# Forward and Backward-Secure Range-Searchable Symmetric Encryption

Jiafan Wang and Sherman S. M. Chow

Department of Information Engineering
Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{wj016,sherman}@ie.cuhk.edu.hk

**Abstract.** Dynamic searchable symmetric encryption (DSSE) allows a client to search or update over an outsourced encrypted database. Range query is commonly needed (AsiaCrypt'18) but order-preserving encryption approach is vulnerable to reconstruction attacks (SP'17). Previous range-searchable schemes (SIGMOD'16, ESORICS'18) require an ad-hoc instance of encrypted database to store the updates and/or suffer from other shortcomings, some brought by the usage of asymmetric primitives. In this paper, with our encrypted index which enables queries for a sequence of contiguous keywords, we propose a *generic upgrade* of any DSSE to support range query (a.k.a. range DSSE), and a concrete construction which provides *a new trade-off of reducing the client storage* to "reclaim" the benefits of outsourcing.
Our schemes achieve forward security, an important property which mitigates file injection attacks. We identify *a variant of file injection attack* against a recent solution (ESORICS'18). We also extend the definition of backward security to range DSSE and show our schemes are compatible with a generic transformation for achieving backward security (CCS'17). We comprehensively analyze the computation and communication overheads including some parts which were *ignored* in previous schemes, e.g., index-related operations in the *client side*. Our experiments demonstrate the high efficiency of our schemes.

**Keywords:** dynamic symmetric searchable encryption · range query

## 1 Introduction

Searchable symmetric encryption (SSE) allows a client to outsource encrypted data to an untrusted server. The client can issue search queries over the encrypted data to retrieve files containing specific keywords, with leakage to the server under precise control. Various SSE schemes with different trade-offs among efficiency, security, and functionality have been proposed. A milestone is dynamic SSE (DSSE) by Kamara *et al.* [18], which further enables the client to efficiently update over the outsourced data without re-encrypting from scratch. Some later schemes [17,23] also support parallelism.

In the context of DSSE, *forward security* [29] requires newly updated data remains private against the server who possesses some knowledge about previous queries. Leakages of non-forward-secure schemes [7,33] can be exploited.

Zhang *et al.* [33] clearly point out that forward security can mitigate the most efficient attack proposed by them. There are a number of forward-secure DSSE schemes based on either advanced cryptographic primitives [5,6] or novel data structures [24,31,28]. Among them, Bost [5] gives the formal definition of forward security and Lai and Chow give a generalized version [24]. Security (*e.g.*, security against chosen keyword attacks, forward security) is established by formally confining the leakage.

Also regarding updates, *backward security* ensures that results which are deleted will be indeed inaccessible to the server afterwards. More specifically, for a keyword-based DSSE, after a keyword-file pair has been added and then deleted, subsequent search queries of this keyword reveal nothing about the file in this updated pair. Bost *et al.* [6] define three levels of backward security. The techniques for backward security are still evolving [30,9].

Most existing forward/backward-secure DSSE schemes or generic transformations for forward security [24] or backward security [6] only support single-keyword queries or similarity search [31]. Searching over a range of keywords is a common operation. To have meaningful support [12,10,11,34], it should be more efficient than searching each point in the range one by one. Handling range queries without order-preserving encryption is posed as an open problem for supporting SQL on encrypted databases [16].

Demertzis *et al.* [10] design range-queryable searchable encryption schemes. The core idea is to reduce the query to the multiple single-keyword queries of the underlying SSE using the range-covering techniques with tree-like indexes. However, their instantiations can only handle updates in batch by setting up a new instance. It is essential for DSSE to support *flexible update*, which enables any number of updates (even just one) to be processed in real-time without involving *a new instance* of encrypted indexes. It might be possible to instantiate their schemes with DSSE instead, yet there is neither security proof nor performance analysis for it. As we will illustrate, multiple inter-related issues are involved in designing such a scheme, such as forward security and client-side storage.

A recent work by Zuo *et al.* [34] exploits a similar tree-like index, which is essentially a complete binary tree with leaf nodes representing keywords, to allow queries for a range of contiguous keywords (over consecutive leaf nodes). For each query, the client finds a set of minimum tree nodes which covers all keywords (leaf nodes) in the queried range, and generates search queries for the nodes in the set. Compared with searching for every single keyword in the range, the number of search queries is reduced. Yet, their schemes are not satisfactory.

### 1.1 Another Look at the State-of-the-Art

Zuo *et al.* [34] obtained two schemes by applying the above complete binary tree to a DSSE scheme [5] (scheme-A) or constructing a file index encrypted by homomorphic encryption [27] (scheme-B). Both possess some shortcomings.

Before searching/updating, the client needs to build a binary tree, whose number of leaf nodes equals to the maximum sequence number of existing keywords, which makes the search/update computation overhead linear in it. Also,

in the worst case [34] for a update, scheme-A requires the server to send back *all* updates since initialization.

Both constructions do not possess desirable features of some recent DSSE schemes, *e.g.*, physical deletion [1] for reclaiming space and caching the previous search results [28,15] for boosting search efficiency. Their use of asymmetric primitives (trapdoor permutation [5] or Paillier encryption [27]) also negatively impacts their performance.

In particular, scheme-B uses homomorphic encryption (HE) to encrypt as many bits as the maximum number of documents (each serves as a *presence bit* denoting whether a keyword is in the corresponding document) Using HE enables processing encrypted data, yet it also brings in a number of inconveniences.

- The number of documents supported by scheme-B is restricted by the security parameter of some *asymmetric* cryptographic primitives.
- Each query and index node bear the cost incurred by many "empty slots" especially when the pre-defined maximum is too large (when compared with the number of stored documents).
- This pre-defined maximum number also impacts the dynamism. Another instance is needed when adding a new document after the limit is reached.
- As the client needs to decrypt the ciphertext storing the matching identifiers, getting the results takes 2 rounds.
- The update operation (which crucially relies on the additive homomorphism of HE) can only switch the presence bit for the corresponding document. Without another search (or a local copy of the database), the client essentially cannot confirm insertion/deletion without the risk of error.

Table 1: Property Comparison of (Range) DSSE Schemes

| Scheme | Range Query | Update Flexibility | $\rightarrow$ | $\leftarrow$ | # | No False Positive | Symmetric-key Building Block |
|---|---|---|---|---|---|---|---|
| $\sum o\varphi o\varsigma$ [5] | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |
| FASTIO [28] | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Janus++ [30] | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Logarithmic-SRC [10] | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| Logarithmic-SRC-i [10] | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| Logarithmic-BRC/URC [10] | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Scheme-A [34] | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |
| Scheme-B [34] | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ |
| ServeDB [32] | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ |
| Our Generic Construction | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Our Less-Storage Construction | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

* $\rightarrow$: Forward Security, $\leftarrow$: Backward Security, #: Large Number of Documents

Table 2: Efficiency Comparison of Range DSSE Schemes

| Scheme | Client Computation | | Server Computation | | Communication | |
|---|---|---|---|---|---|---|
| | Search | Update | Search | Update | Search | Update |
| Log.-SRC [10] | See Sec. 2 | n/a | $\mathcal{O}(N)$ | n/a | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert+\epsilon)$ | n/a |
| Log.-SRC-i [10] | | | $\mathcal{O}(w_q+\lvert\mathtt{DB}(q)\rvert)$ | | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert+\epsilon')$ | |
| Log.-BRC/URC [10] | | | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$ | | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$ | |
| Scheme-A [34] | $\mathcal{O}(W_{\mathrm{x}}+w_q)$ | $\mathcal{O}(W_{\mathrm{x}})$ worst: $\mathcal{O}(N)^{\dagger}$ | $\mathcal{O}(n_q)$ | $\mathcal{O}(\log W)$ worst: $\mathcal{O}(N)^{\dagger}$ | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$ | $\mathcal{O}(\log W)$ worst: $\mathcal{O}(N)^{\dagger}$ |
| Scheme-B [34] | $\mathcal{O}(W_{\mathrm{x}}+w_q)$ | $\mathcal{O}(W_{\mathrm{x}})$ | $\mathcal{O}(w_q)^{\ddagger}$ | $\mathcal{O}(\log W)$ | $\mathcal{O}(w_q)^{\ddagger}$ | $\mathcal{O}(\log W)$ |
| Generic Range DSSE (using FASTIO [28]) | $\mathcal{O}(w_q)$ | $\mathcal{O}(\log\lvert\mathcal{W}\rvert)$ | $\mathcal{O}(\overline{n_q})$ | $\mathcal{O}(\log\lvert\mathcal{W}\rvert)$ | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$ | $\mathcal{O}(\log\lvert\mathcal{W}\rvert)$ |
| Range DSSE with Less Client Storage | $\mathcal{O}(w_q)$ | $\mathcal{O}(W+\log\lvert\mathcal{W}\rvert)$ | $\mathcal{O}(\overline{n_q})$ | $\mathcal{O}(\log\lvert\mathcal{W}\rvert)$ | $\mathcal{O}(\lvert\mathtt{DB}(q)\rvert)$ | $\mathcal{O}(\log\lvert\mathcal{W}\rvert)$ |

* $W$ is the number of distinct keywords in the database. $W_{\mathrm{x}}$ is the largest sequence number of existing keywords. $w_q$ is the number of keywords within a range query. $|\mathcal{W}|$ is the size of keyword space. $\overline{n_q}(< n_q)$ is the total number of updates that contain the keywords in a query $q$ since the last search of them. $n_q$ is the total number of updates (add + del) that contain the keywords in a range query $q$ since initialization. $|\mathtt{DB}(q)|$ ($= n_q$ if no deletion) is the number of files matching a range query $q$. $\epsilon/\epsilon'$ refers to the false positives. $N$ is the database size (largest here), *i.e.*, the total number of updates (add + del) since initialization. Special notes for complexity marked with $^{\dagger/\ddagger}$ can be found in Section 2/4.1.

More importantly, apart from the aforementioned issues of features and performance, inelegant design, and the use of public-key primitives, we point out a security issue. We design a variant of the adaptive file injection attack [33], which targets at determining the queried range instead of a single queried keyword. This attack breaks the claim about the forward security of scheme-B [34]. Our result may carry independent interest in studying the security of dynamic range-searchable encryption schemes (or range DSSE). In this paper, we keep our focus on provably secure schemes[1].

### 1.2 New Constructions

Under a range DSSE framework, we propose two schemes using range-covering techniques with tree-like indexes.

Our first scheme is a generic construction which transforms any typical DSSE to range DSSE. It can be viewed as a generalization of scheme-A [34] with a different implementation of the tree. As shown in Table 2, compared with scheme-A [34], our instantiation realizes a better performance on many criteria (*e.g.*, search complexity of both client and server) and avoids the worst update overhead, at a little more cost of client storage (Table 3).

We propose our second range DSSE scheme to reduce the client storage requirement. While most features remain competitive, this scheme guarantees the client storage for a complete database, before all possible non-overlapping range

[1] ✗ in Table 1 (*e.g.*, for forward security) means no proof ([32]) or a refuted one ([34]). More related works will be discussed in Section 2.

have been queried, is always less than that of previous round-optimal constructions. Table 2 shows that our schemes outperforms others under many criteria.

In previous works, the algorithmic details of the range-covering technique with tree-like index were not mentioned [10]. This was also omitted in the analysis of the client computation overhead [34]. We clearly illustrate how to realize it with an implicit tree-like index in our constructions and include the overhead in Table 2. It can serve as a reference for instantiating related structures.

Both our schemes realize all the desired properties listed in Table 1. Particularly, our schemes only use *symmetric* primitives and achieve forward security. With a two-roundtrip generic upgrade [6], they also provide backward security, under the definition we extend from single-keyword DSSE to range DSSE. Note that backward security is less studied [6,30,9], it turns out we need to be careful in defining it for range DSSE (see Section 3.4). We also devote Section 7 to explain how to obtain backward privacy starting from our two constructions.

Our schemes support both searched results archives and physical deletion, which are provided by recent DSSE schemes (*e.g.*, [28]). With a cache of the previous search results, a DSSE scheme can archive the file identifiers in previously authorized searches[2]. This saves the future search time. Physical deletion supports reclamation of space allocated for the deleted items. Reclaiming space at a later point can be non-trivial (possibly due to other requirements such as backward security). For example, it may take a secure two-party oblivious sorting protocol [29] to maintain the search efficiency of the underlying data structure while preventing the leakage incurred by the (logical) deletion and the reclaiming process, which takes $O(\log^2 N)$ update complexity where $N$ is the database size. It is desirable to ensure $O(1)$ (amortized) overhead for a single deletion.

Table 1 summarizes a comparison of various properties of (range) DSSE schemes. We can see that scheme-A of Zuo *et al.* [34] achieves the most desirable properties among the listed range DSSE schemes except ours. In Table 2, we show that both our constructions require less search computation overhead and avoid the massive update overhead in the worst case. Furthermore, we concretely list the client storage overhead of forward-secure range DSSE schemes in Table 3, which exhibits the advantage of our construction with less client storage. We implemented both constructions and comprehensively analyze their efficiency and security.

*Organization.* In Section 2, we discuss the related work for range query over encrypted data. Section 3 introduces the preliminaries and a framework of range DSSE, and discusses forward security and backward security in the context of range DSSE. Section 4 reviews file injection attack and presents a variant of the attack against an existing construction [34]. We propose two constructions of range DSSE in Sections 5 and 6 respectively, together with their efficiency and security analyses. Section 7 illustrates how to equip our constructions with backward security. Section 8 presents our implementations and evaluations.

[2] Archiving does not contradict with forward security since the scheme can still leak nothing during the update until the same search has been authorized again. Backward security is deferred to Section 7.

Table 3: Client Storage of *Forward-Secure* Range DSSE

| Scheme | Client Storage |
|---|---|
| Logarithmic-SRC [10] | $(2.5 - 2^{1-m})W - \log W$ |
| Logarithmic-SRC-i [10] | $(2.5 - 2^{1-m})W - \log W$ $+ (2.5 - 2^{1-\ell})D - \log D$ |
| Logarithmic-BRC/URC [10] | $2W$ |
| Scheme-A [34] | $2W$ |
| Our Generic Range DSSE | $2W + \log \lvert\mathcal{W}\rvert - \log W$ |
| Our Less-Client-Storage Range DSSE | $W + s$ |

We follow the notations in Table 2. $D$ is the number of distinct files in the database. $s$ is the number of distinct overlapping elements in all queries. (See Section 6.3). For simplicity, we assume that $W = 2^m$, $D = 2^\ell$, and existing keywords are contiguous.

## 2 Related Work

Range query is one of the most common query types. Naturally, there are multiple efforts for supporting range queries in SSE. Faber *et al.* [12] support range queries by building a binary tree for the dataset. The tree nodes covering the queried range are searched with an underlying static SSE called OXT [8] which supports disjunctive queries. Also, keywords are located in leaf nodes labeled according to the tree level. Adding/deleting node probably requires changing the existing information for consistency, and it is unclear how to do it efficiently.

The work by Demertzis *et al.* [10] utilizes a similar index as Faber *et al.* [12] and calls it range-covering techniques. Based on this index and its extensions, they propose six range-queryable SSE schemes with different trade-offs between security and efficiency. Among those adaptively-secure schemes[3], Logarithmic-SRC and Logarithmic-SRC-i (where SRC stands for Single Range Cover) provide constant-size search token for any queried range while incurring *false positive* in the result. Besides, Logarithmic-SRC-i requires an extra round of interaction between the client and the server. Logarithmic-BRC and Logarithmic-URC (where BRC and URC stand for Best Range Cover and Uniform Range Cover respectively) provide good performance without false positive.

Notably, all these constructions [10] are instantiated with static SSE scheme and the authors do not discuss whether they can be instantiated with DSSE. Instead, they design a method for batched updates, *i.e.*, create 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, for the same keyword, the client needs many tokens for different instances before the instances are merged, which increases the search overhead. Yet, the combination of static

[3] Constant-BRC and Constant-URC avoid extra storage overhead for the index. Yet, due to the inherent limitation of underlying delegatable pseudorandom functions [21], both cannot be proven secure against adaptive adversaries that are allowed to issue intersecting range queries, as acknowledged by the original paper [10].

SSE and batched updates makes the consideration of forward security a non-issue, as the subsequent updates, which happen in another instance, naturally cannot be linked with previous search queries.

Different realizations for index provide diverse trade-offs between the *search computation overhead* and the client-side storage overhead, *e.g.*, the client can reconstruct the index during search [34]. 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 index to the server inevitably violates security, we assume the client stores it locally. We can then estimate the client storage in the three different settings:

- Logarithmic-BRC/URC: The index is essentially a binary tree with leaf nodes representing keywords.
- Logarithmic-SRC: The index is the same as that of Logarithmic-BRC/URC except that there will be a common node between every two neighboring nodes at the same level.
- Logarithmic-SRC-i: Apart from a tree like that in Logarithmic-SRC for keywords, the same structure is used for existing files.

Table 3 provides the concrete number of stored tuples of Demertzis *et al.* [10], yet this evaluation is purely for the index and there may be some other storage overhead omitted, *e.g.*, the storage required by the underlying SSE.

Zuo *et al.* [34] propose two schemes with range-covering techniques. Their design is similar to Logarithmic-BRC/URC [10] yet supporting flexible updates. Before searching or updating, the client needs to reconstruct a binary tree whose number of leaf nodes equals to the maximum sequence number of existing keywords, for getting the relations between tree nodes. In other words, the tree size depends on existing keywords. This incurs heavy update overheads whenever the update for a new keyword meets a perfect binary tree. In this case, the client is required to get back *all* historical updates of the old root and re-encrypt them for the new root, which incurs the worst complexity ($\mathcal{O}(N)$ in Table 2) in all update-related criteria. Assume keywords are inserted in order, the scheme encounters the worst case for approximately $\mathcal{O}(\log |\mathcal{W}|)$ times before all keywords exist. It is hard to accept such a heavy overhead, especially for large databases.

Their scheme-A uses $\sum o\varphi o\varsigma$ [5] as the underlying DSSE, which requires trapdoor permutation (an asymmetric cryptographic primitive). To realize forward security, $\sum o\varphi o\varsigma$ requires the client to store a state for every keyword.

To save client storage and avoid the worst update overhead, scheme-B uses ciphertexts of HE [27] to store file identities for every node in the server side. However, we figure out that scheme-B is not forward-secure in contrast to what was claimed [34] and suffers from a variant of adaptive file injection attack [33]. There are also other drawbacks due to the usage of the asymmetric cryptographic primitive (refer to Section 4.1 for a discussion).

We also note that the *client computation overhead* for operating the tree-like index, before *searching* or *updating*, seems to be not seriously considered in both existing works [10,34]. Demertzis *et al.* [10] omitted details for this in their

instantiations. Zuo *et al.* [34] explained how to realize this part, which requires rebuilding the index whenever there is a search or update. Yet, this overhead is not included in their comparison table (Table 1 in [34]). The search/update efficiency may not be as good when it is implemented.

Order-preserving encryption (OPE) is another way to support range query over encrypted data. OPE preserves the order of plaintexts in the ciphertext domain, yet leaks information about the plaintexts [3]. Order-revealing encryption (ORE) [4] aims to reduce the leakage of OPE. Instead of comparing ciphertexts as is, ORE uses an algorithm to reveal the order of the selected ciphertexts. However, as observed by Kerschbaum and Tueno [20], ORE-based SSE schemes still suffer from plaintext-guessing attack [14]. They proposed a scheme which is free from the attack and only needs linear-space complexity in the server side (*cf.* replicated index [10,12,34]). It, however, requires multiple rounds of interactions.

Beyond the one-dimensional case, Wu *et al.* [32] propose ServeDB, which considers multi-dimensional range queries by 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 [2] to determine whether a query involves its descendants, which incurs false positive. Wu *et al.* explain how their scheme supports updates, yet their security analysis do not consider the update leakage, let alone forward or backward security. Their work additionally considers verifiability [26].

## 3 Preliminary

*Notation.* We denote the security parameter by $n$ and $\mathsf{negl}(n)$ is a negligible function in $n$. PPT stands for probabilistic polynomial-time. For a set $X$, $x \leftarrow_{\$} X$ samples an element $x$ uniformly from $X$. For an algorithm $A$, $x \leftarrow A$ means $x$ is an output of $A$. 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, $c_{\text{in}}$ (*resp.*, $s_{\text{in}}$) and $c_{\text{out}}$ (*resp.*, $s_{\text{out}}$) are inputs and outputs of the client (*resp.*, server). $||$ denotes string concatenation.

### 3.1 Dynamic Searchable Symmetric Encryption

We review the syntax and security definition of DSSE [5].

**Definition 1.** *A* DSSE *scheme is a tuple of polynomial-time algorithms/protocols* $(\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$.

$(K, \mathsf{EDB}, st) \leftarrow \mathsf{Setup}(1^n)$ *is a probabilistic algorithm that takes as input a security parameter* $1^n$*. It outputs a secret key* $K$*, an (initially empty) encrypted database* EDB*, and a state* $st$*.*

$((st', \mathcal{R}); (\mathsf{EDB}', \mathcal{R})) \leftarrow \mathsf{Search}(K, st, q; \mathsf{EDB})$ *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* $\mathsf{EDB}'$*. Both parties output a sequence of responses* $\mathcal{R}$*.*

$(st'; \mathsf{EDB}') \leftarrow \mathsf{Update}(K, st, op, up; \mathsf{EDB})$ *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 query* $up$ *parsed as a file identity* $id$ *and a set of keywords* $\{w\}_{id}$*) and the server (with an encrypted database* $\mathsf{EDB}$*). The client outputs a new state* $st'$ *and the server outputs an updated encrypted database* $\mathsf{EDB}'$*.*

*In a round-optimal scheme,* $\mathsf{Search}$ *(*resp.*,* $\mathsf{Update}$*) outputs a token* $t_q$ *(*resp.*,* $t_u$*) for the server to process locally over* $\mathsf{EDB}$ *without further help from the client.*

A $\mathsf{DSSE}$ scheme is correct if for all security parameters $1^n$, all $(K, \mathsf{EDB}, st) \in \mathsf{Setup}(1^n)$, and all sequences of $\mathsf{Search}$ and $\mathsf{Update}$ operations, $\mathsf{Search}(K, st, q; \mathsf{EDB})$ must returns the correct result.

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

**Definition 2.** *Let* $\mathsf{DSSE} = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a dynamic searchable symmetric encryption scheme. We say* $\mathsf{DSSE}$ *is* $\mathcal{L}$*-adaptively-secure, where* $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}\}$ *is a set of stateful leakage functions, if for any* $\mathsf{PPT}$ *adversary* $\mathcal{A}$*, there exists a* $\mathsf{PPT}$ *simulator* $\mathcal{S}$ *such that:*

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

*where* **Real** *and* **Ideal** *are probabilistic experiments defined below.*

$\mathbf{Real}_{\mathcal{A}}(1^n)$*: The challenger executes* $\mathsf{Setup}(1^n)$ *and sends (initially empty)* $\mathsf{EDB}$ *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* $\mathsf{Search}$ *or* $\mathsf{Update}$ *protocol on* $q$ *or* $(op, up)$ *respectively. Eventually,* $\mathcal{A}$ *returns a bit* $b$ *that is output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^n)$*:* $\mathcal{S}$ *generates (initially empty)* $\mathsf{EDB}$ *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* $\mathcal{L}^{\mathsf{Updt}}(op, up)$*. Eventually,* $\mathcal{A}$ *returns a bit* $b$ *that is output by the experiment.*

**Search Pattern and Update History.** Definition 2 captures the information leaked in $\mathsf{DSSE}$ with the leakage function set $\mathcal{L}$. Concretely, $\mathcal{L}$ 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 $(\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}})$ 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 taking all historical operations into consideration.

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

$$\mathsf{sp}(w) = \{u | (u, w) \in Q\} \text{ and } \mathsf{hist}(w) = \{(u, op, id) | (u, op, w, id) \in Q\}.$$

### 3.2 Best Range-Covering Technique

We review the *best range cover* (BRC) [10] used by Zuo *et al.* [34] and our constructions. We construct a full binary tree over its values bottom-up. Given a range (*i.e.*, a sequence of contiguous values) over a domain $\mathcal{D}$, BRC essentially selects the minimum set of nodes that cover exactly the range. For the example in Figure 1, $\mathcal{D} = \{0, \ldots, 7\}$. BRC of range $[2, 7]$ contains nodes $\tau_{2,3}$ and $\tau_{4,7}$ (shown as black nodes). Obviously, the number of nodes in BRC is always no more than the number of values in the range.

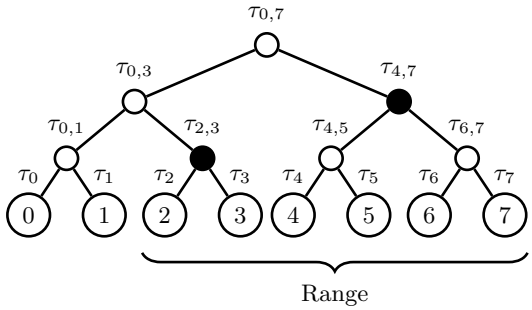


Fig. 1: Example for Best Range-Covering of $[2, 7]$

### 3.3 A General Framework of Range DSSE

Faber *et al.* [12] design a range SSE scheme by extending OXT [8] while Demertzis *et al.* [10] provide a generic framework of range DSSE with batched updates. Zuo *et al.* [34] propose two range DSSE schemes based on $\sum o\varphi o\varsigma$ [5] or homomorphic encryption. The main idea of these works is to reduce a range search to multiple single-keyword searches of an index-based SSE.

The range considered here is one-dimensional which is over a single attribute from a domain $\mathcal{W}$. We assume $\mathcal{W}$ is a set of contiguous positive integers. It is possible to convert a domain in real-world applications, *e.g.*, temperature records, to $\mathcal{W}$ with scaling and transformation. Each file owned by the client contains at least one keyword in $\mathcal{W}$. The goal is to build a secure index in the server side such that the server can answer range queries from the client that retrieve the identifiers of files containing any keyword in the range.

In this paper, we consider an extended version for range DSSE, which supports flexible updates instead of batched updates [10] and can be built on top of typical DSSE. Scheme-A of Zuo *et al.* [34] implicitly exploits this idea. We describe the general framework of range DSSE as follows. Note that the "keyword" in the following discussion is an artifact used for searching and updating over the underlying DSSE mechanism, which is different from real keywords in $\mathcal{W}$ that may appear in the queried range.

**Setup.**

1. Break $\mathcal{W}$ into a set of (potentially overlapping) ranges, and associate a unique "keyword" of the underlying $\mathsf{DSSE}$ to every range.
2. Utilize $\mathsf{DSSE.Setup}$ to generate an (initially empty) encrypted database.

**Search.**

1. Break the queried range into *sub-ranges*, and map them to "keyword".
2. Generate search tokens for these "keyword" with $\mathsf{DSSE.Search}$.

**Update.**

1. Parse an update query *up* as a file identity and a set of keywords in $\mathcal{W}$.
2. Associate the identity with the "keyword" whose corresponding range covers any keyword of the set, by executing $\mathsf{DSSE.Update}$ for these "keyword".

**Security.** The $\mathcal{L}$-adaptive security for range $\mathsf{DSSE}$ can be defined as that of $\mathsf{DSSE}$ in Definition 2. The security of range $\mathsf{DSSE}$ obtained via the above generic framework is highly related to that of the underlying $\mathsf{DSSE}$. Its leakage can be obtained by augmenting the leakage functions $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}\}$ of the underlying $\mathsf{DSSE}$ scheme to capture the extra leakage stemming from the keyword mapping[4] and index structure.

### 3.4 Forward and Backward Security for (Range) DSSE

Forward security requires that $\mathsf{Update}$ reveals nothing about which keywords are involved in the keyword-file pairs to be updated. We review the definition by Bost [5] for single-keyword $\mathsf{DSSE}$, which is widely used/extended in follow-up works. This definition is also applicable to range $\mathsf{DSSE}$.

**Definition 3 (Forward Security [5]).** *An $\mathcal{L}$-adaptively-secure (range) $\mathsf{DSSE}$ scheme is forward-secure if the update leakage function $\mathcal{L}^{\mathsf{Updt}}$ can be written as:*

$$\mathcal{L}^{\mathsf{Updt}}(op, up) = \mathcal{L}'(op, \{(id_i, \mu_i)\})$$

*where $op \in \{\mathsf{add}, \mathsf{del}\}$ is an operation, up is an update input parsed as a file identity and a set of keywords, $\{(id_i, \mu_i)\}$ is a set which captures all updates as the number of keywords $\mu_i$ modified in file $id_i$; $\mathcal{L}'$ is stateless, i.e., the output solely depends on the input.*

Backward security requires that, whenever a keyword-file pair $(w, id)$ has been added then deleted, searching over keyword $w$ reveals nothing about file $id$. Bost *et al.* [6] formalize backward security by introducing three leakage functions constructed from the operation list $Q$:

$$\mathsf{TimeDB}(w) = \{(u, id) \mid (u, \mathsf{add}, w, id) \in Q \ \wedge \ \forall u', (u', \mathsf{del}, w, id) \notin Q\},$$

[4] The leakage caused by different keyword-mapping strategies and its trade-off with the efficiency have been discussed throughly [10,12].

which represents the documents currently matching $w$ and when they are added;

$$\mathsf{UpdTime}(w) = \{u \mid (u, \mathsf{add}, w, id) \ \vee \ (u, \mathsf{del}, w, id) \in Q\},$$

which indicates the timestamp when all the updates on $w$ happened, and

$$\mathsf{DelHist}(w) = \{(u^{\mathsf{add}}, u^{\mathsf{del}}) \mid \exists id \ s.t. \ (u^{\mathsf{del}}, \mathsf{del}, w, id) \in Q \ \wedge \ (u^{\mathsf{add}}, \mathsf{add}, w, id) \in Q\},$$

which lists the timestamp-pairs of deletions and corresponding insertions on $w$.

With above leakage functions, we extend three levels of backward security [6] 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 3.3. Since (range) DSSE usually processes Update for every single keyword-file pair, we parse an update *up* as a keyword-file pair $(w, id)$ to be updated.

We omit the leakages that are confirmed from the initialization of the database and unrelated to forward/backward security, *e.g.*, the size of the keyword space. The definitions of backward security below start from the strongest one.

**Definition 4 (Backward Security).** *An $\mathcal{L}$-adaptively-secure range* DSSE *is*

- *insertion-pattern revealing backward-secure if*

$$\mathcal{L}^{\mathsf{Updt}}(op, w, id) = \mathcal{L}'(op),$$
$$\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}''((\mathsf{TimeDB}(k), a_k)_{k \in KSet_q})$$

  *where $a_k$ is the total number of updates on $k$.*
- *update-pattern revealing backward-secure if*

$$\mathcal{L}^{\mathsf{Updt}}(op, w, id) = \mathcal{L}'(op, w),$$
$$\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}''((\mathsf{TimeDB}(k), \mathsf{UpdTime}(k))_{k \in KSet_q}).$$

- *weakly backward-secure if*

$$\mathcal{L}^{\mathsf{Updt}}(op, w, id) = \mathcal{L}'(op, w),$$
$$\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}''((\mathsf{TimeDB}(k), \mathsf{DelHist}(k))_{k \in KSet_q}).$$

*where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless, i.e., their outputs solely depend on the inputs.*

It is oblivious that 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 identity $id$ associated with the deletion. Bost *et al.* [6] exclude this case when considering backward security for single-keyword DSSE.

For range DSSE, we cannot reuse this verbatim. In other words, 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 may trivially link file $f$ with the deleted file of 6 by observing the insertion and deletion time revealed by a subsequent query on 6. In this case, trivial extension cannot guarantee the privacy of the deleted file.

All three levels of backward security satisfy the intuitive requirement mainly by controlling the search leakage. As argued by Bost *et al.* [6], even a combination of forward security and weak backward security is enough for a DSSE scheme to limit the update leakage to the type of involved operations. Moreover, while a scheme which hides the update pattern can be obtained by ORAM, the usage of it would be rather expensive. Thus, weaker notions are considered for efficiency.

## 4 Injection Attack against (Range) DSSE

We first start with a critical review of a recently proposed range DSSE scheme [34] called scheme-B here. We then review the file injection attack against DSSE and illustrate how a variant of it can be executed over this scheme.

### 4.1 Review of Scheme-B [34]

Scheme-B of Zuo *et al.* [34] adopts the best range covering technique and relies on the homomorphism of HE, *e.g.*, Paillier encryption [27] in their instantiation to realize a range DSSE scheme with less client storage. Its communication complexity for Update is $\mathcal{O}(\log W)$, while their scheme-A [34] can be as worst as $\mathcal{O}(N)$ when a new root is created for the binary tree of existing keywords.

Concretely, keywords are represented as leaf nodes in the binary-tree-like index for range covering techniques. Every tree node corresponds to a bit-string of length being the maximum number of documents supported by the scheme. Whether the $i$-th document contains any keyword within the covering range of the node is indicated by the $i$-th bit of the string (1 for positive and 0 for negative). The server stores an HE encryption of the bit-string for every node.

With this structure, the maximum document number is limited to the maximum length of plaintext supported by underlying *public-key* HE, which accounts for the "*small*" number of documents in Table 1. In contrast, for most other DSSE schemes, even if some kinds of limitations on the number of documents exist, it is related to *symmetric-key* primitives and enlarging it has less implication on the efficiency of the whole system. Also, for every existing tree node, the server has to store a bit-string of length being the maximum document number. Many of these bits are "empty slots" before corresponding documents are uploaded.

To issue a range query, the client finds the minimal set of nodes covering the range, *i.e.*, BRC, and generates the search tokens for these nodes. With search tokens, the server accesses the locations for the encrypted bitstrings and returns these ciphertexts, which can be decrypted by the client for the matching file identities of the queried range. This accounts for $\mathcal{O}(w_q)$ search communication overhead in Table 2, which are actually *ciphertexts of Paillier encryption* [27].

It also requires the client to *perform decryption* and incurs *one more round* of communication to get the matching files.

To update a file containing a specific keyword, the client finds the path from the leaf node of this keyword to the root of the tree. The client generates an encryption of a bitstring to be homomorphically added to the ciphertext of the bitstring at each node along the path. The expected result is to *flip* every bit corresponding to this file along the path. This procedure by itself requires only some basic information of the index, which accounts for its $\mathcal{O}(1)$ client storage. However, the client is not supposed to know the original bit value to be updated in the *outsourced* database, and thus cannot make sure whether an update can really work as an insertion or a deletion, unless taking an additional round of Search before Update. (Applying an insertion update over an existing record, *cf.*, clicking "Save" twice in editing a document, now results in deletion!)

### 4.2 File Injection Attack

The efficient file-injection attack by Zhang *et al.* [33] shows the importance of forward security for DSSE. In their setting, the server sends files of its choice to the client who then encrypts and uploads them as normal DSSE schemes. Suppose that the keyword space is $\mathcal{W}$ and the target dataset supports $|\mathcal{W}|$ distinct keywords at most. The server could inject $\lceil \log |\mathcal{W}| \rceil$ files, each of which contains exactly half of the keywords from $\mathcal{W}$. By observing the set of injected files matching a search token, the adversary could tell which keyword is contained in the token. This is a non-adaptive attack and can be mitigated by a threshold $T$ which limits the number of keywords in a single file.

An adaptive attack is thus proposed against the *threshold-based countermeasure*, where the keyword space is divided into $\lceil |\mathcal{W}|/T \rceil$ subsets. The server first injects $(\lceil |\mathcal{W}|/T \rceil - 1)$ files, each of which contains $T$ 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. The result of Zhang *et al.* [33] showed that, after the target subset is known, only 8 new files have been injected when $T = 200$. However, this attack requires the leakage of the relation between newly injected files and the keyword in the target search token, which is exactly hidden by forward security. In other words, this adaptive attack, together with some other similar attacks, will not work for a forward-secure DSSE scheme.

### 4.3 Attack on Scheme-B [34]

We have the following observations for scheme-B of [34]. 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 3. The construction is thus not forward-secure

as claimed[5]. We then concretely show how an adversary with the power of file injection could exploit this information to efficiently determine the queried range.

Consider the threshold setting [33] reviewed in Section 4.2, 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, which are 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 the 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 target 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, the range for the previous search can be figured out.

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

## 5 Generic Forward-Secure Range DSSE

Our generic construction of forward-secure range DSSE is based on a binary-tree-like index and any forward-secure DSSE $\Pi$. Figure 2 describes its details. Its security analysis can be found in Appendix A.

In our construction, *CSet* is the covering node set which helps the client to record elements in the BRC of a given range, and *RSet* is for the server to record the search result during the search operation.

Without loss of generality, we assume the size of keyword space $|\mathcal{W}| = 2^m$. We represent the $(k+1)$-bit binary form of an integer $a$ as $\mathsf{a}_0 \cdots \mathsf{a}_k := [a]_{\mathsf{bin}}$ (prepending 0s if needed) and set $\mathsf{a}_0 =$ " ".

Our construction uses the binary representation of the keyword value to implicitly maintain a binary tree with depth logarithmic in the size of keyword space. If $\mathsf{w}_1 \cdots \mathsf{w}_m$ is the binary form of a keyword $w$, we represent the ancestors for the leaf node corresponding to $w$ 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 the ancestors, $\mathsf{w}_0$ always represents the root with representation " ".

[5] The update leakage $\mathcal{L}^{\mathsf{Updt}}$ of scheme-B is claimed [34] to be the number of updates made to the keyword $w$ and when the update happened. As we show, it leaks more.

$\mathsf{Setup}(1^n)$

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

$\mathsf{Update}(K, st, op, up; \mathsf{EDB})$

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 : $\quad up_i := (\mathsf{w}_0 \cdots \mathsf{w}_i, id)$
5 : $\quad (st'; \mathsf{EDB}') \leftarrow$
$\quad\quad \Pi.\mathsf{Update}(K, st, op, up_i; \mathsf{EDB})$
6 : $\quad st := st', \mathsf{EDB} := \mathsf{EDB}'$
7 : **endfor**
8 : **return** $(st'; \mathsf{EDB}')$

$\mathsf{Search}(K, st, q; \mathsf{EDB})$

1 : **Parse** $q$ **as** $[a, b]$
2 : $CSet, RSet \leftarrow \emptyset, i := 0$
3 : $\mathsf{a}_0 \cdots \mathsf{a}_m := [a]_{\mathsf{bin}}, \mathsf{b}_0 \cdots \mathsf{b}_m := [b]_{\mathsf{bin}}$
4 : **while** $\mathsf{a}_0 \cdots \mathsf{a}_{m-i} < \mathsf{b}_0 \cdots \mathsf{b}_{m-i}$ **do**
5 : $\quad$ **if** $\mathsf{a}_{m-i} = 1$ **then**
6 : $\quad\quad CSet := CSet \cup \{\mathsf{a}_0 \cdots \mathsf{a}_{m-i}\}$
7 : $\quad$ **if** $\mathsf{b}_{m-i} = 0$ **then**
8 : $\quad\quad CSet := CSet \cup \{\mathsf{b}_0 \cdots \mathsf{b}_{m-i}\}$
9 : $\quad \mathsf{a}_0 \cdots \mathsf{a}_{m-i} := \mathsf{a}_0 \cdots \mathsf{a}_{m-i} + [1]_{\mathsf{bin}}$
10 : $\quad \mathsf{b}_0 \cdots \mathsf{b}_{m-i} := \mathsf{b}_0 \cdots \mathsf{b}_{m-i} - [1]_{\mathsf{bin}}$
11 : $\quad i := i + 1$
12 : **endwhile**
13 : **if** $\mathsf{a}_0 \cdots \mathsf{a}_{m-i} = \mathsf{b}_0 \cdots \mathsf{b}_{m-i}$ **then**
14 : $\quad CSet := CSet \cup \{\mathsf{a}_0 \cdots \mathsf{a}_{m-i}\}$
15 : **for** $\tau \in CSet$ **do**
16 : $\quad ((st', \mathcal{R}); (\mathsf{EDB}', \mathcal{R}))$
$\quad\quad\quad \leftarrow \Pi.\mathsf{Search}(K, st, \tau; \mathsf{EDB})$
17 : $\quad RSet := RSet \cup \mathcal{R}$
18 : $\quad st := st', \mathsf{EDB} := \mathsf{EDB}'$
19 : **endfor**
20 : **return** $((st', RSet); (\mathsf{EDB}', RSet))$

Fig. 2: Our Generic Construction of Forward-Secure Range DSSE

Given a range query, the client first figures out the BRC of the range and stores them in the covering node set *CSet* (line 1–14 of Search). For every element in *CSet*, the client performs search operations of the underlying DSSE (line 15–19). The server will return the union set of all results.

### 5.1 The Best Range Covering (BRC) Technique

We realize the best range covering technique with an implicit tree-like index. It can serve as a reference for instantiating related structures assumed by the literature [34,10]. The pseudocode is in line 1–14 of Search protocol in Figure 2.

Suppose we want to figure out the BRC for a range of contiguous positive integers within a domain. We consider a binary tree, whose leaf nodes from

left to right representing values from 0 to the largest value in the domain. We represent the value of every leaf node as a binary string with the length being the depth of the tree. The binary string for any parent is that for its child with its last bit truncated. In addition, we denote the root node as a null string " ". For example, the binary form of node $\tau_4$ in Figure 1 is " 100" and its parent ($\tau_{4,5}$) can be represented as " 10". With the knowledge of the domain, we can follow the representation of tree nodes without explicitly building the tree.

For any given range, we transform its upper bound and lower bound to their binary form first (line 3 of Search). 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 the BRC (line 5–8). This case implicitly corresponds to the case that 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 9–10).

Then, we truncate the last bit of the binary form of both bounds by moving a cursor (line 11) and set the truncated binary strings as the new bounds. We repeat the above operations until the lower bound is not smaller than the upper bound. If eventually the lower bound equals the upper bound, we also insert the value of the bound into the BRC (line 13–14).

Consider BRC of [2, 7] in Figure 1, we transform it into " 010" and " 111".

- The last bit of the lower bound is not 1, so we increase it by 1 (*i.e.*, " 011") and get the new lower bound by truncating the last bit of the resulting binary string (*i.e.*, " 01").
- The last bit of the upper bound is not 0, so we decrease it by 1 (*i.e.*, " 110") and get the new upper bound by truncating the last bit of the resulting string (*i.e.*, " 11").
- The last bit of the new lower bound is 1, so we insert " 01" into the BRC and update the lower bound as above (*i.e.*, " 1").
- The last bit of the new upper bound is not 0, so we update the upper bound as above (*i.e.*, " 1").
- Two bounds are now equal, we insert their value " 1" into the BRC and end the loop.

The BRC of range $[2, 7]$ is {" 01", " 1"} corresponding to $\tau_{2,3}$ and $\tau_{4,7}$ in Figure 1.

### 5.2 Update Protocol

When adding/deleting the index with a keyword-file pair, 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.* [34]) since the ancestors' locations can be found with the knowledge of the (binary form of) keyword value. The client just needs to

record the height of the implicit tree (which is also the bit length of the keyword value) to process this binary-tree-like index correctly for the next Search/Update operation.

### 5.3 Efficiency Analysis

For a clear comparison with previous solutions, we give the asymptotic complexity of our generic construction instantiated with a scheme called FASTIO [28]. We defer experimental evaluations to Section 8.

In the client side, the computational overhead for Search is $\mathcal{O}(w_q)$, where $w_q$ is the number of keywords within a range query $q$, as the number of the covering nodes is at most $w_q$. The computational overhead for Update is $\mathcal{O}(\log|\mathcal{W}|)$ as the updates need to be done for every ancestor of the leaf node corresponding to the keyword in the binary tree. Assume that keywords are added in order, our client stores approximately $(2W + \log|\mathcal{W}| - \log W)$ tuples, which are essentially the number of existing keywords and the non-leaf nodes on their way to the root.

In the server side, the computational overhead for Search is $\mathcal{O}(\overline{n_q})$, where $\overline{n_q}$ is the number of updates that contain the keywords in the queried range since the last search of them. For Update, the computational overhead is $\mathcal{O}(\log|\mathcal{W}|)$.

The communication for Search is $\mathcal{O}(|\mathtt{DB}(q)|)$, where $|\mathtt{DB}(q)|$ is the number of files matching query $q$. The communication for Update is $\mathcal{O}(\log|\mathcal{W}|)$, which is related to the number of the update tokens.

**Comparison with the Existing Forward-Secure Range DSSE.** Our construction avoids the index reconstruction during Search and Update of scheme-A [34], which incurs overhead linear in the maximum sequence number of existing keywords $W_{\mathrm{x}}$. For the client search computation, ours is definitely better. For the client update computation, ours is better than scheme-A if $W_{\mathrm{x}}$ is larger than $\log|\mathcal{W}|$. Consider the size of keyword space $|\mathcal{W}| = 2^{20}$. We perform better under this criterion if $W_{\mathrm{x}} > 20$, which is likely to happen in practice for a database containing more than 20 distinct keywords. Our client storage is more than that of scheme-A (Table 3). Yet, when the number of distinct keywords increases, the gap narrows down. Eventually, two constructions realize the same overhead, which is $\mathcal{O}(|\mathcal{W}|)$.

**Advantages of Our Generic Construction.** Our generic construction naturally benefits from the development of DSSE. For example, when instantiated with FASTIO [28], our construction inherits the advantages of physical deletion and caching the previous search results. The search complexity of the server is $\mathcal{O}(\overline{n_q})$, where $\overline{n_q}$ is the total number of updates that contain the keywords in the queried range since the last search of them. In contrast, scheme-A [34] requires $\mathcal{O}(n_q)$, where $n_q$ is the total number of updates (add + del) that contain the keywords in the queried range since initialization.

$\mathsf{Setup}(1^n)$

$\mathbf{W}, \mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c} \leftarrow \emptyset$
$K_1, K_2 \leftarrow_\$ \{0,1\}^\lambda$
**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:**
**Parse** $up$ **as** $(w, id)$
$SSet, t_\mathsf{u} \leftarrow \emptyset, cnt := 0$
$\mathsf{w}_0 \cdots \mathsf{w}_m := [w]_\mathsf{bin}$
$(k_\tau, c^\mathsf{h}, c^\mathsf{u}) \leftarrow \mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_m]$
**if** $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) = \perp$ **then**
  $k_\tau := F(K_1, \mathsf{w}_0 \cdots \mathsf{w}_m)$
  $c^\mathsf{h} := c^\mathsf{u} := 0$
**endif**
$SSet := SSet \cup \{(k_\tau, c^\mathsf{u})\}, cnt := c^\mathsf{h}$
$\mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_m] := (k_\tau, c^\mathsf{h} + 1, c^\mathsf{u} + 1)$
**for** $i := m - 1$ **to** $0$ **do**
  $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) \leftarrow \mathbf{W}[\mathsf{w}_0 \cdots \mathsf{w}_i]$
  **if** $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) = \perp$ **then**
    $cnt := cnt +$ $\mathsf{histUptCnt}(\mathbf{W}, m, \mathsf{w}_0 \cdots \mathsf{w}_i \bar{\mathsf{w}}_{i+1})$
    $k_\tau := F(K_1, \mathsf{w}_0 \cdots \mathsf{w}_i), c^\mathsf{h} := cnt$
    $SSet := SSet \cup \{(k_\tau, c^\mathsf{h})\}$
  **else**
    $SSet := SSet \cup \{(k_\tau, c^\mathsf{u})\}, cnt := c^\mathsf{h}$
  **endif**
**endfor**
**for** $(k_\tau, c) \in SSet$ **do**
  $addr := H_1(k_\tau || (c+1))$
  $val := (id || op) \oplus H_2(k_\tau || (c+1))$
  $t_\mathsf{u} := t_\mathsf{u} \cup \{(addr, val)\}$
**endfor**
**send** $t_\mathsf{u}$ **to server**
**Server:**
**for** $(addr, val) \in t_\mathsf{u}$ **do**
  $\mathbf{T}_\mathsf{e}[addr] := val$
**endfor**

$\mathsf{Search}((K_1, K_2), \mathbf{W}, [a, b]; \mathbf{T}_\mathsf{e}, \mathbf{T}_\mathsf{c})$

**Client:**
**Generate** $CSet$ **as** Search **in Figure** 2
$t_\mathsf{q} \leftarrow \emptyset$
**for** $\tau \in CSet$ **do**
  $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) \leftarrow \mathbf{W}[\tau]$
  **if** $(k_\tau, c^\mathsf{h}, c^\mathsf{u}) = \perp$ **then**
    $c^\mathsf{h} := \mathsf{histUptCnt}(\mathbf{W}, m, \tau)$
    **if** $c^\mathsf{h} = 0$ **then**
      **continue**
    $k_\tau := F(K_1, \tau), c := c^\mathsf{u} := c^\mathsf{h}$
  **endif**
  **if** $c^\mathsf{u} \neq 0$ **then**
    $k_w := k_\tau, k_\tau \leftarrow_\$ \{0,1\}^\lambda, c := c^\mathsf{u}$
  **else**
    $k_w := \perp, c := 0$
  $\mathbf{W}[\tau] := (k_\tau, c^\mathsf{h}, 0)$
  $nym_w := F(K_2, \tau)$
  $t_\mathsf{q} := t_\mathsf{q} \cup \{(nym_w, k_w, c)\}$
**endfor**
**send** $t_\mathsf{q}$ **to server**
**Server:**
$\mathcal{R} \leftarrow \emptyset$
**for** $(nym_w, k_w, c) \in t_\mathsf{q}$ **do**
  $RSet := \mathbf{T}_\mathsf{c}[nym_w]$
  **if** $k_w \neq \perp$ **then**
    **for** $i := 1$ **to** $c$ **do**
      $addr := H_1(k_w || i)$
      $(id, op) := \mathbf{T}_\mathsf{e}[addr] \oplus H_2(k_w || i)$
      **if** $op = \mathsf{del}$ **then**
        $RSet := RSet \setminus \{id\}$
      **else**
        $RSet := RSet \cup \{id\}$
      **delete** $\mathbf{T}_\mathsf{e}[addr]$
    **endfor**
  **endif**
  $\mathbf{T}_\mathsf{c}[nym_w] := RSet$
  $\mathcal{R} := \mathcal{R} \cup RSet$
**endfor**
**send** $\mathcal{R}$ **to client**

Fig. 3: Our Less-client-storage Construction of Forward-Secure Range DSSE

# 6 Range DSSE with Less Client Storage

## 6.1 Motivation

To our knowledge, the best client storage for round-optimal single-keyword *forward-secure* DSSE (*e.g.*, [5,24]) is linear in the number of distinct keywords. From the discussion in Section 2 and Table 3, we observe that existing forward-secure range DSSE constructions require the client storage to be at least twice as much as that of single-keyword forward-secure DSSE, even if no range query has been issued. Scheme-B of Zuo *et al.* [34] attempts to solve this problem, but it is not forward-secure (Section 4.3).

In practice, this extra client-side overhead for supporting range query may dampen users' enthusiasm. On one hand, for a very large database with billions of distinct keywords, even a constant-number increase over complexity incurs a large cost for the client-side disk space. On the other, 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 unreasonable.

We thus propose a new scheme with less client storage, which provides the same client storage as single-keyword forward-secure DSSE if no range query happens. The client storage will increase when any range is queried for the first time. Only when all possible ranges have been queried will the client storage grow as previous range DSSE.

## 6.2 Description of Our Scheme

*Overview.* Observe that for every keyword existing in a forward-secure DSSE, the client needs to store a tuple to record related information (*e.g.*, a key and a counter). When a binary-tree-like structure is used for range query, current constructions (*e.g.*, scheme-A [34] and our generic range DSSE in Section 5) additionally store the same type of tuple for every tree node, which makes the client storage at least twice in the number of existing keywords in total.

For the binary-tree-like structure, storing such tuples for the leaf nodes is hard to avoid as they are the "basis" required by existing forward-secure DSSE schemes. Yet, it is possible that some information about the non-leaf nodes (*e.g.*, the update number) can be computed from the tuples of related leaf nodes. We also note that the key is useless before any search over the node happens. Thus, we record the key only when necessary (concretely, after the first search happens over the node corresponding to the key in our scheme). This is how our scheme saves client storage.

For forward security, we adopt a generic trick [24] (also similar to FAST-IO [28]) which uses a new random key after every search to avoid the linkage.

*Formal Description.* Figure 3 presents the detailed construction of our forward-secure range DSSE with less client storage in which utilizes the implicit binary-tree-like index. Without loss of generality, we assume the size of keyword space is $|\mathcal{W}| = 2^m$.

Besides the covering node set *CSet* and the result set *RSet* as the generic construction, we use a state set *SSet* to record *the state of elements to be updated*. $t_{\mathsf{q}}$ and $t_{\mathsf{u}}$ are *collections of the search tokens and the update tokens* respectively.

**Setup.** $H_1$ and $H_2$ are cryptographic hash functions with an appropriate domain and output length. The client outputs two $n$-bit *pseudorandom function* (PRF) keys $K_1$ and $K_2$ for $F$, 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 in his/her side, 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 $k_\tau$ that is either an output of PRF when the node $\tau$ is added for the first time or a random string after each search over the node,
  - a history counter $c^{\mathsf{h}}$ to record the number of historical updates over the node since the initialization, and
  - an update counter $c^{\mathsf{u}}$ to indicate the number of updates over the node after last search.

Before describing $\mathsf{Search}$, we define $\mathsf{histUptCnt}$, a function for the client to compute $c^{\mathsf{h}}_\tau$ of non-leaf node $\tau$ (without storing it) from handy information stored at its descendant nodes. This count is required in $\mathsf{Search}$ and $\mathsf{Update}$.

$c^{\mathsf{h}}_\tau \leftarrow \mathsf{histUptCnt}(\mathbf{W}, m, \tau)$ is a deterministic algorithm that takes the map $\mathbf{W}$, the depth of the implicit binary tree $m$, and a node $\tau$ in the implicit binary tree. It outputs $c^{\mathsf{h}}_\tau$, which is essentially the sum of $c^{\mathsf{h}}$ over the existing leaf descendants of $\tau$. Section 6.3 will discuss its implementation.

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

1. (Line 1–2) The client figures out the covering node set *CSet*, *i.e.*, the BRC of the queried range (Section 5.1), and resets the update token collection $t_{\mathsf{q}}$.
2. For each $\tau$ in *CSet*:
   (Line 3–10) If $\mathbf{W}[\tau]$ does not exist, *i.e.*, node $\tau$ has not been retrieved before, the client uses $\mathsf{histUptCnt}$ to get the historical updates $c^{\mathsf{h}}$ of $\tau$. If $c^{\mathsf{h}} \neq 0$, node $\tau$ will be searched for the first time. The client sets $k_\tau$ as $F(K_1, \tau)$ and $c^{\mathsf{u}}$ as $c^{\mathsf{h}}$ for $\tau$.
   (Line 11–20) The client generates $(nym_w, k_w, c)$:
   - $nym_w$ is a pseudonym for locating previous records related to $\tau$ from $\mathbf{T}_{\mathsf{c}}$,
   - $k_w$ is set as $k_\tau$ from $\mathbf{W}[\tau]$ if $c^{\mathsf{u}} \neq 0$, and
   - $c$ is a counter that indicates how many updates have been performed on $\tau$ after the last search for it or since the initialization if not searched.

   For every node to be retrieved, the client assigns a random key to $k_\tau$, records history counter $c^{\mathsf{h}}$, and resets update counter $c^{\mathsf{u}}$. All search tokens $\{(nym_w, k_w, c)\}$ are collected in $t_{\mathsf{q}}$ and sent to the server.

3. (Line 21) For every search token in $t_{\mathsf{q}}$:
   (Line 22) The server accesses the last search results in $\mathbf{T}_{\mathsf{c}}$ with $nym_w$, and puts them into the result set $RSet$.
   (Line 23–34) 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 the operation $op = \mathsf{del}$, remove the corresponding file identity $id$ from $RSet$; otherwise, insert $id$ into $RSet$. The accessed storage for current search in $\mathbf{T}_{\mathsf{e}}$ can be physically deleted afterward. The search results are archived into $\mathbf{T}_{\mathsf{c}}$.
4. (Line 35–37) The server outputs a response $\mathcal{R}$, a union set of the result sets for all search tokens in $t_{\mathsf{q}}$.

**Update.** Updateworks similarly as our generic construction since both use the same tree-like index, but with special housekeeping for saving client storage.

The server input is $\mathbf{T}_{\mathsf{e}}$. For an update tuple $(op, up = (w, id))$, The client takes $(K_1, \mathbf{W})$ as input, sends to the server the update token collection $t_{\mathsf{u}}$, which contains update tokens for nodes corresponding to $w$ and all of its ancestors.

1. (Line 3–10) For keyword $w$ (*i.e.*, leaf node), if $\mathbf{W}[[w]_{\mathsf{bin}}]$ does not exist, the client sets $k_\tau$ as $F(K_1, \tau)$ and initializes $c^{\mathsf{h}}$ and $c^{\mathsf{u}}$ for $w$. The tuple $(k_\tau, c^{\mathsf{u}})$ from $\mathbf{W}[[w]_{\mathsf{bin}}]$ is put to $SSet$. $c^{\mathsf{u}}$ and $c^{\mathsf{h}}$ 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 $\mathbf{W}$. If $(k_\tau, c^{\mathsf{h}}, c^{\mathsf{u}}) \neq \perp$, *i.e.*, the node has been retrieved, the client inserts $(k_\tau, c^{\mathsf{u}})$ into $SSet$; otherwise, the client gets the number of historical updates $c^{\mathsf{h}}$ with $\mathsf{histUptCnt}$, assigns $k_\tau$ to be the PRF output , and inserts $(k_\tau, c^{\mathsf{h}})$ into $SSet$. Note that the client is not required to store these values currently. To avoid repetitive access for $\mathbf{W}$ 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 $(k_\tau, c)$ in $SSet$, the client generates update token $(addr, val)$, where $addr$ is a location based on the hash of $k_\tau$ and the incremental counter $c$, and $val$ is the concatenation of $id$ and $op \in \{\mathsf{add}, \mathsf{del}\}$ encrypted by another hash of $k_\tau$ and $c$. sent as $t_{\mathsf{u}}$.
4. (Line 27–29) For each $(addr, val)$ in $t_{\mathsf{u}}$, the server stores $val$ for addition/deletion of a keyword-file pair at the location $addr$ of the encrypted index $\mathbf{T}_{\mathsf{e}}$.

### 6.3 Efficiency Analysis

In the client side, the storage overhead is $\mathcal{O}(W + s)$ where $s$ is *the number of distinct overlapping elements in all queries*, *i.e.*, the size of the union set of all *CSet*s (BRCs of range queries) since initialization. The value of $s$ only increases when a query accesses any implicit node that has never been retrieved. Consider the example in 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 any range query beyond $[2, 7]$, $[2, 3]$, and $[4, 7]$ is issued.

In this construction, 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 which are hardly found

in other range DSSE constructions. Our construction 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 that of existing range DSSE constructions before every possible non-overlapping range has been queried, *i.e.*, all nodes in the index have been retrieved.

The client overhead for Update is $\mathcal{O}(W+\log|\mathcal{W}|)$. Compared with our generic construction, 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. Yet, when the number of queried ranges increases, we can implement histUptCnt with an $\mathcal{O}(1)$ overhead. The idea is to exploit the information for the adjacent nodes of the target exists in the client storage since the number of historical updates over a non-leaf node is exactly the sum of historical updates of its children. 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 search complexity for the client is still $\mathcal{O}(w_q)$, as either the size of *CSet* or the number of accessed nodes by histUptCnt is no more than $w_q$.

In the server side, the computational overheads for search and update are the same as our generic construction instantiated with FASTIO [28]. In particular, the server does not carry out any operations over the part of search results which was archived into $\mathbf{T_c}$ beyond putting it in the result set. Our less-client-storage construction keeps the optimal search communication overhead (*i.e.*, $\mathcal{O}(|\mathtt{DB}(q)|)$).

### 6.4 Discussion on Reconstruction Attack

The proof of our scheme is given in Appendix B. Here, we discuss reconstruction attacks [19,13] which reconstruct the database by observing the (volumes of) search results of an enough number of search queries.

Range DSSE is less likely to suffer from this attack than a static range SSE since the addition/deletion of records will change the volume which has already been observed by the adversary[6]. Some typical attacks have the assumption that the database is static. One may argue that the volume of the response given by range DSSE may also stay unchanged, say, between two updates or when there is no subsequent update. Even so, the state-of-the-art reconstruction attack [13] requires the observation of $\mathcal{O}(|\mathcal{W}|^2\log|\mathcal{W}|)$ uniformly distributed range queries. Consider our setting in Section 8 where $|\mathcal{W}|=2^{20}$. The adversary needs to see around $20\cdot 2^{40}$ uniformly distributed range queries to launch the attack. It may be too many for an attack in practice.

Moreover, the client can choose to rebuild the encrypted database at the proper time to reduce the risk. The client storage in our second scheme, which is

[6] Update recovery attack of Grubbs *et al.* [13] assumes that the adversary has already executed either the reconstruction attack or a one-time compromise of the database, which is out of the scope here.

directly related to the number of distinct range queries, can serve as an indicator for this proper time without incurring additional storage.

There are also attacks [22] which fully negate encryption or at least reconstruct the original data within a constant ratio of error. Such attacks require the target dataset to be dense or with an auxiliary distribution available to the adversary. For example, age data cannot be secured by current encrypted search approaches. In this case, one may consider additional protection strategy [25].

## 7 Backward Security for Range DSSE

### 7.1 A Quick Review of a Generic Upgrade

We upgrade both our range DSSE constructions with the two-roundtrip transformation proposed by Bost *et al.* [6]. The transformation provides at least update-pattern-revealing backward security, originally for single-keyword DSSE, by using an additional PRF secret key to derive $K_w$ for keyword $w$. To carry out update operation $op$ for the keyword-file pair $(w, id)$, the transformation lets the client add a pair of $(w, \mathsf{Enc}_{K_w}(id, op))$ instead. Thus, the server only sees ciphertexts (*e.g.*, $\mathsf{Enc}_{K_w}(id, \mathsf{add}), \mathsf{Enc}_{K_w}(id, \mathsf{del})$) even after Search. The client then needs to decrypt all retrieved results and remove the deleted file identities locally to get the actual search results. It thus takes an additional round for the client to get the matching files and re-encrypt the ciphertexts of non-deleted identities.

### 7.2 Backward-Secure Range DSSE

Our generic construction of range DSSE is based on any DSSE. Thus, we apply this generic upgrade on the underlying DSSE of our generic construction. We omit the largely repetitive description of the resulting scheme but highlight the important aspects. For Update, since our generic construction adopts an implicit binary-tree-like index, the client needs to operate along the path from the leaf, corresponding to the updated keyword, to the root of the tree. Following the transformation, any update over any node of this path are now encrypted under a different key (*e.g.*, [3], [2, 3]) which the server never gets to see. Thus, the file identities, inserted for nodes in the covering node set *CSet* of the query, will not be exposed before the client removes the deleted ones. After each search query, we follow the generic upgrade and require the matching file identities to be re-encrypted and uploaded to the server. According to our Definition 4, the transformed scheme satisfies update-pattern-revealing backward-security[7].

For our less-client-storage scheme, recall that the server keeps the archives for the searched keyword ($\mathbf{T}_\mathsf{c}$), one may concern whether the adversary can violate backward security, say, by analyzing the searched results in different stages and gaining some information of deleted files. However, the archives are only refreshed when a new search query is processed, that is, any potential leakage involved

[7] Bost *et al.* [6] claimed that for a certain type of DSSE, the transformed scheme can be insertion-pattern-revealing backward-secure.

with the archive can only manifest itself after a new search. As discussed in Section 3.4, the case that a search query happens between the insertion and the corresponding deletion is excluded from the consideration of backward security.

Concretely, for $\mathsf{Update}$ over a keyword, the transformation over our less-client-storage construction requires the client inserts the ciphertext of the concatenation of the operation and the updated document. This will leak the size of *SSet* (*i.e.*, the size of keyword space), yet it is not relevant to backward privacy. For $\mathsf{Search}$ of a range $q$, the client decrypts the retrieved ciphertexts corresponding to the best range covering set of $q$. Then, the client removes the deleted file identities since the last search of it, which guarantees backward privacy. The server is informed of the identities, 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 less-client-storage property.

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

- $\mathcal{L}^{\mathsf{Stp}} = \perp$.
- $\mathcal{L}^{\mathsf{Srch}}(q) = ((\mathsf{sp}(\tau)^8, \mathsf{DB}(\tau), \mathsf{UpdTime}(\tau))_{\tau \in \text{BRC of } q})$.
- $\mathcal{L}^{\mathsf{Updt}}(op, up) = |\mathcal{W}|$.

## 8 Experimental Evaluation

We implement our generic range $\mathsf{DSSE}$ (denoted by $\mathsf{RSSE\text{-}BRC}$) and range DSSE with less client storage (denoted by $\mathsf{RSSE\text{-}LC}$) in C++11. We instantiate the $\mathsf{DSSE}$ of $\mathsf{RSSE\text{-}BRC}$ with an adaptation of FASTIO, a recent forward-secure single-keyword $\mathsf{DSSE}$ scheme [28]. For cryptographic components, 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[9]. In our experiment, we consider the size of keyword space $|\mathcal{W}| = 2^{20}$, *i.e.*, the keyword domain is $\{0, \ldots, 1048575\}$. We deploy the server with a single Intel Core i7-4790 3.60GHz CPU and 16GB of RAM, and the client machine with a single Intel Core i5-6500 3.20GHz CPU and 8GB of RAM in a LAN setting.

### 8.1 Update Evaluation

Table 4 evaluates the update efficiency by measuring the time needed to update keyword-file pairs. Concretely, we measure the time for performing $10^3$–$10^6$ update operations (with $9 \times 10^2$–$9 \times 10^5$ distinct keywords respectively) and calculate the average time for updating a single keyword-file pair in $\mathsf{RSSE\text{-}BRC}$ and $\mathsf{RSSE\text{-}LC}$ respectively. Both constructions only require symmetric cryptographic primitives, which explains the high performance.

[8] Some backward-secure (range) $\mathsf{DSSE}$ also leak the search pattern $\mathsf{sp}(w)$, *e.g.*, Janus [6], Janus++[30], and Scheme-B [34].

[9] RocksDB: https://rocksdb.org; gRPC: https://grpc.io

Table 4: Update Efficiency of RSSE-BRC and RSSE-LC

| RSSE Scheme | Time for updating $n$ pairs (s) | | | | Avg. single update time |
|---|---|---|---|---|---|
| | $n = 10^3$ | $n = 10^4$ | $n = 10^5$ | $n = 10^6$ | |
| BRC | 1.43 | 14.15 | 141.41 | 1483.32 | $1.47 \times 10^{-3}$ |
| LC | 3.38 | 22.80 | 223.42 | 2225.14 | $2.23 \times 10^{-3}$ |

The update operation in RSSE-LC is slower than RSSE-BRC, which is what RSSE-LC trades for less client storage.

### 8.2 Search Evaluation

We calculate how many search tokens our schemes can reduce compared with single-keyword DSSE. The number of search tokens is exactly the size of BRC for the queried range in RSSE-BRC and RSSE-LC, while in single-keyword DSSE it is the size of the queried range. For comparison, we issue multiple queries with ranges of $5 \times 10^2, 5 \times 10^3$, and $5 \times 10^4$ uniformly distributed over the keyword domain, and measure the *average number* of search tokens. As shown in Table 5, the gap could be large especially when the query covers a wide range. Note that the network latency is a common bottleneck for DSSE in practice, our schemes save the communication overhead and the time needed for Search.

Table 5: Average Number of Search Tokens

| Queried Range | $5 \times 10^2$ | $5 \times 10^3$ | $5 \times 10^4$ |
|---|---|---|---|
| Single-keyword DSSE | $5 \times 10^2$ | $5 \times 10^3$ | $5 \times 10^4$ |
| Our constructions | 8.17 | 9.67 | 13.16 |

We also measure the search time over a database containing $5 \times 10^7$ keyword-file pairs with $5 \times 10^5$ distinct keywords. We start counting when the client starts generating the search token until the client receives the search results. Multiple queries with ranges of $5 \times 10^2, 5 \times 10^3$, and $5 \times 10^4$ are uniformly distributed over the keyword domain. Table 6 shows the performance. The search operation in RSSE-BRC is slightly faster than that of RSSE-LC since RSSE-LC exploits histUptCnt to get the update counters when generating the search tokens. For either of constructions, the search complexity remains competitive in practice.

### 8.3 Storage Evaluation

To evaluate the client storage, we first measure the space required for database sizes of $10^3$–$10^6$. We issue a different number of queries $(0, 20, \ldots, 20000)$ to demonstrate the disk space required by RSSE-LC since its storage overhead increases with the number of newly queried elements (*i.e.*, $s$ in Table 2). Each

Table 6: Search Efficiency of RSSE-BRC and RSSE-LC

| Scheme | Time for searching a range of $\ell$ (ms) | | | | | |
|---|---|---|---|---|---|---|
| | $\ell = 5 \times 10^2$ | | $\ell = 5 \times 10^3$ | | $\ell = 5 \times 10^4$ | |
| | First | Rep. | First | Rep. | First | Rep. |
| RSSE-BRC | 71.34 | 41.80 | 110.35 | 57.41 | 283.01 | 195.16 |
| RSSE-LC | 74.22 | 45.49 | 121.82 | 61.81 | 298.70 | 204.22 |

* "First" denotes a query issued for the first time.
"Rep." denotes the repeats of a previous query.

query, covering a range of 500, is uniformly distributed over the keyword domain. In Tables 7 and 8, the number of issued queries is appended to RSSE-LC.

Table 7 illustrates that our RSSE-LC requires less client storage than our generic RSSE-BRC even after many range queries. We remark that it only reflects the storage complexity to some degree, as RocksDB requires additional storage for the long-term startup information and periodically updated log files. The performance may vary when different data structures are used for instantiation.

For a more precise illustration of the storage overhead, we measure the actual number of the stored state tuples in the same setting. Note that the state tuple number of RSSE-LC-0 is essentially the number of the existing keyword.

RSSE-LC only stores information of leaf nodes in the implicit binary tree while RSSE-BRC additionally stores information of elements from the leaf to the root. Table 8 confirm that before any query has been issued, the client storage of RSSE-LC is significantly less than that of RSSE-BRC.

Table 7: Storage Overhead of RSSE-BRC and RSSE-LC

| Scheme | Storage for $n$ keyword-file pairs (MB) | | | |
|---|---|---|---|---|
| | $n = 10^3$ | $n = 10^4$ | $n = 10^5$ | $n = 10^6$ |
| RSSE-BRC | 0.86 | 6.19 | 37.08 | 123.17 |
| RSSE-LC-0 | 0.16 | 1.31 | 12.32 | 38.87 |
| RSSE-LC-20 | 0.16 | 1.32 | 12.34 | 38.90 |
| RSSE-LC-200 | 0.17 | 1.68 | 14.92 | 46.12 |
| RSSE-LC-2000 | 0.30 | 2.45 | 18.60 | 55.23 |
| RSSE-LC-20000 | 0.63 | 4.97 | 28.33 | 71.02 |

## 9 Conclusion

We study DSSE for range queries [10], highlight the importance of forward security [5,24] in this context, and extend the definition and generic construction for backward privacy [6] for range DSSE. We design a variant of file injection attack [33], which aims at revealing the queried range instead of a single keyword, to illustrate that a recently proposed construction [34] fails to provide forward security as claimed. Based on an implicit realization of the tree-like range covering technique, we propose a generic construction and a less-client-storage construc-

Table 8: Comparison for The Number of State Tuples

| Scheme | # of state tuples for $n$ keyword-file pairs | | | |
|---|---|---|---|---|
| | $n = 10^3$ | $n = 10^4$ | $n = 10^5$ | $n = 10^6$ |
| RSSE-BRC | 11178 | 78319 | 454955 | 1610020 |
| RSSE-LC-0 | 996 | 9955 | 95457 | 644650 |
| RSSE-LC-20 | 1009 | 10005 | 95572 | 644814 |
| RSSE-LC-200 | 1101 | 10487 | 96710 | 646445 |
| RSSE-LC-2000 | 1765 | 14536 | 106126 | 659573 |
| RSSE-LC-20000 | 4350 | 33076 | 165537 | 754111 |

tion of range DSSE, both of which are forward-secure and backward-secure. Our experiments demonstrate their high efficiency.

Our work complements existing research for supporting SQL over encrypted databases [16] without using order-preserving encryption which is vulnerable to reconstruction attack [19,14]. We left as future works to consider integration of our range DSSE framework with the techniques for reducing dimension [32] or achieving verifiability [26].

## References

1. Amjad, G., Kamara, S., Moataz, T.: Breach-resistant structured encryption. PoPETs (1), 245–265 (2019)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
3. Boldyreva, A., Chenette, N., O'Neill, A.: Order-preserving encryption revisited: Improved security analysis and alternative solutions. In: Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings. pp. 578–595 (2011)
4. Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., Zimmerman, J.: Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II. pp. 563–594 (2015)
5. Bost, R.: $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria, October 24-28, 2016. pp. 1143–1154 (2016)
6. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1465–1482 (2017)
7. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, Denver, CO, USA, October 12-6, 2015. pp. 668–679 (2015)
8. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In:

Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. pp. 353–373 (2013)
9. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 1038–1055 (2018)
10. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.N.: Practical private range search revisited. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 185–198 (2016)
11. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.N., Papamanthou, C.: Practical private range search in depth. ACM Trans. Database Syst. **43**(1), 2:1–2:52 (2018)
12. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. In: Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II. pp. 123–145 (2015)
13. Grubbs, P., Lacharité, M., Minaud, B., Paterson, K.G.: Pump up the volume: Practical database reconstruction from volume leakage on range queries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 315–331 (2018)
14. Grubbs, P., Sekniqi, K., Bindschaedler, V., Naveed, M., Ristenpart, T.: Leakage-abuse attacks against order-revealing encryption. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 655–672 (2017)
15. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014. pp. 310–320 (2014)
16. Kamara, S., Moataz, T.: SQL on structurally-encrypted databases. In: Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I. pp. 149–180 (2018)
17. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers. pp. 258–274 (2013)
18. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012. pp. 965–976 (2012)
19. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1329–1340 (2016)
20. Kerschbaum, F., Tueno, A.: An efficiently searchable encrypted data structure for range queries. CoRR abs/1709.09314 (2017)
21. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Proceedings of the 2013 ACM SIGSAC

Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013. pp. 669–684 (2013)
22. Lacharité, M., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. pp. 297–314 (2018)
23. Lai, R.W.F., Chow, S.S.M.: Parallel and dynamic structured encryption. In: Security and Privacy in Communication Networks - 12th International Conference, SecureComm 2016, Guangzhou, China, October 10-12, 2016, Proceedings. pp. 219–238 (2016)
24. Lai, R.W.F., Chow, S.S.M.: Forward-secure searchable encryption on labeled bipartite graphs. In: Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings. pp. 478–497 (2017)
25. Markatou, E.A., Tamassia, R.: Mitigation techniques for attacks on 1-dimensional databases that support range queries. Cryptology ePrint Archive, 2019/396 (2019)
26. Ogata, W., Kurosawa, K.: Efficient no-dictionary verifiable searchable symmetric encryption. In: Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers. pp. 498–516 (2017)
27. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Advances in Cryptology - EUROCRYPT 1999, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding. pp. 223–238 (1999)
28. Song, X., Dong, C., Yuan, D., Xu, Q., Zhao, M.: Forward private searchable symmetric encryption with optimized I/O efficiency. IEEE Trans. Dependable Sec. Comput. (2018), online first
29. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014 (2014)
30. Sun, S., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 763–780 (2018)
31. Wang, Q., He, M., Du, M., Chow, S.S.M., Lai, R.W.F., Zou, Q.: Searchable encryption over feature-rich data. IEEE Trans. Dependable Sec. Comput. **15**(3), 496–510 (2018)
32. Wu, S., Li, Q., Li, G., Yuan, D., Yuan, X., Wang, C.: ServeDB: Secure, verifiable, and efficient multidimensional range queries on outsourced database. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macau, April 8-11, 2019 (2019), to appear
33. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 707–720 (2016)
34. Zuo, C., Sun, S., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II. pp. 228–246 (2018)

## A Security for Our Generic Construction

Our generic construction relies on an underlying forward-secure DSSE, which normally leaks the search pattern (sp) and the update history (hist) for the keywords (*e.g.*, [5,24,28]). 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.

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

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

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

For $\mathcal{L}^{\mathsf{Stp}}$, our scheme leaks nothing. For $\mathcal{L}^{\mathsf{Updt}}$, we only leak the size of keyword space $|\mathcal{W}|$, which is a constant value since the initialization. Hence, the simulator directly invokes the simulator-algorithm of $\Pi$ to simulate Setup and Update.

To simulate the 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 $\Pi$ for every element in BRC of $q$ to simulate Search.

Since $\Pi$ 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 [33].

## B Security for Our Specific Range DSSE

We summarize the leakage functions $\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}$ of our less-client-storage range DSSE construction as follows.

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

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

*Proof.* We derive a game sequence from the real-world game $\mathbf{Real}_{\mathcal{A}}(1^n)$ to the last game which is exactly the ideal-world game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^n)$. By showing that each game (except the first) is indistinguishable from its previous one, we conclude that the adversary cannot distinguish $\mathbf{Real}_{\mathcal{A}}(1^n)$ from $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^n)$ with non-negligible probability. Without loss of generality, we assume that the adversary $\mathcal{A}$ makes at most $q_1$ and $q_2$ queries to the $H_1$ oracle and the $H_2$ oracle respectively, and the output length of PRF is $\lambda$.

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

**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 $(\tau, nym_w)$ pairs. When a new $\tau$ 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 then 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 $\mathsf{Update}$, *i.e.*, $H_1(k_\tau||(c+1))$, $G_3$ picks a random string from $\{0,1\}^{\mu_1}$ and stores it in a map $\mathbf{T}_{\mathsf{a}}$:

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

Then, during $\mathsf{Search}$ of the client, if $k_w \neq \perp$ when $(nym_w, k_w, c)$ is collected in $t_{\mathsf{q}}$, we update the reference table $\mathsf{H}_1$ 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 $(k_\tau, c+1)$ is generated in $\mathsf{Update}$ but will not be updated in $\mathsf{H}_1$ until a corresponding search query is executed. If the adversary queries $\mathsf{H}_1$ for $(k_\tau, c+1)$ before $\mathsf{Search}$, the returned value which is a random string picked by the random oracle will have an overwhelming probability to be different from the one programmed for $\mathsf{H}_1[k_w||i]$ (*i.e.*, $\mathsf{H}_1[k_\tau||i]$) later during $\mathsf{Search}$. Once this inconsistency is observed, the adversary could figure out s/he is in $G_3$. 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 $\mathsf{BAD}$ is the event that the inconsistency happens.

Since $k_\tau$ is sampled from $\{0,1\}^{\lambda}$, the probability that the adversary queries $\mathsf{H}_1$ for it equals to $2^{-\lambda}$. 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 $\mathsf{Update}$ (*i.e.*, $val := (id||op) \oplus H_2(k_\tau||(c+1))$), $G_3$ picks a random string from $\{0,1\}^{\mu_2}$ and stores it in a map $\mathbf{T}_{\mathsf{a}}$:

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

Then, during $\mathsf{Search}$ of the client, if $k_w \neq \perp$ when $(nym_w, k_w, c)$ is collected in $t_{\mathsf{q}}$, we update the reference table $\mathsf{H}_2$ 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

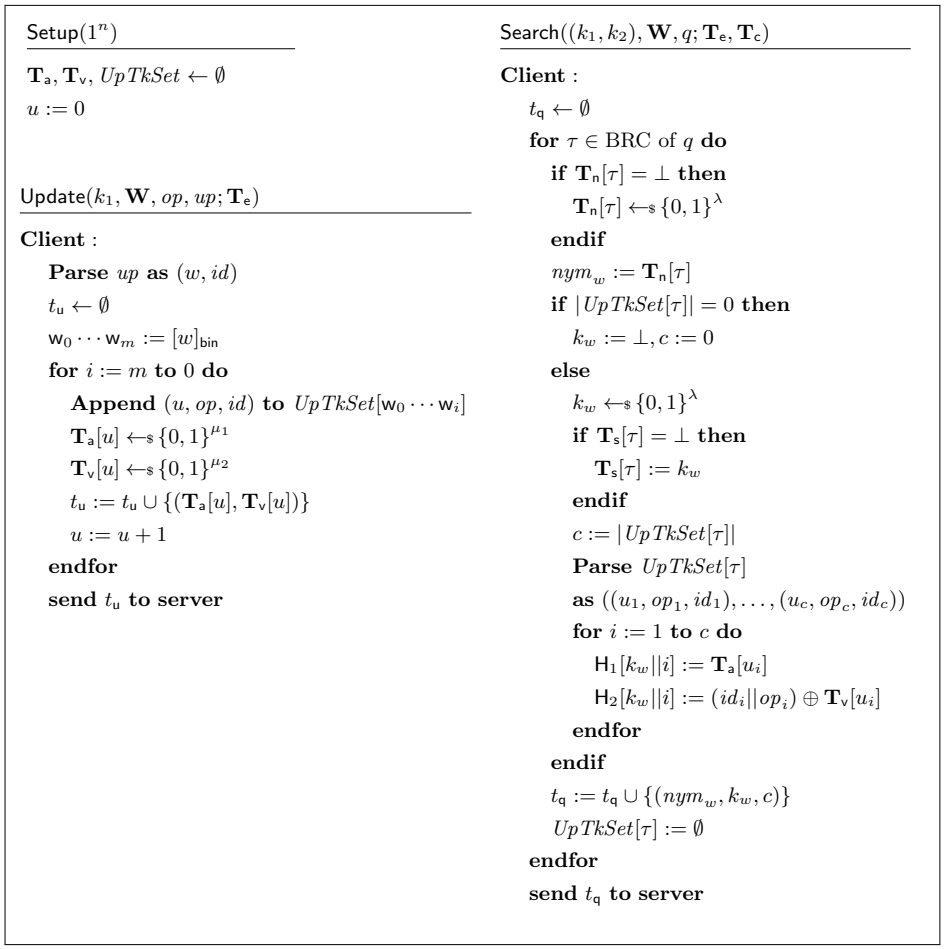


Fig. 4: Description of Game $G_5$

discovers the inconsistency of the random oracle $H_2$ is at most $\frac{q_2}{2^\lambda}$ which is negligible. Thus, we can conclude that $G_3$ and $G_4$ are indistinguishable.

**Game** $G_5$: We present $G_5$ in Figure 4. 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.*, $k_\tau$) is sampled during Search. Also, instead of directly mapping $k_\tau||(c+1)$ to the values picked for $\mathbf{T}_\mathsf{v}$ and $\mathbf{T}_\mathsf{a}$, we implicitly map $k_\tau||(c+1)$ 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 $\Pr[G_4 = 1] = \Pr[G_5 = 1]$.

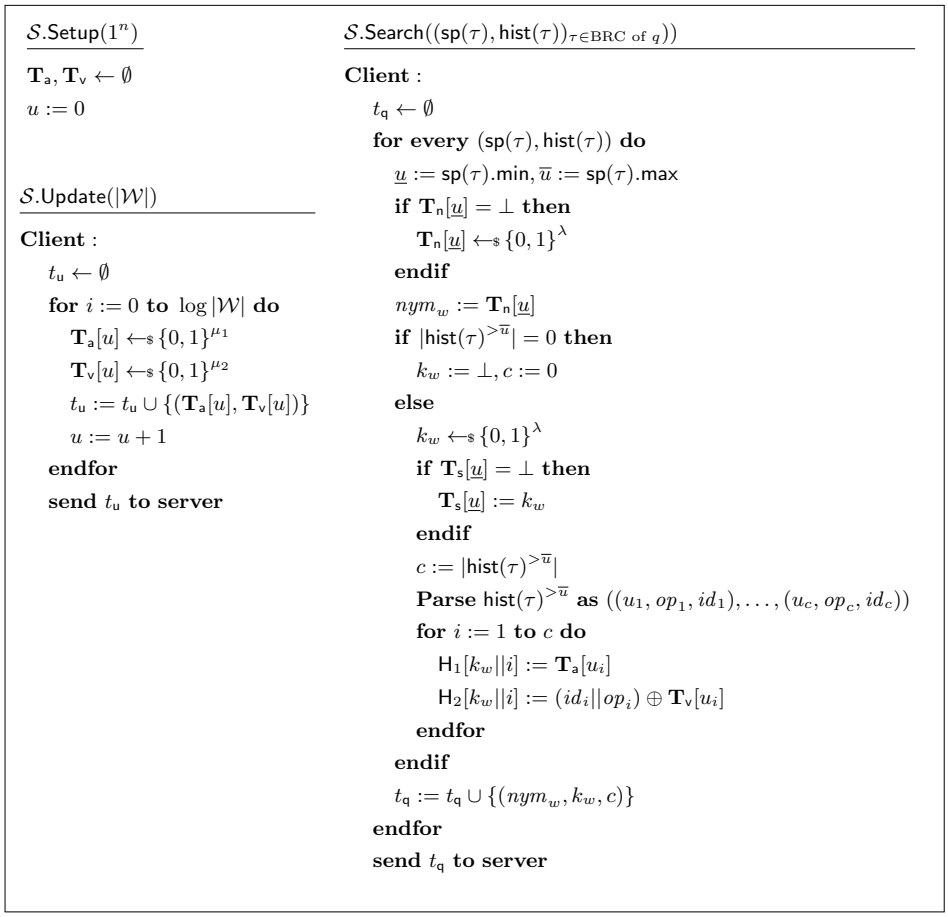


Fig. 5: Description of Simulator $\mathcal{S}$

**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 $\mathcal{L}^{\mathsf{Updt}}$ is the size of the keyword domain $|\mathcal{W}|$. The search leakage $\mathcal{L}^{\mathsf{Srch}}$ contains the search pattern $\mathsf{sp}$ and the update history $\mathsf{hist}$ for every element in the BRC of the queried range.

We present $G_6$ in Figure 5. $\underline{u} := \mathsf{sp}(\tau).\mathsf{min}$ denotes the timestamp when $\tau$ is retrieved for the first time. Instead of using the unknown $\tau$ directly, the simulator uses $\underline{u}$ to uniquely identify the items related to $\tau$ from $\mathbf{T}_\mathsf{s}$ and $\mathbf{T}_\mathsf{n}$. $\overline{u} := \mathsf{sp}(\tau).\mathsf{max}$ denotes the timestamp when $\tau$ 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 $\tau$ 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 $|\Pr[\mathbf{Real}_\mathcal{A}(1^n) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^n) = 1]| \leq \mathsf{negl}(n)$.