206.2
MoRSE	71.3	41.8	110.3	57.4	283.0	195.1
(RS)^2	74.2	45.4	121.8	61.8	298.7	204.2

Table 7. Client Storage Overhead

Range DSSE Scheme	Storage for n keyword-file pairs (MB)			
	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
FBDSSE-RQ	0.87	6.21	38.75	125.81
MoRSE	0.86	6.19	37.08	123.17
(RS)^2-0	0.16	1.31	12.32	38.87
(RS)^2-20	0.16	1.32	12.34	38.90
(RS)^2-200	0.17	1.68	14.92	46.12
(RS)^2-2000	0.30	2.45	18.60	55.23
(RS)^2-20000	0.63	4.97	28.33	71.02

Table 8. Number of Tuples stored in the Local State

Range DSSE Scheme	# of state tuples for n keyword-file pairs			
	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
FBDSSE-RQ	11178	78319	454955	1610020
MoRSE	11178	78319	454955	1610020
(RS)^2-0	996	9955	95457	644650
(RS)^2-20	1009	10005	95572	644814
(RS)^2-200	1101	10487	96710	646445
(RS)^2-2000	1765	14536	106126	659573
(RS)^2-20000	4350	33076	165537	754111

DSSE. The number of search tokens is exactly the size of BRC for the queried range, while it is the size of the queried range in single-keyword DSSE. We issue multiple queries with ranges of 5×10^i, $i \in \{2,3,4\}$, uniformly distributed over the keyword domain, and measure the *average number* of search tokens. As shown in Table 5, the gap can be large, especially when the query covers a wide range. Note that the network latency is a common bottleneck for DSSE in practice; we save the communication overhead and the time needed for Search.

We also measure the search time from the start of token generation until the client receives the search results over a database of 5×10^7 keyword-file pairs with 5×10^5 distinct keywords. We use multiple queries with ranges of 5×10^i, $i \in \{2,3,4\}$, while the range boundaries are uniformly distributed over the keyword domain. Table 6 shows that both MoRSE and (RS)^2 outperform FBDSSE-RQ in search performance (and caching was not implemented by their paper [39]). Searching in MoRSE is slightly faster than (RS)^2 since (RS)^2 reduces the client storage via the histUptCnt computation.

8.2 Storage Evaluation

We first measure the client storage for outsourcing a database of size 10^3–10^6. We issue a different number of queries $(0, 20, \dots, 20000)$ since the storage of (RS)^2 increases with the number of newly queried elements (*i.e.*, s in Table 2). In Tables 7 and 8, s is appended to the scheme name (RS)^2. Each query, covering a range of 500, is uniformly distributed over the keyword domain.

Table 7 illustrates that our (RS)^2 requires less client storage than our modular construction MoRSE or FBDSSE-RQ even after many range queries. We remark that it only reflects the client storage to some degree, as RocksDB requires storage for the long-term startup information and periodically updated log files. The performance may vary when instantiated differently.

We also measure the number of state tuples being stored in the same setting. (RS)^2 only stores information of leaf nodes in the implicit binary tree, while MoRSE/FBDSSE-RQ additionally stores those from the leaf to the root. Table 8 confirms that the client storage of (RS)^2 is significantly less than others, *e.g.*, it is essentially the number of existing keywords for (RS)^2-0.

9 Conclusion

We propose a generic construction and a reduced-client-storage construction of dynamic searchable symmetric encryption for range queries, both forward-secure and backward-secure. Our modular approach frees us from many design-specific shortcomings of existing works, allowing our schemes to be competitive in security, efficiency, and features. We also empirically show their high efficiency on a million scale. We left it as future work to consider multi-writer extensions [34].

10 Acknowledgements

Sherman Chow is supported by General Research Fund (CUHK 14210217 and CUHK 14209918) from Research Grant Council, Hong Kong. We thank the anonymous reviewers for their comments in improving this work.

References

[1] G. Amjad, S. Kamara, and T. Moataz. Breach-resistant structured encryption. *PoPETs*, (1):245–265, 2019.

[2] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[3] R. Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *CCS*, 2016.

[4] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[5] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*, 2014. Full version at Crypto. ePrint 2014/853.

[6] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO Part I*, 2013.

[7] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[8] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*, 2010.

[9] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[10] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. N. Garofalakis. Practical private range search revisited. In *SIGMOD*, 2016.

[11] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *ESORICS Part II*, 2015.

[12] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, 2018.

[13] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *SP*, 2017.

[14] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *CCS*, 2014.

[15] S. Kamara and T. Moataz. SQL on structurally-encrypted databases. In *ASIACRYPT Part I*, 2018.

[16] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT*, 2019.

[17] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[18] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC*, 2013.

[19] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS*, 2012.

[20] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[21] F. Kerschbaum and A. Tueno. An efficiently searchable encrypted data structure for range queries. In *ESORICS Part II*, 2019.

[22] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable pseudorandom functions and applications. In *CCS*, 2013.

[23] M. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*, 2018.

[24] R. W. F. Lai and S. S. M. Chow. Structured encryption with non-interactive updates and parallel traversal. In *ICDCS*, 2015.

[25] R. W. F. Lai and S. S. M. Chow. Parallel and dynamic structured encryption. In *SecureComm*, 2016.

[26] R. W. F. Lai and S. S. M. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, 2017.

[27] E. A. Markatou and R. Tamassia. Mitigation techniques for attacks on 1-dimensional databases that support range queries. In *ISC*, 2019.

[28] W. Ogata and K. Kurosawa. Efficient no-dictionary verifiable searchable symmetric encryption. In *FC*, 2017.

[29] F. I. Rusu. *Sketches for aggregate estimations over data streams*. PhD thesis, University of Florida, 2009.

[30] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao. Forward private searchable symmetric encryption with optimized I/O efficiency. *IEEE TDSC*, 17(5):912–927, 2020.

[31] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[32] J. Wang and S. S. M. Chow. Forward and backward-secure range-searchable symmetric encryption. Cryptology ePrint Archive, Report 2019/497, 2019. Preliminary version of this paper, submitted to ePrint on 14-May-2019.

[33] J. Wang and S. S. M. Chow. Simple storage-saving structure for volume-hiding encrypted multi-maps (A slot in need is a slot indeed). In *DBSEC*, 2021.

[34] J. Wang and S. S. M. Chow. *Omnes pro uno*: Practical multi-writer encrypted database. In *USENIX Security*, 2022.

[35] J. Wang, M. Du, and S. S. M. Chow. Stargazing in the dark: Secure skyline queries with SGX. In *DASFAA Part III*, 2020.

[36] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang. ServeDB: Secure, verifiable, and efficient multidimensional range queries on outsourced database. In *ICDE*, 2019.

[37] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, 2016.

[38] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk. Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In *ESORICS Part II*, 2018.

[39] C. Zuo, S. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu. Forward and backward private DSSE for range queries. *IEEE Trans. Dependable Sec. Comput. (TDSC)*, 2020. Early Access. Also available as Cryptology ePrint Archive: Report 2019/1240, submitted to ePrint on 22-Oct-2019.

A Review of FASTIO

Figure 5 reviews the FASTIO scheme [30]. We refer the readers to [30] for a more detailed treatment. H_1 and H_2 are cryptographic hash functions of appropriate domains and output lengths. F denotes PRF using key k_s.

For Setup, the client outputs three empty maps: $\mathbf{W}$ stores a random key and a counter (*i.e.*, state) per keyword, $\mathbf{T}_\mathsf{e}$ stores the encrypted index, and $\mathbf{T}_\mathsf{c}$ stores previous search results if any keyword was searched before. The client keeps k_s and $\mathbf{W}$. $\mathbf{T}_\mathsf{e}$ and $\mathbf{T}_\mathsf{c}$ are outsourced.

During updates, the client first retrieves the state of the keyword to be updated from $\mathbf{W}$. The state is used to generate the encryption of the update and determine the location to store it in $\mathbf{T}_\mathsf{e}$, so the server can update $\mathbf{T}_\mathsf{e}$ accordingly. Each update increments the counter by 1. To search a keyword, the client provides its pseudonym to retrieve its previous search results in $\mathbf{T}_\mathsf{c}$ and its state to retrieve its recent updates in $\mathbf{T}_\mathsf{e}$; the server retrieves entries from the related locations of $\mathbf{T}_\mathsf{c}$ and $\mathbf{T}_\mathsf{e}$, and updates $\mathbf{T}_\mathsf{c}$ with the new results. After the search, the client refreshes the random key and resets the counter of the keyword for forward security.

Regarding efficiency, it takes $\mathcal{O}(1)$ time for both the client and the server to update. Meanwhile, the client can generate the search token in $\mathcal{O}(1)$ time. As previous search results are archived in $\mathbf{T}_\mathsf{c}$, it only requires $\mathcal{O}(\overline{n_q})$ for the server to perform a search query, where $\overline{n_q}$ is the number of updates since the last search of the same keyword. As the client needs to store a key and a counter per keyword in $\mathbf{W}$, the client-side storage overhead is $\mathcal{O}(W)$, where W is the number of distinct keywords.

B Security of MoRSE(Π)

Our modular construction MoRSE(Π) relies on an underlying forward-secure DSSE Π, which usually leaks the search pattern (sp) and the update history (hist) for the keywords (*e.g.*, [3, 26, 30]). Beyond that, the only information leaked by our construction for Search is a partitioning of sp and hist, which exposes the overlapping nodes induced by the overlapping ranges. We summarize the leakages as follows.

- $\mathcal{L}^{\mathsf{Stp}} = \perp$.
- $\mathcal{L}^{\mathsf{Srch}}(q) = ((\mathsf{sp}(\tau), \mathsf{hist}(\tau))_{\tau \in \mathsf{BRC} \text{ of } q})$.
- $\mathcal{L}^{\mathsf{Upd}}(op, up) = |\mathcal{W}|$.

Theorem 1. Let Π be a forward-secure DSSE scheme. Our modular construction is $\mathcal{L}$-adaptively secure for $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}\}$ as defined above.

Proof. To see the security of our modular construction, we extend the simulator of the underlying forward-secure DSSE scheme Π and explain how it can deal with the simulation for (range) queries.

The simulator directly invokes the simulator algorithm of Π to simulate Setup and Update. For $\mathcal{L}^{\mathsf{Stp}}$, our scheme leaks nothing. $\mathcal{L}^{\mathsf{Upd}}$ only leaks the keyword space size $|\mathcal{W}|$, a constant value since the initialization.

To simulate a range query q, the simulator exploits the search pattern sp to determine if any element in BRC of q has been retrieved before. Then the simulator updates hist for elements that exist in any previous search and have been updated afterward. As a result, it leaks the overlapping elements induced by the overlapping queried ranges. With $\mathcal{L}^{\mathsf{Srch}}$, which includes the leakage of underlying DSSE, the simulator invokes the simulator algorithm of Π for every element in BRC of q to simulate Search.

Since Π is forward secure and our construction only additionally leaks the size of keyword space $|\mathcal{W}|$ during Update, our construction is forward secure and thus immune to the adaptive file injection attack [37]. □

C Security of (RS)^2

The leakage functions $\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}$ of our reduced-storage range DSSE scheme (RS)^2 are as follows.

- $\mathcal{L}^{\mathsf{Stp}} = \perp$.
- $\mathcal{L}^{\mathsf{Srch}}(q) = ((\mathsf{sp}(\tau), \mathsf{hist}(\tau))_{\tau \in \mathsf{BRC} \text{ of } q})$.
- $\mathcal{L}^{\mathsf{Upd}}(op, up) = |\mathcal{W}|$.

Theorem 2. Let F be a pseudorandom function, H_1 and H_2 be two hash functions modeled as random oracles. Our reduced-storage range DSSE scheme is $\mathcal{L}$-adaptively secure for $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}\}$ as above.

Proof. We derive a game sequence from the real-world game $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ to the last game, which is exactly the ideal-world game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$. By showing that each game (except the first) is indistinguishable from its previous one, we conclude that the adversary cannot distinguish $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ from $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$ with non-negligible probability. Without loss of generality, we assume that adversary $\mathcal{A}$ makes at most q_1 and q_2 queries to the

```
Setup$(1^\lambda)$
$k_s \leftarrow_\$ \{0,1\}^\lambda; \mathbf{W}, \mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c} \leftarrow \emptyset$
return $(k_s, \mathbf{W}, (\mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c}))$

Update$(k_s, \mathbf{W}, op, (w, id); \mathbf{T}_\mathsf{e})$
Client :
    $(k_\mathcal{T}, c) \leftarrow \mathbf{W}[w]$
    if $\mathbf{W}[w] = \perp$ then $k_\mathcal{T} \leftarrow_\$ \{0,1\}^\lambda; c := 0$
    $u := H_1(k_\mathcal{T} || (c+1))$
    $e := (id||op) \oplus H_2(k_\mathcal{T} || (c+1))$
    $\mathbf{W}[w] := (k_\mathcal{T}, c+1)$
    send $(u, e)$ to server
Server :
    $\mathbf{T}_\mathsf{e}[u] := e$
```

```
Search$(k_s, \mathbf{W}, w; \mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c})$
Client :
    $(k_\mathcal{T}, c) \leftarrow \mathbf{W}[w]$
    if $(k_\mathcal{T}, c) = \perp$ then return $\perp$
    $nym_w \leftarrow F(k_s, w)$
    if $c \neq 0$ then
      $k_w := k_\mathcal{T}$
      $k_\mathcal{T} \leftarrow_\$ \{0,1\}^\lambda$
      $\mathbf{W}[w] := (k_\mathcal{T}, 0)$
    else
      $k_w := \perp$
    endif
    send $(nym_w, k_w, c)$ to server
Server :
    $\mathcal{R} := \mathbf{T}_\mathsf{c}[nym_w]$
    if $k_w = \perp$ then return $\mathcal{R}$
    for $i := 1$ to $c$ do
      $u_i := H_1(k_w || i)$
      $(id, op) := \mathbf{T}_\mathsf{e}[u_i] \oplus H_2(k_w || i)$
      if $op = \mathsf{del}$ then $\mathcal{R} := \mathcal{R} \setminus \{id\}$
      else $\mathcal{R} := \mathcal{R} \cup \{id\}$
      delete $\mathbf{T}_\mathsf{e}[u_i]$
    endfor
    $\mathbf{T}_\mathsf{c}[nym_w] := \mathcal{R}$
    send $\mathcal{R}$ to client
```

Fig. 5. The Construction of FASTIO

H_1 oracle and the H_2 oracle, respectively. The output lengths of F, H_1, and H_2 are λ, μ_1, and μ_2, respectively.

Game G_0: G_0 is **Real** in the real world. $\Pr[\mathbf{Real}_\mathcal{A}(1^\lambda) = 1] = \Pr[G_0 = 1]$.

Game G_1: Instead of invoking PRF with K_2 when generating nym_w, G_1 maintains a map $\mathbf{T}_\mathsf{n}$ to store (τ, nym_w) pairs. When a new τ is queried, G_1 returns a random string from $\{0,1\}^\lambda$ and stores it in $\mathbf{T}_\mathsf{n}[\tau]$. Other parts of G_1 are exactly the same as those of G_0. If an adversary can distinguish G_0 from G_1, we can distinguish between the PRF and a truly random function.

Game G_2: We obtain G_2 from G_1, similar to how G_1 is derived from G_0. Instead of invoking PRF with K_1, G_2 maintains a map $\mathbf{T}_\mathsf{s}$ and processes the map like $\mathbf{T}_\mathsf{n}$ in G_1. With the same argument before, G_1 and G_2 are indistinguishable.

Game G_3: Instead of querying H_1 in Update, *i.e.*, $H_1(k_\mathcal{T}||(c+1))$, G_3 picks a random string from $\{0,1\}^{\mu_1}$ and stores it in a map $\mathbf{T}_\mathsf{a}$:

$$addr \leftarrow_\$ \{0,1\}^{\mu_1}, \qquad \mathbf{T}_\mathsf{a}[k_\mathcal{T}||c+1] := addr.$$

Then, during Search of the client, if $k_w \neq \perp$ when (nym_w, k_w, c) is collected in t_q, we update the reference table H_1 for the random oracle H_1 by setting $\mathsf{H}_1[k_w||i] := \mathbf{T}_\mathsf{a}[k_w||i]$ for $i \in [1, c]$.

In G_3, $addr$ for tuple $(k_\mathcal{T}, c{+}1)$ is generated in Update but will not be updated in H_1 until a corresponding search query is executed. If the adversary queries H_1 for $(k_\mathcal{T}, c{+}1)$ before Search, the random oracle will return a random string. With a non-negligible probability, it will be different from the one programmed for $\mathsf{H}_1[k_w||i]$ (*i.e.*, $\mathsf{H}_1[k_\mathcal{T}||i]$) later during Search. Once this inconsistency is observed, the adversary could figure out the game is G_3. As other parts of G_3 are exactly the same as those of G_2, $\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq \Pr[\mathsf{BAD}]$, where BAD is the event that the inconsistency happens.

Since $k_\mathcal{T}$ is sampled from $\{0,1\}^\lambda$, the probability that the adversary queries H_1 for $k_\mathcal{T}$ is $2^{-\lambda}$. As we assume the adversary could at most make q_1 queries to the H_1 oracle, we have $\Pr[\mathsf{BAD}] \leq \frac{q_1}{2^\lambda}$, which is negligible. Thus, we conclude that G_2 and G_3 are indistinguishable.

Game G_4: We obtain G_4 from G_3 in a similar way as how G_3 is derived from G_2. Instead of querying H_2 in Update (*i.e.*, $val := (id||op) \oplus H_2(k_\mathcal{T}||(c+1))$), G_3 picks a random string from $\{0,1\}^{\mu_2}$ and stores it in a map $\mathbf{T}_\mathsf{a}$:

$$v \leftarrow_\$ \{0,1\}^{\mu_2}, \qquad \mathbf{T}_\mathsf{v}[k_\mathcal{T}||c+1] := v,$$
$$val := (id||op) \oplus v.$$

Then, during Search of the client, if $k_w \neq \perp$ when (nym_w, k_w, c) is collected in t_q, we update the reference table H_2 for the random oracle H_2 by setting $\mathsf{H}_2[k_w||i] := \mathbf{T}_\mathsf{v}[k_w||i]$ for $i \in [1, c]$. Like G_3, the probability that the adversary discovers the inconsistency of the random oracle H_2 is at most $\frac{q_2}{2^\lambda}$, which is negligible. Thus, we conclude that G_3 and G_4 are indistinguishable.

Game G_5: We present G_5 in Figure 6. The server part is omitted as all our protocols are round-optimal, and the transcripts of the client are not influenced by the server. For every element, we use *UpTkSet* to record all updates over it since its last search. Different from G_4, k_w (*i.e.*, $k_\mathcal{T}$) is sampled during Search. Also, instead of directly mapping $k_\mathcal{T}||(c+1)$ to the values picked for $\mathbf{T}_\mathsf{v}$ and $\mathbf{T}_\mathsf{a}$, we implicitly map $k_\mathcal{T}||(c+1)$ to the global time index via *UpTkSet* and update the random oracle accordingly during Search.

We argue that G_4 and G_5 are indistinguishable. Update protocols in both games output two uniformly random values. The distributions of (nym_w, k_w, c), *i.e.*, the client-side output of Search, are the same. So, $\Pr[G_4 = 1] = \Pr[G_5 = 1]$.

Game G_6: G_6 is exactly **Ideal**, where the simulator $\mathcal{S}$ generates a view only based on the leakage function $\mathcal{L}$. The only update leakage $\mathcal{L}^{\mathsf{Upd}}$ is the size of the keyword domain $|\mathcal{W}|$. The search leakage $\mathcal{L}^{\mathsf{Srch}}$ contains the search pattern sp and the update history hist for every element in BRC of the queried range.

We present G_6 in Figure 7. $\underline{u} := \mathsf{sp}(\tau).\mathtt{min}$ denotes the timestamp when τ is retrieved for the first time. Instead of using the unknown τ directly, the simulator uses $\underline{u}$ to uniquely identify the items related to τ from $\mathbf{T}_\mathsf{s}$ and $\mathbf{T}_\mathsf{n}$. $\overline{u} := \mathsf{sp}(\tau).\mathtt{max}$ denotes the timestamp when τ is retrieved last time. We define $\mathsf{hist}(\tau)^{>\overline{u}}$ as $\{(u, op, id) | u > \overline{u} \wedge (u, op, w, id) \in Q\}$. We use $|\mathsf{hist}(\tau)^{>\overline{u}}|$ to indicate whether there is any update after τ is retrieved last time. Then we program the random oracles accordingly with $\mathsf{hist}(\tau)^{>\overline{u}}$. The view of G_6 is exactly the same as that of G_5. So $\Pr[G_5 = 1] = \Pr[G_6 = 1]$.

By combining the above (in)equalities, we have $|\Pr[\mathbf{Real}_\mathcal{A}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$. □

D More Related Work

Update in (range) DSSE is known to be tricky because it complicates leakage in general for maintaining search and update efficiency. Stefanov *et al.* [31] designed an oblivious sorting protocol to perform housekeeping over the deleted records and minimizes update leakages. Some DSSE schemes [18, 25, 26] support parallel search and update. Kamara and Papamanthou [18] use a red-black tree of indicator bit-vectors, while Lai and Chow [25] realize a regular inverted index. Parallel DSSE is tricky since maintaining the data-structure invariant for parallelism could make the update too leaky. A subsequent work of Lai and Chow [26] proposed a new data structure called cascaded triangle, which is a parallel-traversable structure. A distinctive feature is that any update can be localized without affecting parallelism. In other words, any potential update leakage is thus minimized.

Proposing modular design is instrumental in understanding the intrinsic values of different techniques. Meanwhile, formulating a formal security definition requires a dedicated effort. Lai and Chow [26] also proposed a generic transformation from any DSSE to forward-secure DSSE and a generalized definition of forward privacy. Notably, it guarantees protection not via omission of the inputs to the leakage function, but the typical indistinguishability framework subject to a general condition over the inputs. Most definitions [3, 7] are restricted to a specific list of allowed inputs.

Supporting richer queries is a popular pursuit. Cash *et al.* [6] consider boolean queries, mainly conjunctive keyword searches, with an information retrieval perspective on optimal search efficiency and its implication in inherent leakage. Faber *et al.* [11] built on top of their work [6] for boolean queries over range, substring, wildcard, and phrase sub-queries. Lai and Chow [24–26] generalize the notion of searchable encryption over typical keyword-file relation into the notion of structured encryption (STE) with meta-data [8] over bipartite graphs. The meta-data is semi-private in the sense that it remains secure until locked by a search token. It can be used to support richer queries, such as rank queries, by using it to store ranking information.

SEAL [9], a recent oblivious-primitive-based SSE, can be extended to static range SSE with prior techniques [10]. It aims for a set of obliviousness/efficiency trade-offs but still incurs high overheads in all aspects.

Order-preserving encryption (OPE), which preserves the plaintext order in the ciphertext domain [2], is another way to support range queries over encrypted data. Unfortunately, it is often too leaky, especially under plaintext-guessing attacks [13]. Kerschbaum and Tueno [21] proposed a linear server-side storage scheme (*cf.*, replicated index [10, 11, 38]) that is free from the above attack. However, a query takes multiple rounds.

Wu *et al.* [36] consider multi-dimensional range queries using a hierarchical encoding system to map data of different dimensions to a single dimension. The encoded data are also arranged in the leaf nodes of a binary tree. Non-leaf nodes are associated with Bloom filters to determine whether a query involves its descendants, incurring false positives. This is especially dangerous in the DSSE context since the adversary can break the security trivially once they discover a false positive, which the probability is often not confined to be negligible in the security parameter. The scheme is claimed to support updates, yet the security analysis does not consider the update leakage, let alone forward or backward security. Their work also considers verifiability, which (no-dictionary) generic upgrade exists [28].

```
Setup(1^λ)
$\mathbf{T}_a, \mathbf{T}_v, UpTkSet \leftarrow \emptyset$
$u := 0$

Update($k_1, \mathbf{W}, op, up; \mathbf{T}_e$)
Client :
    Parse $up$ as $(w, id)$
    $t_u \leftarrow \emptyset$
    $w_0 \cdots w_m := [w]_{bin}$
    for $i := m$ to $0$ do
        Append $(u, op, id)$ to $UpTkSet[w_0 \cdots w_i]$
        $\mathbf{T}_a[u] \leftarrow_\$ \{0,1\}^{\mu_1}$
        $\mathbf{T}_v[u] \leftarrow_\$ \{0,1\}^{\mu_2}$
        $t_u := t_u \cup \{(\mathbf{T}_a[u], \mathbf{T}_v[u])\}$
        $u := u + 1$
    endfor
    send $t_u$ to server

Search($(k_1, k_2), \mathbf{W}, q; \mathbf{T}_e, \mathbf{T}_c$)
Client :
    $t_q \leftarrow \emptyset$
    for $\tau \in$ BRC of $q$ do
        if $\mathbf{T}_n[\tau] = \perp$ then
            $\mathbf{T}_n[\tau] \leftarrow_\$ \{0,1\}^\lambda$
        endif
        $nym_w := \mathbf{T}_n[\tau]$
        $c := |UpTkSet[\tau]|$
        if $c = 0$ then
            $k_w := \perp$
        else
            $k_w \leftarrow_\$ \{0,1\}^\lambda$
            if $\mathbf{T}_s[\tau] = \perp$ then
                $\mathbf{T}_s[\tau] := k_w$
            endif
            Parse $UpTkSet[\tau]$ as $((u_1, op_1, id_1), \ldots, (u_c, op_c, id_c))$
            for $i := 1$ to $c$ do
                $H_1[k_w || i] := \mathbf{T}_a[u_i]$
                $H_2[k_w || i] := (id_i || op_i) \oplus \mathbf{T}_v[u_i]$
            endfor
        endif
        $t_q := t_q \cup \{(nym_w, k_w, c)\}$
        $UpTkSet[\tau] := \emptyset$
    endfor
    send $t_q$ to server
```

Fig. 6. Description of Game G_5

```
$\mathcal{S}$.Setup(1^λ)
$\mathbf{T}_a, \mathbf{T}_v \leftarrow \emptyset$
$u := 0$

$\mathcal{S}$.Update($|\mathcal{W}|$)
Client :
    $t_u \leftarrow \emptyset$
    for $i := 0$ to $\log |\mathcal{W}|$ do
        $\mathbf{T}_a[u] \leftarrow_\$ \{0,1\}^{\mu_1}$
        $\mathbf{T}_v[u] \leftarrow_\$ \{0,1\}^{\mu_2}$
        $t_u := t_u \cup \{(\mathbf{T}_a[u], \mathbf{T}_v[u])\}$
        $u := u + 1$
    endfor
    send $t_u$ to server

$\mathcal{S}$.Search($(\mathsf{sp}(\tau), \mathsf{hist}(\tau))_{\tau \in \text{BRC of } q}$)
Client :
    $t_q \leftarrow \emptyset$
    for each $(\mathsf{sp}(\tau), \mathsf{hist}(\tau))$ do
        $\underline{u} := \mathsf{sp}(\tau).\mathsf{min}, \overline{u} := \mathsf{sp}(\tau).\mathsf{max}$
        if $\mathbf{T}_n[\underline{u}] = \perp$ then
            $\mathbf{T}_n[\underline{u}] \leftarrow_\$ \{0,1\}^\lambda$
        endif
        $nym_w := \mathbf{T}_n[\underline{u}]$
        $c := |\mathsf{hist}(\tau)^{>\overline{u}}|$
        if $c = 0$ then
            $k_w := \perp$
        else
            $k_w \leftarrow_\$ \{0,1\}^\lambda$
            if $\mathbf{T}_s[\underline{u}] = \perp$ then
                $\mathbf{T}_s[\underline{u}] := k_w$
            endif
            Parse $\mathsf{hist}(\tau)^{>\overline{u}}$ as $((u_1, op_1, id_1), \ldots, (u_c, op_c, id_c))$
            for $i := 1$ to $c$ do
                $H_1[k_w || i] := \mathbf{T}_a[u_i]$
                $H_2[k_w || i] := (id_i || op_i) \oplus \mathbf{T}_v[u_i]$
            endfor
        endif
        $t_q := t_q \cup \{(nym_w, k_w, c)\}$
    endfor
    send $t_q$ to server
```

Fig. 7. Description of Simulator $\mathcal{S}$