Design and Analysis of a Stateless Document Database Encryption Scheme

Seny Kamara
MongoDB

Tarik Moataz
MongoDB

August 10, 2023

Abstract

The problem of designing end-to-end encrypted databases has received a lot of attention in large part because having commercially-available encrypted databases would decrease the impact and occurrence of data breaches. Most of the research on encrypted database systems has focused on relational databases but over the last ten years NoSQL and, in particular, document databases have gained a great deal of popularity in Industry.

In this work, we design the first encrypted document database scheme. A key focus of our work is in designing a scheme that is practical not only in terms of asymptotic and concrete efficiency but also with respect to real-world constraints that emerge when trying to build and deploy real database systems at scale for commercial use. These constraints present many new technical challenges that have not been considered in the research literature before.

Contents

1 Introduction

The problem of searching on encrypted data has been studied for the last twenty years. The initial motivation was to design searchable symmetric encryption (SSE) schemes to encrypt unstructured document collections while supporting keyword search [54]. Since, the field of encrypted search has expanded beyond SSE to consider more complex problems like the design of encrypted databases. This shift was motivated in large part by the realization that having commercially-available end-to-end encrypted databases would have a tremendous impact on data privacy since databases are where most data is stored and processed.

Encrypted relational databases. The first encrypted database schemes focused on relational databases and used a quantization-based approach [37] and property-preserving encryption [1] (PPE), which leaked approximations of the data items and frequency and order information, respectively. In both of these approaches, leakage was available from one database-level snapshot; that is, with access to one encryption of the database and no additional information like transcripts of operations or auxiliary knowledge of the database and/or queries. Motivated by this, a new generation of encrypted relational databases were designed based on structured encryption (STE) that provided improved leakage profiles in stronger adversarial models [19, 40, 43].

Document databases. While relational databases are an important class of database systems, NoSQL and, in particular, document databases have gained a great deal of popularity in industry over the last decade. Roughly speaking, a document database stores data as a collection of structured documents each of which can be viewed as a set of field/value pairs. Document databases are queried using an expressive query language based on matching field values. Some examples, include "find all documents with field $f = v$" or "find all documents with field $f_1 = v_1$ and $f_2 = v_2$". The most well-known document databases include MongoDB, Amazon's DocumentDB, Microsoft's CosmosDB, Couchbase and CouchDB.

Designing a real-world encrypted document database. Perhaps surprisingly, as far as we know, the problem of designing an encrypted document database has not been considered in the past. We note that this problem is very different from the problem of designing an SSE scheme. SSE encrypts document *collections* and supports keyword search; specifically, the documents are unstructured in the sense that they are simply set of keywords and keyword search queries only need to return documents that contain the queried keyword. In this work, we propose the first STE-based encrypted document database scheme. Unlike SSE, an encrypted document *database* encrypts a collection of *structured* documents each of which is a set of keyword/value pairs and has to support more complex queries. An additional and important goal of our work, however, is to design a *practical* solution in several respects including asymptotic and concrete efficiency, scalability, concurrency and other real-world considerations we will discuss below. These practical considerations present many technical challenges but need to be solved to make a solution usable in practice.

Statelesness. One of the real-world constraints that our solution addresses is support for multiple clients. In practice, databases are accessed by many clients so any practical encrypted database solution must work in the multi-writer multi-reader (MWMR) setting. Designing for the multi-writer setting is much more challenging than the standard single-writer single-reader setting considered in previous work [19,40,43]. One of the biggest challenges in designing MWMR schemes is dealing with state. All modern dynamic STE schemes require the client to keep state which becomes difficult to manage in a multi-client setting because clients need to maintain a consistent view of it. Another important consideration in our setting is that that clients can crash at unexpected time which would cause state information to be lost. For these reasons, one of our main technical goals will be to design schemes that are stateless.

Concurrency. Another challenge of the MWMR setting is that clients can issue update operations at the same time which can cause contention and reduce update throughput. A naive way of handling this would be to use a multi-threaded implementation of a SWSR construction where the update operations are protected by locks or wrapped in transactions. This would guarantee that updates are executed “safely” since they are executed sequentially but would lead to very poor throughput. Instead, our constructions will address this challenge in a way that supports high throughput at the cost of a slight decrease in query efficiency.

Interaction. Interaction is known to provide several advantages in STE but in practical settings it can lead to very expensive operations since a single round trip over the Internet can cost 50 to 100ms. For comparison, plaintext transactional database queries typically run in a few milliseconds.

Client-side filtering. When designing encrypted search solutions, it can be useful to filter results at the client. One example is to support conjunctive queries for, say, $f_1 = v_1$ and $f_2 = v_2$. Here, one could execute a query for $f_1 = v_1$ on the server and filter out the subset of documents in the result set that match $f_2 = v_2$. Client-side filtering, however, is highly non-trivial to implement in a database setting because it requires implementing a module that is capable of parsing and interpreting documents at the client.

Snapshot security. There are two adversarial models usually considered in the encrypted search literature: (1) the persistent model, where the adversary corrupts the server; and (2) the snapshot model, where the adversary is only given access to the the encrypted database. Clearly, security against persistent adversary is preferred over snapshot security but here we focus on snapshot security for two reasons: (1) it is not clear that designing an efficient, non-interactive dynamic scheme that is secure against a persistent adversary and stateless is possible given that all known STE dynamic STE schemes require state; (2) snapshot adversaries capture real-world security concerns of some database users; and (3) the state of the art solutions available in industry are based on deterministic encryption and leak frequency information even against snapshot adversaries.

1.1 Our Contributions

In this work, we design a document database encryption scheme OST that is efficient, stateless, concurrent and non-interactive. We make the following contributions.

Security definitions. We formalize security against snapshot adversaries in the ideal/real-world paradigm typically used to capture the security of secure multi-party computation protocols. Our definitions are similar in spirit to the more standard definitions of [23] but the use of ideal functionalities allows us to capture the security of different STE schemes more concisely, i.e., without needing a separate definition for each data structure we consider in this work.

A new way to model leakage. The standard way to model leakage is to use functions that map data and/or operations to some observed leakage. Typically, each operation is associated with a set of leakage functions, called patterns, and the collection of these leakage patterns is a leakage profile. This approach, proposed in [21,23], works well for leakage that is relatively simple and only a function of a single type of operation. Our construction, however, is complex as it makes use of a multi-map encryption that itself is based on two multi-map encryption schemes, a dictionary encryption scheme and a set encryption schemes. Furthermore, some of the leakage we need to reason about spans across operations. Because of this, analyzing the leakage of our construction using the standard functional model of leakage quickly became unmanageable. To address this, we propose a new way to model and analyze leakage. In our approach, a leakage profile is represented as a function $\mathcal{L}$ that takes as input the data structure and a sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_n)$ and outputs a leakage graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ that captures the entire leakage of the scheme as follows. Each operation op_i is associated with a vertex vx_i composed of a public component and a private component. The public component holds information that is visible to the adversary whereas the secret component holds information that is not visible to the adversary and that the leakage function can use to establish adversarially-visible edges between vertices. The edges in the graph capture correlations between operations and the secret component makes it possible to formally and concisely describe these correlations when describing leakage and writing proofs.

We acknowledge that our new graph-based approach to modeling leakage may seem unnecessary and complex at first glance compared to the more traditional functional approach. We stress, however, that we introduced this model out of necessity as the proofs using the functional approach became unwieldly and too difficult to manage.

A stateless concurrent encrypted multi-map. Our main construction OST relies heavily on a new multi-map encryption scheme Ω that is stateless, non-interactive and concurrent. The scheme itself makes use of four encrypted structures as building blocks; each one with different characteristics and used for a different purpose. The first structure EMM_M stores the input multi-map but achieves statelessness by allowing for overwrites. The second structure EDX_C, however, is used to store metadata about EMM_M needed to avoid overwriting items in EMM_M. ESET_S is used to store information needed to compact EDX_C; that is, to reduce its space consumption. Finally, EMM_R is used to store information that enables efficient conjunctive search and helps us deal with limitations of the underlying database management system which we discuss below. The most complex of our building blocks is EMM_C so we provide an overview of its underlying techniques.

More precisely, EMM_M is a dictionary-based multi-map encryption scheme but without any client state. Recall that in these constructions, the state stores counters for every label. These counters are used to generate unique tags that are then used as labels in an underlying (unencrypted) dictionary. Omitting the state obviously makes the EMM stateless but it introduces a new problem which is that pairs can now be overwritten since no counter information is available. To address this, we store and manage the counters in an auxiliary encrypted structure EDX_C that maps the labels in EMM_M to their latest counters. The idea is that, when querying EMM_M, clients can first query EDX_C so that the server can recover the latest counter needed to query EMM_M. And after updating EMM_M, clients can edit EDX_C so that the counter associated with the label is updated to the new counter. Unfortunately, this high-level idea does not work as-is. In particular, there is a subtle security problem that must be overcome: if edits to EDX_C are done "in-place" then a modification at the same location of EDX_C reveals to a multi-snapshot adversary that the same label was updated. To address this, EDX_C must be immutable and handle updates "out of place". We achieve this by instantiating EDX_C with a specially-constructed encrypted multi-map that is stateless, append-only and handles get operations in logarithmic time.

Conjunction and lookup limitations. With the above structures, OST can already support conjunctive queries but it comes at the cost of a higher computational complexity. Specifically, it would cost m times the cost of a standard query, where m is the number of clauses in the conjunction. This overhead can be prohibitive and particularly wasteful when a conjunction is composed of clauses with values that have both very small and high frequencies. Instead, we leverage an idea first introduced in [18]. The idea consists of first performing a query on the conjunctive term with the smallest frequency and then only executing membership test operations on the remaining values. To implement this idea we introduce a fourth structure EMM_R that serves as an encrypted set structure and allows us to significantly reduce query overhead in the conjunctive case. But EMM_R has an additional and a more fundamental use in OST. The underlying database system OST was designed for can only support a fixed number of standard lookup operations which means that the number of addresses we can read from in EMM_M is upper-bounded by a value `limit`. To handle this limitation, whenever the frequency of the query term is higher than `limit`, we use a form of linear scan where EMM_R is used to test whether a document exists or not. Note that `limit` is very high so the likelihood of using the scan-like approach in practice is very unlikely.

Compaction of Ω. The design described so far achieves statelessness, correctness and query efficiency but has one major limitation: it is not space efficient. In fact, the space complexity of the structures described so far is

$$O\left(\sum_{\ell\in\mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell] + \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}\right),$$

where MM is the input multi-map and $\#\mathsf{puts}$ are the total number of puts. Note that this depends on the total number of puts *ever made* and not on the size of the input multi-map. To address this, we use a process called *compaction* to remove stale data from EDX_C and bring the size down to

$$O\left(\sum_{\ell\in\mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell] + c\cdot\#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c\right),$$

where c is the number of compactions and $\#\mathsf{puts}$ are the total number of puts performed since the last compaction (or initialization if no compaction has occurred). The compaction process is executed by the server which means it needs access to information stored in EDX_C. More precisely, it needs the ability to query these structures, to delete certain pairs and to add new ones. To enable this, the client generates get and put tokens for EDX_C whenever it executes a put for EMM_M and stores these tokens in an auxiliary encrypted set structure ESET_S that is used at compaction time.

Security. Our scheme is designed to achieve database-level multi-snapshot security [3]. As we show in Section 10, it leaks the number and time of insert, update and delete operations, the size of the database and which document was updated or deleted. After compaction, OST also reveals the number of unique values inserted since the last compaction. We note that this is considerably less leakage in a stronger adversarial model than previous work on encrypted database systems.

Efficiency. OST achieves $O(r)$ query communication complexity which is optimal and

$$O\left(p \cdot \log\left(\sum_{f\in\mathbf{F}} \left(c \cdot p_f \cdot \#\mathbb{S}_f + \#\mathsf{insUp}_{c,f}\right)\right) + r + \#\mathsf{delUp}\right)$$

query computational complexity, where r is the number of documents that match the query, $\mathbf{F}$ are the fields in the document database, c is the number of compactions executed, $\mathbb{S}_f$ is the support (or value space) of field f, p_f is the contention factor of f which is a parameter that can be tuned to tradeoff the throughput of concurrent insert and update operations, $\#\mathsf{insUp}_{c,f}$ is the number of inserts and updates made on field f since the last compaction, and $\#\mathsf{delUp}$ is the number of deletes and updates made on the field/value being queried. After a compaction operation and assuming that the number of updates and deletes $\#\mathsf{delUp}$ is dominated by the response r, then the computation complexity can be simplified to

$$O\left(p \cdot \log\left(\sum_{f\in\mathbf{F}} c \cdot p_f \cdot \#\mathbb{S}_f\right) + r\right).$$

For more details on the asymptotics, we refer the reader to Section 10. Note that OST's queries are almost optimal except for an additive sub-linear term in the size of the database and the number of updates made to a specific field/value pair. Insert complexity is

$$O\left(\#\mathbf{F} \cdot \log\left(\sum_{f\in\mathbf{F}} \left(c \cdot p_f \cdot \#\mathbb{S}_f + \#\mathsf{insUp}_{c,f}\right)\right)\right),$$

which shows that inserts require only a logarithmic factor per field in the inserted document.

Remarks on implementation and locality. Though the construction we present in this work has been implemented in a real-world commercial database we chose not to discuss implementation in this work. There are two reasons for this: (1) the architecture and implementation is non-trivial and needs to be discussed on its own as it would considerably lengthen and complicate the current work; and (2) our focus here is on the security of the underling scheme OST and extending the scope to include how the system is architected and built would be a distraction. Also, we note that we plan on publishing work focused on the system's architecture, design and performance in future work. For the purposes of this work, we can report that the implementation is highly efficient and, in the worst case, was $8\times$ the performance of the DBMS on plaintext data.[1]

An interesting point about the implementation is that, contrary to what has been stated in many previous works, the lack of locality did not impact the performance of the system in any meaningful way. What we mean by this is that, while locality could potentially improve performance, the efficiency of our scheme was high enough that it was decided that the design and engineering challenges associated with local schemes were not worth pursuing at the moment (though this may change in the future).

2 Related Work

Encrypted search is the area in cryptography that focuses on the design, cryptanalysis and implementation of protocols and systems that support search on encrypted data. Encrypted search was first considered explicitly by Song, Wagner and Perrig in [54] which introduced the notion of searchable symmetric encryption (SSE). Curtmola, Garay, Kamara and Ostrovsky later introduced and formulated the notion of adaptive semantic security for SSE together with the first sub-linear and optimal-time SSE constructions [23]. Chase and Kamara introduced the notion of structured encryption [21] which generalizes index-based SSE schemes [23] to arbitrary data structures. The most common and important type of STE schemes are multi-map encryption schemes which are a basic building block in the design of sub-linear SSE schemes [17, 23, 47], expressive SSE schemes [18, 25, 28, 42, 53] and STE-based encrypted databases [19, 40, 43, 45]. STE and encrypted multi-maps have been studied in the context of SSE along several dimensions including dynamism [13, 14, 17, 30, 38, 46, 47, 53] and I/O efficiency [5, 6, 17, 20, 24, 26, 51]. In this work, we consider snapshot security which was introduced by [3] in the context of structured encryption.

Other approaches for encrypted search include oblivious RAMs (ORAM) [33], secure multi-party computation [8], functional encryption [12] and fully-homomorphic encryption [31] as well as solutions based on deterministic encryption [7] and order-preserving encryption (OPE) [11].

Encrypted relational databases. As far as we know the first encrypted relational DB solution was proposed by Hacigümüs et al. [37] and was based on quantization. Roughly speaking, the attribute space of each column is partitioned into bins and each element in the column is replaced with its bin number. Popa et al. proposed CryptDB [1] which was the first non-quantization-based solution and can handle a large subset of SQL. Instead of quantization, CryptDB relies on property-preserving encryption schemes like deterministic [7] and order-preserving [2, 11] encryption. The CryptDB design influenced the Cipherbase system from Arasu et al. [4] and the

[1]In terms of absolute numbers, the worst-case we saw in our experiments was 38ms which was to insert documents with 20 fields, 3 of them being encrypted. While this might seem like a small number of encrypted fields, this particular workload was designed based on a survey of large real-world customer needs. Finds ranged from 3ms to 26ms depending on the workload.

SEEED system from Grofig et al. [34]. The security of these PPE-based solutions was later studied in [9,27,52].

Cryptanalysis. In addition to the design of encrypted algorithms, encrypted search also focuses on leakage attacks. The first such attack was proposed by Islam, Kuzu and Kantarcioglu and was a known-data query-recovery attack that exploited the co-occurrence pattern and knowledge of a (large) fraction of the dataset. This was followed up by several works including [10,16,30,35,49,50,55]. For a sumamry of leakage attacks and their practical evaluations we refer the reader to [41]. We stress that none of these attacks apply to our work since they exploit leakage patterns that our constructions do not reveal.

3 Preliminaries

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$. $a := b$ means that a is set to b and $a \overset{\circ}{=} \mathsf{expr}$ means that a is defined to be the expression expr. The output y of a probabilistic algorithm $\mathcal{A}$ on input x is denoted by $y \leftarrow \mathcal{A}(x)$. The output y of a deterministic algorithm $\mathcal{A}$ on input x is denoted by $y := \mathcal{A}(x)$. If S is a set then $x \overset{\$}{\leftarrow} S$ denotes sampling from S uniformly at random. Given a sequence $\mathbf{s}$ of n elements, we refer to its ith element as s_i and to s_i's rank in $\mathbf{s}$ as $\mathsf{rank}_{\mathbf{s}}(s_i)$. If S is a set then $\#S$ refers to its cardinality. Throughout, k will denote the security parameter.

Graphs. A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of a set of vertices $\mathbf{V}$ and a set of edges $\mathbf{E}$ over $\mathbf{V}$. The sum of two graphs $\mathbf{G}_1$ and $\mathbf{G}_2$ is defined as the graph $\mathbf{G}_1 + \mathbf{G}_2 \overset{\circ}{=} \big(\mathbf{V}_1 \cup \mathbf{V}_2, \mathbf{E}_1 \cup \mathbf{E}_2\big)$. We sometimes also write this as $\mathbf{G}_1 + (\mathbf{V}_2, \mathbf{E}_2)$. Given a vertex $\mathsf{vx} \in \mathbf{V}$ and a subset of vertices $\mathbf{V}' \subseteq \mathbf{V} - \{\mathsf{vx}\}$, we denote the set of edges $\big\{(\mathsf{vx}, \mathsf{vx}')\big\}_{\mathsf{vx}' \in \mathbf{V}'}$ by $\mathsf{vx} \times \mathbf{V}'$.

Dictionaries & multi-maps. A dictionary $\mathsf{DX} : \mathbb{L} \rightarrow \mathbb{V}$ over a label space $\mathbb{L}$ and value space $\mathbb{V}$ is a collection of label/value pairs $\{(\ell_i, v_i)\}_{i \le n}$, where $\ell_i \in \mathbb{L}$ and $v_i \in \mathbb{V}$. Dictionaries typically support Get and Put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of putting the value v_i in DX with label ℓ_i. The removal of a pair (ℓ, v) from DX is written as $\mathsf{DX} - (\ell, v)$ or sometimes $\mathsf{DX} - \ell$. We denote by $\mathsf{DX}^{-1}[v]$ the set of labels in DX that are associated with v.

A multi-map $\mathsf{MM} : \mathbb{L} \rightarrow \mathbb{V}$ over a label space $\mathbb{L}$ and a value space $\mathbb{V}$ is a collection of label/tuple pairs $\{(\ell_i, \mathbf{v}_i)_i\}_{i \le n}$ where $\ell_i \in \mathbb{L}$ and each $\mathbf{v} \in \mathbb{V}^n$. Multi-maps typically support Get and Put operations. We write $\mathbf{v}_i := \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] := \mathbf{v}_i$ to denote operation of putting the tuple $\mathbf{v}_i$ to label ℓ_i. If a multi-map supports appends we denote by $\mathsf{MM}[\ell] \overset{+}{:=} v$ the operation that consists of appending the value v to ℓ's tuple. Multi-maps are the abstract data type instantiated by an inverted index. The removal of a pair (ℓ, v) from MM is written as $\mathsf{MM} - (\ell, v)$ or sometimes $\mathsf{MM} - \ell$ and the removal of a value v from a label ℓ' s tuple is written $\mathsf{MM}[\ell] - v$. We denote by $\mathsf{MM}^{-1}[v]$ the set of labels in MM whose tuples include v.

Priority queues. One of our constructions makes use of a priority queue which is a data structure PQ that stores label/value pairs where the labels are from an ordered set and that supports the following three operations: $\mathsf{enqueue}$ which takes as input a label/value pair (ℓ, v) and stores it; $\mathsf{dequeue}$ which returns the value v associated to the smallest label ℓ and removes the pair (ℓ, v) from the queue; and peek which returns the value associated with the smallest label but does not remove the pair. For our purposes we will only use priority queues to store single elements which can be trivially implemented by storing pairs of the form (e, e), where e is the element being stored and retrieved.

Document databases. A document database of size n holds n documents $\{\mathbf{D}_1, \ldots, \mathbf{D}_n\}$ each of which is a set of field/value pairs. For ease of exposition and without loss of generality, we will assume throughout that all documents in a database have the same number of field/value pairs. More precisely, for all $1 \leq i \leq n$, we have $\mathbf{D}_i = \big\{(f_1, v_1), \ldots, (f_m, v_m)\big\}$. Here, we consider document databases $\Delta = (\mathsf{Insert}, \mathsf{Find}, \mathsf{DeleteOne}, \mathsf{UpdateOne})$ that work as follows:

- $\mathsf{Insert}(\mathbf{D})$: takes as input a document $\mathbf{D}$ and inserts it into the database. We write this $\mathsf{DDB} \cup \mathbf{D}$. We require that every document in the database have a $\mathsf{\_id}$ field set to a unique value id.
- $\mathsf{Find}(\mathsf{filter})$: takes as input a filter filter which is either: (1) an exact filter of the form $f = v$; or (2) a conjunctive filter of the form $f_1 = v_1 \bigwedge \cdots \bigwedge f_m = v_m$. The Find operation returns all the documents that match the filter. In the case of an exact filter this includes the documents with field f set to v and in the case of a conjunctive filter this includes the documents such that, for all $1 \leq i \leq m$, field f_i is set to v_i. We sometimes write finds with exact filters as $\mathbf{R} := \mathsf{DDB}[f = v]$ and with conjunctive filters as $\mathbf{R} := \mathsf{DDB}[\bigwedge_{i \in [m]} f_i = v_i]$.
- $\mathsf{DeleteOne}(\mathsf{filter})$: takes as input a filter filter and deletes one of the documents, chosen uniformly at random, that match filter. We sometimes write $\mathsf{DDB} - \mathsf{id}$ as shorthand for executing the Delete operation on $\mathsf{filter} \equiv \mathsf{\_id} = \mathsf{id}$.
- $\mathsf{UpdateOne}(\mathsf{filter}, \mathsf{action})$: takes as input a filter filter and an action action which, in this work, is a field/value pair (f, v). The $\mathsf{UpdateOne}$ operation updates the field f of one document, chosen uniformly at random, that matches filter to the value v.

Symmetric encryption. A symmetric encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm that takes a key K and a message m and returns a ciphertext ct; Dec is a deterministic algorithm that takes a key K and a ciphertext ct and returns m if K was the key under which ct was produced. We denote by $\gamma(\cdot)$, the ciphertext expansion of the scheme, i.e., the size of ciphertexts is $\gamma(|m|)$. Informally, a symmetric encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say that a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle. Note that RCPA-security implies CPA-security and key-privacy, the latter of which guarantees that given two ciphertexts one cannot distinguish whether they were generated under the same key or not. We refer the reader to [39] for formal definitions of these notions.

Pseudo-random functions. In addition to encryption schemes, we also make use of pseudo-random functions (PRF), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. We usually denote the evaluation of a pseudo-random function F with a key K on an input x as $F_K(x)$ but sometimes as $F(K,x)$ for visual clarity. We refer the reader to [48] for standard notions of security for PRFs.

4 Definitions

Encrypted sub-linear search on encrypted data is formally captured by the notion of *structured encryption* (STE) which we define below. Informally, a STE scheme encrypts a data structure in such a way that it can be queried without revealing any useful information about the data or the query.

4.1 Structured Encryption

STE was introduced in [21] as a generalization of index-based [2] SSE schemes [23]. The notion of SSE was introduced in [54] and formalized in [23]. There are several forms of structured encryption. The original definition of [21] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [22], the authors also describe *structure-only* schemes which only encrypt structures. One can also distinguish between *response-hiding* and *response-revealing* schemes: the former reveal the response to queries whereas the latter do not. Many of the STE schemes in this work will also support a compaction operation which can be used to shrink the size of the encrypted structures.

Definition 4.1 (Structured encryption)**.** *A response-hiding, dynamic structured encryption scheme* $\Sigma_{\mathsf{DS}} = (\mathsf{Init}, \mathsf{Add}, \mathsf{Query}, \mathsf{Delete})$ *consists of two-party protocols that work as follows:*

- $(K, st; \mathsf{EDS}) \leftarrow \mathsf{Init}_{\mathbf{C},\mathbf{S}}(1^k; 1^k)$*: is a probabilistic algorithm that takes as input from the client and server a security parameter* 1^k*. It outputs to the client a key* K *and a state* st *and to the server an encrypted structure* EDS*;*

- $(st'; \mathsf{EDS}') \leftarrow \mathsf{Add}_{\mathbf{C},\mathbf{S}}(K, st, a; \mathsf{EDS})$*: takes as input from the client a key* K*, a state* st *and an update* a*; and from the server an encrypted structure* EDS*. It outputs to the client an updated state* st' *and to the server an updated encrypted structure* EDS'*;*

- $(st', r; \perp) \leftarrow \mathsf{Query}_{\mathbf{C},\mathbf{S}}(K, st, q; \mathsf{EDS})$*: takes as input from the client a key* K*, a state* st *and a query* q*; and from the server an encrypted structure* EDS*. It outputs to the client a (possibly) updated state* st' *and a response* r *and to the server* $\perp$*;*

[2]In the literature structure-based schemes are also called index-based schemes.

- $(st'; \mathsf{EDS}') \leftarrow \mathsf{Delete}_{\mathbf{C},\mathbf{S}}(K, st, d; \mathsf{EDS})$*: takes as input from the client a key K, a state st and an item to delete d; and from the server an encrypted structure* EDS*. It and outputs to the client an updated state st' and to the server an updated encrypted structure* EDS'*;*

- $(st'; \mathsf{EDS}') \leftarrow \mathsf{Compaction}_{\mathbf{C},\mathbf{S}}(K, st; \mathsf{EDS})$*: takes as input from the client a key K and a state st and from the server an encrypted structure* EDS*. It outputs an encrypted structure* EDS'*;*

If the scheme does not support delete operations (i.e., it has no Delete *protocol), we say that Σ_{DS} is* semi-dynamic*. A scheme is* stateless *if none of its protocols take as input or output a state st.*

Single-round schemes. A single-round STE scheme has operation protocols that require only one round which consists of the client sending a token to the server and the server returning a message. For example, single-round (response-hiding) query protocols consists of the client sending a query token and the server returning an encrypted data item which the client then decrypts. All the constructions proposed in [21] are single round and many SSE constructions are as well. There are, however, several constructions that are multi-round including [29, 32, 44, 53]. All the constructions in this work are single round.

Protocol decomposition. Given a structured encryption scheme $\Sigma_{\mathsf{DS}} = (\mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Add}_{\mathbf{C},\mathbf{S}}, \mathsf{Query}_{\mathbf{C},\mathbf{S}}, \mathsf{Delete}_{\mathbf{C},\mathbf{S}})$ it will sometimes be useful for us to refer to the different phases of its protocols. For example, assuming the client sends the first message of the query protocol, we will denote by $\mathsf{Query}_{\mathbf{C}_1}(K, st, q)$ the execution of Query's first phase which includes the steps needed for the client to generate and send the first message. Similarly we will denote by $\mathsf{Query}_{\mathbf{S}_2}(\mathsf{EDS}, \mathsf{qtk})$ the execution of the second phase which includes the steps needed for the server to compute and send the second message and so on.

Two-Party structured encryption In a standard STE scheme there is only one party (the client) that can insert data items into the encrypted structure. In this work, we introduce a new form of STE we refer to as *multi-party STE* that allows multiple parties to contribute to a data item. For example, in the case of a dictionary, a *two-party* encrypted dictionary allows the client to insert the label of a label/value pair and the server to insert the value of that pair. Note that multi-party STE is fundamentally different than multi-client STE which focuses on supporting queries made by multiple clients efficiently (i.e., with high concurrent throughput). We now describe the syntax of a two-party STE scheme.

Definition 4.2 (Two-party STE)**.** *A response-hiding, dynamic two-party structured encryption scheme $\Sigma_{\mathsf{DS}} = (\mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Add}_{\mathbf{C},\mathbf{S}}, \mathsf{Query}_{\mathbf{C},\mathbf{S}}, \mathsf{Delete}_{\mathbf{C},\mathbf{S}})$ consists of four polynomial-time algorithms where $\mathsf{Init}_{\mathbf{C},\mathbf{S}}$, $\mathsf{Query}_{\mathbf{C},\mathbf{S}}$, $\mathsf{Delete}_{\mathbf{C},\mathbf{S}}$ are as in Definition 4.1 and $\mathsf{Add}_{\mathbf{C},\mathbf{S}}$ works as follows:*

- $(st'; \mathsf{EDS}') \leftarrow \mathsf{Add}(K, st, a_1; \mathsf{EDS}, a_2)$*: takes as input from the client a key K, a state st and partial update a_1; and from the server an encrypted structure* EDS *and a partial update a_2. It outputs to the client an updated state st' and to the server an updated encrypted structure* EDS*.*

4.2 Adversarial Models

The security of an encrypted database can be analyzed using a variety of adversarial models which can be categorized along two dimensions: (1) the adversary's view, e.g., the database, the disk, certain logs, the memory or even the entire server; and (2) a schedule that determines when and for how long it has access to this view, e.g., a snapshot in time, multiple snapshots in time or continuously.

Adversarial models in the literature. The adversarial models considered in the literature include *persistent adversaries* and *snapshot adversaries*. Roughly speaking, persistent adversaries have access to the entire database server whereas snapshot adversaries only receive the encrypted database. These two models have been useful in research but as first pointed out by [36] they do not necessarily capture real-world implementations of encrypted database *systems*. Indeed, database management systems (DBMS) are very complex systems that satisfy many important requirements that are sometimes in conflict with security. To address this and to better capture the security guarantees of real-world encrypted database systems, we extend and increase the number of adversarial models for encrypted databases. In the following, we briefly and informally describe these new models.

Snapshot views and schedules. A *disk-level snapshot* includes the server's disk. This includes the database or collection, its indexes, various logs and virtual memory swap space. This captures data breaches that occur due to disk theft in data centers or to lost or stolen laptops. Disk-level snapshots are easy to protect against using disk encryption since they only contain data at rest. A *database-level snapshot* includes the database in memory and on disk. This captures database exfiltration attacks due to, e.g., misconfigured access control settings.

The time and cadence of snapshots is described by a schedule. A *single-snapshot* schedule provides the adversary with a single snapshot in time. A *multi-snapshot* schedule is one that provides multiple snapshots at different points in time. An *atomic multi snapshot* schedule is one that provides snapshots only after an operation terminates but never *during* the execution of an operation. A *periodic and atomic multi-snapshot* schedule provides a snapshot after every operation. In the literature, the latter is usually just referred to as a snapshot.

Persistent view. Unlike snapshot views, a persistent view includes all the information on the entire server on a continuous basis. This means that it includes memory, disk, the network, registers and trusted execution environments. This means that, in addition to the contents of memory, they also have access to all the queries ever made to the database. This captures settings where the entire server has been compromised for a long period of time.

4.3 Leakage Profiles & Patterns

Intuitively, we would like an STE scheme to guarantee that

> *the encrypted structure reveals no useful information about the underlying structure and that the tokens reveal no useful information about the underlying query.*

Four our purposes, we will consider "non-useful" any information that can be derived from the security parameter k. For example, this would include information about the structure space or query space. Unfortunately, such a security notion seems hard to achieve efficiently so we would like to weaken it to allow for trade-offs.

Leakage patterns. To capture the leakage of a scheme, we will associate every operation the STE scheme supports with a set of *leakage patterns* and the collection of all of these leakage patterns is the scheme's *leakage profile*. Leakage patterns are (families of) functions over the various spaces associated with the underlying structure. We refer to leakage patterns that reveal an item completely as *identity patterns*, leakage patterns that reveal whether two items are equal as *equality patterns*, leakage patterns that reveal the size of a set as *size patterns* and leakage patterns that reveal the length of an item as *length patterns*. We recall some common leakage patterns as defined in [44]:

- the *query equality* pattern qeq (also known as the search pattern) reveals if and when queries are the same: $\mathsf{qeq}(\mathsf{DS}, q_1, \ldots, q_t) = M$, where M is a binary $t \times t$ matrix with rows and columns indexed by $(q_1, \ldots, q_t)$ and such that $M[q_i, q_j] = 1$ if $q_i = q_j$ and $M[q_i, q_j] = 0$ if $q_i \neq q_j$.
- the *response identity* pattern rid (also known as the access pattern) reveals the responses to queries: $\mathsf{rid}(\mathsf{DS}, q_1, \ldots, q_t) = \big(r_1, \ldots, r_t\big)$;
- the *response length* pattern rlen leaks the number of matching entries: $\mathsf{rlen}\,(\mathsf{DS}, q_1, \ldots, q_t) = (|r_1|, \ldots, |r_t)|)$;
- the *data size* pattern dsize reveals the size of the structure: $\mathsf{dsize}(\mathsf{DS}) = |\mathsf{DS}|$;

We say that a leakage profile is *zero-leakage* (ZL) if it depends only on the security parameter and other public parameters. Note that this does not imply that no leakage occurred but rather that whatever leakage did occur is not useful since it could have been derived solely from the public parameters.

Leakage graphs. Our OST construction is complex and makes use of several lower-level STE schemes which are themselves non-trivial. The complexity of the scheme translates to a complex leakage analysis (though not necessarily complex leakage profile). This complexity motivated us to create a new framework to formalize leakage profiles. Our framework differs considerably from the more traditional "functional" framework introduced in [21] and expanded in [44]. Instead of formalizing leakage profiles as a collection of functions associated to each operation, we represent a leakage profile as a function $\mathcal{L}$ that takes as input the data structure and a sequence of operations and outputs a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ that captures all the leakage of the scheme.

More precisely, consider a data structure DS and a sequence of operations $\mathbf{op} = (\mathsf{init}, \mathsf{op}_1, \ldots, \mathsf{op}_n)$. $\mathcal{L}$ is defined as a *stateful* function that: (1) given init as input returns $\mathbf{G}_0 = (\mathbf{V}_0, \mathbf{E}_0)$, where $\mathbf{V}_0 := \big\langle\, 0, \mathsf{init} \mid \perp \big\rangle$ and $\mathbf{E}_0 = \emptyset$; and (2) given op_t, for $1 \leq t \leq n$, outputs a set of vertices and edges $(\mathbf{V}_t, \mathbf{E}_t)$, where $\mathbf{E}_t$ is over $\bigcup_{i=1}^{t} \mathbf{V}_t$. When $t = 0$, the leakage graph of the sequence is $\mathbf{G}_0$ but when $t \geq 1$ it is defined as $\mathbf{G}_t = \mathbf{G}_{t-1} + (\mathbf{V}_t, \mathbf{E}_t)$. Intuitively, one can view $(\mathbf{V}_t, \mathbf{E}_t)$ as the leakage of operation op_t.

Public/private vertices. Vertices in leakage graphs have a special structure which consists of a *public* component and of a *secret* component. Given a leakage graph, the public component of a vertex is visible to the adversary/simulator but the secret component is not. Intuitively speaking, the public component of a vertex $\mathsf{vx}_t \in \mathbf{V}$ is a set that stores information about op_t that is leaked. The secret component, on the other hand, is a set used by the leakage profile to store information about the operation that will help it define/construct adversarially-visible edges between vertices. Intuitively, these edges capture correlations between different operations. We note that correlations between vertices/operations can exist due to elements of the public component but, by definition, these correlations are public so they are not explicitly modeled by edges; though they can be recovered by just observing the public components of vertices. The purpose of the edges is to capture correlations about *secret* information.

In a leakage graph, we describe a vertex vx with the notation $\left\langle\, P \mid S \,\right\rangle$, where P is the public component and S is the secret component. Given a vertex vx we sometimes write $\mathsf{pub}(\mathsf{vx})$ to refer to its public component and $\mathsf{sec}(\mathsf{vx})$ to refer to its secret component. And given a leakage graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, we write $\mathbf{V}(\, p \mid \cdot \,) \subseteq \mathbf{V}$ to denote the set of vertices in $\mathbf{V}$ whose public component includes p,

$$\mathbf{V}(\, p \mid \cdot \,) \overset{\circ}{=} \big\{\mathsf{vx} \in \mathbf{V} : p \in \mathsf{pub}(\mathsf{vx})\big\}.$$

On the other hand, we write $\mathbf{V}(\, \cdot \mid s \,) \subseteq \mathbf{V}$ to denote the set of vertices in $\mathbf{V}$ whose secret component includes s,

$$\mathbf{V}(\, \cdot \mid s \,) \overset{\circ}{=} \big\{\mathsf{vx} \in \mathbf{V} : s \in \mathsf{sec}(\mathsf{vx})\big\}.$$

We also sometimes write $\mathbf{V}(\, \cdot \mid s_1, \ldots, s_n \,)$ to denote the set of vertices in $\mathbf{V}$ whose secret component includes all of $s_1, \ldots, s_n$. To denote the set of vertices in $\mathbf{V}$ whose public component includes p and secret component includes s we then write

$$\mathbf{V}(\, p \mid s \,) = \left\{\mathsf{vx} \in \mathbf{V} : p \in \mathsf{pub}(\mathsf{vx}) \bigwedge s \in \mathsf{sec}(\mathsf{vx})\right\}.$$

Similarly, we write $\mathbf{G}(\, p \mid \cdot \,)$, $\mathbf{G}(\, \cdot \mid s \,)$ and $\mathbf{G}(\, p \mid s \,)$ to denote the subgraphs of $\mathbf{G}$ induced by the vertices in $\mathbf{V}(\, p \mid \cdot \,)$, $\mathbf{V}(\, \cdot \mid s \,)$ and $\mathbf{V}(\, p \mid s \,)$, respectively.

Operation sequences. Given a sequence of operations $\mathbf{op} = (\mathsf{init}, \mathsf{op}_1, \ldots, \mathsf{op}_n)$, the rank of init is 0 and the rank of op_i is i. If the public components of the vertices include the rank of the corresponding operation, then we denote the set of vertices with rank at least a and at most b by $\mathbf{V}[a, b]$. More precisely, we have

$$\mathbf{V}[a, b] \overset{\circ}{=} \big\{\mathsf{vx} \in \mathbf{V} : a \leq \mathsf{rank}_{\mathbf{op}}(\mathsf{vx}) \leq b\big\}.$$

We sometimes need to refer to the latest operation in a particular subset of vertices $\mathbf{V}^\dagger$ of $\mathbf{V}$. We denote this as

$$\mathsf{maxrank}\left(\mathbf{V}^\dagger\right) \overset{\circ}{=} \max_{\mathsf{vx} \in \mathbf{V}^\dagger} \mathsf{rank}(v),$$

with $\mathsf{maxrank}(\mathbf{V}^\dagger) \overset{\circ}{=} 0$ whenever $\mathbf{V}^\dagger = \emptyset$.

Examples. As a concrete example, recall the query equality which reveals if and when a query is repeated and consider a dynamic multi-map MM and the following sequence of operations

$$\mathbf{op} = \Big(\mathsf{init}, \big(\mathsf{put}, \ell_3, \mathbf{v}_2\big), \big(\mathsf{get}, \ell_3\big), \big(\mathsf{put}, \ell_4, \mathbf{v}_9\big), \big(\mathsf{get}, \ell_3\big), \big(\mathsf{get}, \ell_3\big)\Big).$$

The leakage graph $\mathbf{G}$ of the query equality of $\mathbf{op}$ has vertices

$$\mathbf{V} = \Big\{\mathsf{vx}_1 \overset{\circ}{=} \big\langle\, 0, \mathsf{init} \mid \perp \big\rangle, \mathsf{vx}_2 \overset{\circ}{=} \big\langle\, 1, \mathsf{get} \mid \ell_3 \,\big\rangle, \mathsf{vx}_3 \overset{\circ}{=} \big\langle\, 2, \mathsf{get} \mid \ell_3 \,\big\rangle, \mathsf{vx}_4 \overset{\circ}{=} \big\langle\, 3, \mathsf{get} \mid \ell_3 \,\big\rangle\Big\}$$

and edges $\mathbf{E} = \{(\mathsf{vx}_2, \mathsf{vx}_3), (\mathsf{vx}_2, \mathsf{vx}_4), (\mathsf{vx}_3, \mathsf{vx}_4)\}$ which can also be described as the set of edges between all get vertices/operations in $\mathbf{V}$ with ℓ_3 in their secret component. The query equality leakage subgraph can be described succinctly as the clique $\mathbf{K}(\,\mathsf{get} \mid \ell_3\,)$; that is, the clique over the vertex set $\mathbf{V}(\,\mathsf{get} \mid \ell_3\,)$. Note that the ranks in $\mathbf{K}(\,\mathsf{get} \mid \ell_3\,)$ are with respect to the subsequence $\mathbf{op}' \subset \mathbf{op}$ which only consists of the get operations in $\mathbf{op}$. Also, since every sequence must start with an initialization operation, by convention, we always include the vertex $\big\langle\, 0, \mathsf{init} \mid \perp \big\rangle$.

As another example, consider the same sequence of operations but the leakage profile that includes the operation equality, the rank identity and the operation identity. The resulting leakage graph has vertices

$$\mathbf{V} = \Big\{\mathsf{vx}_1 \overset{\circ}{=} \big\langle\, 0, \mathsf{init} \mid \perp \big\rangle, \mathsf{vx}_2 \overset{\circ}{=} \big\langle\, 1, \mathsf{put} \mid \ell_3 \,\big\rangle, \mathsf{vx}_3 \overset{\circ}{=} \big\langle\, 2, \mathsf{get} \mid \ell_3 \,\big\rangle,$$
$$\mathsf{vx}_4 \overset{\circ}{=} \big\langle\, 3, \mathsf{put} \mid \ell_4 \,\big\rangle, \mathsf{vx}_5 \overset{\circ}{=} \big\langle\, 4, \mathsf{get} \mid \ell_3 \,\big\rangle, \mathsf{vx}_6 \overset{\circ}{=} \big\langle\, 4, \mathsf{get} \mid \ell_3 \,\big\rangle\Big\}$$

and edges

$$\mathbf{E} = \Big\{\big(\mathsf{vx}_2, \mathsf{vx}_3\big), \big(\mathsf{vx}_2, \mathsf{vx}_5\big), \big(\mathsf{vx}_2, \mathsf{vx}_6\big), \big(\mathsf{vx}_3, \mathsf{vx}_5\big), \big(\mathsf{vx}_3, \mathsf{vx}_6\big), \big(\mathsf{vx}_5, \mathsf{vx}_6\big)\Big\}.$$

Modularity. Our main document database encryption scheme OST makes calls to our stateless and concurrent multi-map encryption scheme Ω which itself makes calls to lower-level multi-map, dictionary and set encryption schemes Σ_M, Σ_R, Σ_C and Σ_S. As such, the leakage profile of OST is a function of the leakage profile of Ω which itself a function of the leakage profiles of Σ_M, Σ_C, Σ_S and Σ_R. In our proofs we choose to write the leakage concretely so, for example, the leakage profiles of OST is described in terms of the concrete leakage of its lowest-level building blocks, i.e., Σ_M, Σ_C, Σ_S and Σ_R. Note, however, that when an OST-level operation is executed and itself executes an Ω-level operation that itself executes, say, three Σ_M operations, the adversary will know the relationship between all these operations. That is, it knows that the three Σ_M operations were executed as part of a higher-level Ω operation that itself was executed as part of an even higher-level OST operation.

We capture this information in our model by adding "meta-data" to the public components of the vertices that describes the schemes that lead to its creation. So, for example, an OST-level compaction operation might create a vertex

$$\mathbf{V}_t := \Big\{ \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{init} \mid \perp \big\rangle, \\ \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{comp}, \mathsf{pcount} \mid \perp \big\rangle \Big\}_{f \in \mathbf{F}}$$

which means that the OST-level compaction generates a vertex composed two sub-vertices for every field $f \in \mathbf{F}$. The first sub-vertex describes the leakage that results from initializing an ESET_S structure as part of an Ω-level compaction and the second sub-vertex describes the leakage that results from compacting an EDX_C structure as part of the same Ω-level compaction. The value t denotes the OST-level rank of the operation, the value $t_{\Omega,f}$ denotes the Ω-level rank of the operation, the value $\mathbf{t}_{S,f}$ denotes the Σ_S-level rank of the operation and the value $t_{C,f}$ denotes the Σ_C-level rank of the operation. Note that for visual clarity we do not include the exact OST and Ω operations because those are usually self-evident and can be found from the pseudo-code of the schemes.

4.4 Security Definitions

Here, we formalize the security of STE in the ideal/real-world paradigm [15] typically used to capture the security of secure multi-party computation protocols. The definition guarantees security against static and semi-honest corruptions and is essentially the same as the standard definition of security for STE proposed in [21,23] but expressed in the ideal/real-world paradigm for convenience. Roughly speaking, we require that an execution of the scheme in the real-world is indistinguishable from an execution of an ideal data structure in the presence of a semi-honest adversary that receives atomic database-level multi-snapshots.

Parties. The two executions take place between an environment $\mathcal{Z}$, an adversary which we denote by $\mathcal{A}$ in the real-world execution and by $\mathcal{S}$ in the ideal-world execution, a set of clients $(\mathbf{C}_0, \mathbf{C}_1, \ldots, \mathbf{C}_n)$ and a server $\mathbf{S}$.

Corruptions. One can define various kinds of corruptions based on the adversarial model under consideration. The first is semi-honest *persistent* corruptions in which the adversary is provided with the party' s input, output, incoming and outgoing messages and coins. Another kind, which is our focus in this work, are semi-honest *database-level multi-snapshot* corruptions where the adversary is provided with the encrypted structure after every operation but not the transcript of the operations or the server's coins. Here, we refer to this kind of corruption as *atomic* to emphasize the fact that the snapshots are provided to the adversary *after* each operation has completed and not during an operation.

Real-world execution. In the real-world execution, the environment $\mathcal{Z}$ takes as input a string $z \in \{0,1\}^*$. Given z, the adversary $\mathcal{A}$ corrupts the server. The client $\mathbf{C}_0$ executes $\Sigma_{\mathsf{DS}}.\mathsf{Init}_{\mathbf{C}_0,\mathbf{S}}(1^k)$ with $\mathbf{S}$. $\mathcal{Z}$ then adaptively chooses and sends a polynomial number of commands $(\mathsf{comm}_1, \ldots, \mathsf{comm}_m)$ of the form $\mathsf{comm}_i = (\mathbf{C}_i, \mathsf{op}_i)$, where $\mathbf{C}_i \in \{\mathbf{C}_0, \ldots, \mathbf{C}_n\}$ and op_i is an operation supported by Σ_{DS}. For each of these operations: the parties execute the appropriate

Functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{2D\text{-}AMM}}$

The functionality is parameterized with a leakage profile $\mathcal{L}$ and interacts with n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, a server $\mathbf{S}$ and an ideal adversary $\mathcal{S}$. It stores and manages two-dimensional addressable multi-map MM using the following operations:

- upon receiving $(\mathsf{cid}, \mathsf{init})$ from a client, initialize and store an addressable multi-map MM and send $\big(\mathsf{cid}, \mathsf{init}, \mathcal{L}(\mathsf{init})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{write}, \boldsymbol{\ell})$ from a client and $(\mathsf{write}, \mathbf{v}, \mathbf{a})$ from $\mathbf{S}$, set $\mathsf{MM}[\ell_x, \ell_y] \overset{\mathbf{a}}{:=} \mathbf{v}$ and send $\big(\mathsf{cid}, \mathsf{write}, \mathbf{v}, \mathbf{a}, \mathcal{L}(\mathsf{write}, \mathsf{MM}, \boldsymbol{\ell}, \mathbf{v}, \mathbf{a})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{read}, \boldsymbol{\ell})$ from a client and $(\mathsf{read}, \mathbf{a})$ from $\mathbf{S}$, return $\mathbf{v} \overset{\mathbf{a}}{:=} \mathsf{MM}[\ell_x, \ell_y]$ to the client;
- upon receiving $(\mathsf{cid}, \mathsf{hyperread}, \ell_x)$ from a client and $(\mathsf{hyperread}, \mathbf{L}_y, \mathbf{A})$ from $\mathbf{S}$, parse $\mathbf{A}$ as $\{\mathbf{a}_{\ell_y}\}_{\ell_y \in \mathbf{L}_y}$, compute and return, for all $\ell_y \in \mathbf{L}_y$, $\mathbf{v}_y \overset{\mathbf{a}_{\ell_y}}{:=} \mathsf{MM}[\ell_x, \ell_y]$;
- upon receiving $(\mathsf{cid}, \mathsf{ers}, v)$ from a client, for all $\ell \in \mathsf{MM}^{-1}[v]$, compute $\mathsf{MM}[\ell] - v$. Send $\big(\mathsf{cid}, \mathsf{ers}, a, \mathcal{L}(\mathsf{erase}, \mathsf{MM}, v)\big)$ to $\mathcal{S}$.

Figure 1: The ideal 2-dimensional addressable multi-map functionality for snapshot.

protocol and the honest parties and the adversary $\mathcal{A}$ send their outputs and a message to $\mathcal{Z}$, respectively. At the end of this process, $\mathcal{Z}$ outputs a bit b. We denote the random variable corresponding to this bit $\mathbf{Real}_{\mathcal{Z},\mathcal{A}}(k)$.

Ideal-world execution. In the ideal-world execution, every party has access to an ideal functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{DS}}$ that is parameterized with a leakage profile $\mathcal{L}$. The environment $\mathcal{Z}$ takes as input a string $z \in \{0,1\}^*$. Given z, the simulator $\mathcal{S}$ corrupts the server. $\mathcal{Z}$ then adaptively chooses a polynomial number of commands $(\mathsf{comm}_1, \ldots, \mathsf{comm}_m)$ of the above form. For each of these operations: the corresponding party sends the operation to the functionality $\mathcal{F}^{\mathcal{L}_{\mathsf{DS}}}$; the party and the adversary $\mathcal{S}$ then return the ouptut and a message to $\mathcal{Z}$, respectively. At the end of this process, $\mathcal{Z}$ outputs a bit b. We denote the random variable corresponding to this bit $\mathbf{Ideal}^{\mathcal{L}}_{\mathcal{Z},\mathcal{S}}(k)$.

Definition 4.3 (Adaptive security)**.** *We say that Σ_{DS} is $\mathcal{L}$-secure if there exists a* PPT *simulator $\mathcal{S}$ such that for all* PPT *semi-honest adversaries $\mathcal{A}$ making ideal atomic multi-snapshot corruptions, for all* PPT *standalone environments $\mathcal{Z}$, for all $z \in \{0,1\}^*$,*

$$\Big|\Pr\left[\,\mathbf{Real}_{\mathcal{Z},\mathcal{A}}(k) = 1\,\right] - \Pr\left[\,\mathbf{Ideal}^{\mathcal{L}}_{\mathcal{Z},\mathcal{S}}(k) = 1\,\right]\Big| \leq \mathsf{negl}(k).$$

5 Addressable Multi-Maps

Our first building block Σ_M works as follows. First, it is what we refer to as a two-dimensional multi-map encryption scheme which we will explain in more detail below. Second, it achieves statelessness at the cost of correctness in the sense that the values associated to a label can be overwritten. To better capture this behavior, Σ_M is defined as supporting read, write and erase operations instead of get, put and delete operations. We refer to this kind of multi-map as an *addressable multi-map* and define the operations it supports below.

Concurrency via two-dimensionality. The encrypted multi-map EMM_M will be used by Ω to store the tuple associated with a label ℓ. The way we use this structure, however, will result in contention when multiple clients are writing to the same label which will, in turn, slow down Ω's write throughput under parallel put operations.

We address this in the following way. Instead of using a standard encrypted multi-map, EMM_M will be a 2-dimensional encrypted multi-map by which we mean that it holds label/tuple pairs with labels of the form $\boldsymbol{\ell} = (\ell_x, \ell_y)$ with $\ell_x \in \mathbb{L}_x$ and $\ell_y \in \mathbb{L}_y$. Given a high-contention label ℓ, Ω will treat it as a two-dimensional label $\boldsymbol{\ell} = (\ell, u)$, where u is a value sampled uniformly at random from $\{0, \ldots, p\}$, with $p \geq 1$, and store the pair $(\boldsymbol{\ell}, \mathbf{v})$ in EMM_M. The high-level idea is that if n clients try to write to the same high-contention label ℓ then, in expectation, only n/p writes will be executed on the same two-dimensional label $\boldsymbol{\ell} = (\ell, u)$ in EMM_M. Note that a possible optimization here would be to use two-choice allocation instead of just sampling u at random.

To make this idea work in practice, we will need the two-dimensional encrypted multi-map to support—in addition to read, write and erase operations–read operations on a single dimension. To see why, note that under our approach, write operations for the following n label/tuple pairs $(\ell, \mathbf{v}_1), \ldots, (\ell, \mathbf{v}_n)$ for EMM_M will be transformed to n writes of the form $((\ell, u), \mathbf{v}_i)$ for $0 \leq u \leq p$ and $1 \leq i \leq n$. This does not cause any issues during write operations but it does create a problem for reads since a read for ℓ now needs to return the values associated with every two-dimensional label $(\ell, 0), \ldots, (\ell, p)$. Handling this in the naive way would require the client to compute and send p read tokens to the server; one for each $u \in \{0, \ldots, p\}$.

Hyperreads. Instead, we will design our scheme to support two read operations, a standard Read operation and a $\mathsf{HyperRead}$ operation, that work as follows. Read allows the client to retrieve the subset of values associated to a two-dimensional label $\boldsymbol{\ell} \overset{\circ}{=} (\ell_x, \ell_y)$ indexed by a server-provided sequence $\mathbf{a}$. One can think of it essentially as a normal read (as described above) for a label ℓ defined as $\ell_x \| \ell_y$. $\mathsf{HyperRead}$ allows the client to retrieve the subset of values associated to a *set* of labels with x-coordinate ℓ_x indexed by server-provided sequences $\{\mathbf{a}_y\}_{y \in Y}$.

In our case, the $\mathsf{HyperRead}$ protocol will be a single-round protocol where the client message is $O(1)$. Returning to our concurrency problem, when querying for a label ℓ, instead of executing p Read operations, the client and server will execute one $\mathsf{HyperRead}$ operation with client-provided input ℓ and server-provided input $\big(\{0, \ldots, p\}, \{\mathbf{a}_1, \ldots, \mathbf{a}_p\}\big)$.

A note on revealing values in plaintext. Contrary to standard multi-map encryption schemes, Σ_M explicitly reveals the values during Write operations. While this might seem odd at first, the reason we do this will become clearer after describing our database encryption scheme OST. At a high level, the values here will be server-provided document identifiers that are sampled uniformly at random by OST during document insertions. As such, these values will not reveal any information about the database being encrypted.

The plaintext data structure. We are now ready to describe the multi-map structure that Σ_M encrypts. We refer to it as a 2-dimensional addressable multi-map. Its ideal functionality is given in Figure 1 and it supports the following operations:

- write: takes as input a 2-dimensional label $\boldsymbol{\ell}$, a tuple $\mathbf{v}$ and a sequence of unique addresses $\mathbf{a}$ and stores the pair $(\boldsymbol{\ell}, \mathbf{v}')$ such that for all $1 \leq i \leq \#\mathbf{v}$, v_i is stored at index a_i of $\mathbf{v}'$. Note that $\#\mathbf{v}' \geq \#\mathbf{v}$. We sometimes write this as $\mathsf{MM}[\boldsymbol{\ell}] \overset{\mathbf{a}}{:=} \mathbf{v}$.

- read: takes as input a 2-dimensional label $\boldsymbol{\ell}$ and a sequence of addresses $\mathbf{a}$ and returns the values in (ℓ_x, ℓ_y)'s tuple $\mathbf{v}'$ indexed by $\mathbf{a}$. We sometimes write this as $\mathbf{v} \overset{\mathbf{a}}{:=} \mathsf{MM}[\boldsymbol{\ell}]$.

- hyperread: takes as input an x-coordinate ℓ_x, a set of y-coordinates $\mathbf{L}_y = \{\ell_{y,1}, \ldots, \ell_{y,n}\}$ and a set of addresses $\mathbf{A} = \{\mathbf{a}_{\ell_y}\}_{\ell_y \in \mathbf{L}_y}$ and returns, for all $\ell_y \in \mathbf{L}_y$, the values associated with the two-dimensional label $\boldsymbol{\ell}$ indexed by $\mathbf{a}_y$. We sometimes write this as $\mathbf{V} \overset{\mathbf{A}}{:=} \mathsf{MM}[\ell_x, \mathbf{L}_y]$.

- erase: takes as input a value v and removes v from the tuples in which it appears.

Two-party functionality. We note that our 2-dimensional addressable multi-map will be instantiated as a two-party protocol where the addresses in the protocol instantiations of the read, write and hyperread operations will be provided by the server.

5.1 Construction

We now describe Σ_M, our structured encryption scheme for 2-dimensional addressable multi-maps. The construction is described in detail in Figures 2 and 3 and works as follows. It makes use of a pseudo-random function F and of a symmetric encryption scheme SKE. Note that Σ_M is defined as having a Gen algorithm in addition to an Init algorithm. This is for practicality reasons we can avoid storing multiple keys in our OST scheme. Without a separate Gen protocol, OST's Init algorithm would have to make multiple calls to Ω's Init which in turn would make multiple calls to Σ_M's Init and this would mean the client would have to manage a non-trivial number of keys.

Gen. The client samples and outputs a key $K \overset{\$}{\leftarrow} \{0,1\}^k$.

Init. To initialize the structure, Init initializes a dictionary DX and a multi-map MM and generates two keys $K_t := F_K(1)$ and $K_e := F_K(2)$. It then outputs a key $K = (K_t, K_e)$ and sends the empty encrypted multi-map $\mathsf{EMM} := (\mathsf{DX}, \mathsf{MM})$ to the server.

Write. To write a client-provided label $\boldsymbol{\ell}$ with server-provided values $\mathbf{v}$ at server-provided addresses $\mathbf{a}$, the client sends to the server a write token $\mathsf{wtk} := \left(F_{K_t}[\ell_x, \ell_y], F_{K_e}[\ell_x, \ell_y]\right)$. The server parses wtk as $(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e})$ and for all $1 \leq i \leq \#\mathbf{v}$, stores: (1) a pair $(\mathsf{tag}_{a_i}, \mathrm{ct}_i)$ in DX, where $\mathsf{tag}_{a_i} := F_{K_{\boldsymbol{\ell},t}}(a_i)$ and ct_i is an encryption of v_i under key $K_{\boldsymbol{\ell},e}$; and (2) a pair $(v_i, \mathsf{tag}_{a_i})$ in MM.

Read. To read the values associated to a client-provided label $\boldsymbol{\ell} = (\ell_x, \ell_y)$ and indexed by server-provided addresses $\mathbf{a}$, the client sends to the server a read token $\mathsf{rtk} := \left(F_{K_t}[\ell_x, \ell_y], F_{K_e}[\ell_x, \ell_y]\right)$. The server parses rtk as $(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e})$ and for all $1 \leq i \leq \#\mathbf{a}$, recovers the value v_{a_i} by retrieving and decrypting the ciphertext $\mathrm{ct}_i := \mathsf{DX}[\mathsf{tag}_{a_i}]$ using $K_{\boldsymbol{\ell},e}$, where $\mathsf{tag}_{a_i} := F_{K_t}(a_i)$. Finally, it outputs the values $(v_{a_1}, \ldots, v_{a_n})$.

HyperRead. To hyperread the values associated with all 2-dimensional labels with a client-provided x-coordinate ℓ_x and indexed with server-provided addresses $\{\mathbf{a}_{\ell_y}\}$, the client sends to the server a hyperread token $\mathsf{hrtk} := (F_{K_t}(\ell_x), F_{K_e}(\ell_x))$. The server parses hrtk as $(K_{x,t}, K_{x,e})$ and for all $\ell_y \in \mathbf{L}_y$, computes two keys $K_{\boldsymbol{\ell},t} := F_{K_{x,t}}(\ell_y)$ and $K_{\boldsymbol{\ell},e} := F_{K_{x,e}}(\ell_y)$ and, for all $1 \leq i \leq \#\mathbf{a}_{\ell_y}$, recovers the value v_{ℓ_y,a_i} by retrieving and decrypting the ciphertext $\mathsf{ct}_i := \mathsf{DX}[\mathsf{tag}_{\boldsymbol{\ell},t,i}]$ using key $K_{\boldsymbol{\ell},e}$, where $\mathsf{tag}_{\boldsymbol{\ell},t,i} := F_{K_{\boldsymbol{\ell},t}}(\mathbf{a}_{\ell_y}[i])$. Finally, it outputs the values $(v_{\ell_y,a_i})_{\ell_y \in \mathbf{L}_y, i \in [\#\mathbf{a}_{\ell_y}]}$.

Erase. To erase a value v from the multi-map, the client sends an erase token $\mathsf{etk} := v$ to the server. The server computes $(\mathsf{tag}_1, \ldots, \mathsf{tag}_n) := \mathsf{MM}[v]$ and deletes all the pairs associated with tags $(\mathsf{tag}_1, \ldots, \mathsf{tag}_n)$ from DX. Finally, it deletes the label/tuple pair associated with v from MM.

Remark on correctness. Note that since the scheme is addressable, it does not inherently guarantee correctness since tuple values can be overwritten if writes for two different values are made to the same address. In the next section, we will see how to use another scheme to encrypt an auxiliary structure that will provide "overwrite protection" for EMM_M.

Efficiency. Σ_M is optimal with respect to communication complexity: write tokens are $O(\#\mathbf{v})$, read and erase tokens are $O(1)$ and read responses are $O(\#\mathbf{a})$. The scheme is also optimal with respect to server-side computation since writes and reads are $O(\#\mathbf{a})$, hyperreads are $O(\sum_{\ell_y \in \mathbf{L}_y} \#\mathbf{a}_{\ell_y})$ and erase operations are $O(\#\mathsf{MM}^{-1}[v])$. Finally, client-side operations are also optimal since computing write tokens is $O(\#\mathbf{a})$, computing read, hyperread and erase tokens is $O(1)$.

5.2 Security Against Snapshot Adversaries

We now analyze the security of Σ_M in the database-level snapshot model. The multi-snapshot leakage profile of Σ_M is described in Figure 4 and its security is analyzed in the following Theorem.

Theorem 1. *If F is pseudo-random and SKE is RCPA-secure, then Σ_M as described in Figures 2 and 3 is $\mathcal{L}_M$-secure with respect to atomic database-level multi-snapshots.*

Proof. Consider the simulator $\mathcal{S}$ that simulates $\mathcal{A}$'s view as follows:

(simulating after initialization) given leakage $\mathbf{G}_0 := \mathcal{L}_M(\mathsf{init}, \theta)$ set $\mathbf{G} := \mathbf{G}_0$ and parse $\mathbf{G}_0 = (\mathbf{V}_0, \mathbf{E}_0)$ as

$$\mathbf{V}_0 := \big\langle\, 0, \mathsf{init}, \theta \;|\; \bot \,\big\rangle \qquad \text{and} \qquad \mathbf{E}_0 := \emptyset.$$

Initialize two empty dictionaries $\mathsf{DX}^{\mathsf{sim}}$ and $\mathsf{DX} : \{0,1\}^k \to \{0,1\}^\theta$, a multi-map $\mathsf{MM} : \{0,1\}^\theta \to \{0,1\}^{k \times *}$ and output $(\mathsf{DX}, \mathsf{MM})$. $\mathsf{DX}^{\mathsf{sim}}$ will be used by $\mathcal{S}$ to stay consistent.

(simulating the EMM after writes) given $\mathbf{G}_t := \mathcal{L}_M(\mathsf{write}, \boldsymbol{\ell}, \mathbf{v}, \mathbf{a})$ set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$, and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \Big\{ \mathsf{vx}_i \overset{\circ}{=} \big\langle\, t, \mathsf{write}, v_i \;|\; \boldsymbol{\ell}, a_i \,\big\rangle \Big\}_{i \in [m]} \qquad \text{and} \qquad \mathbf{E}_t = \Big\{ \mathsf{vx}_i \times \mathbf{V}(\,\cdot \;|\; \boldsymbol{\ell}, a_i\,) \Big\}_{i \in [m]}.$$

Let $k, \theta \in \mathbb{N}_{\geq 1}$ and $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. Consider the response-revealing stateless addressable two-dimensional multi-map encryption scheme $\Sigma_M = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Write}_{\mathbf{C},\mathbf{S}}, \mathsf{Read}_{\mathbf{C},\mathbf{S}}, \mathsf{HyperRead}_{\mathbf{C},\mathbf{S}}, \mathsf{Erase}_{\mathbf{C},\mathbf{S}})$ with label space $\mathbb{L} = \mathbb{L}_x \times \mathbb{L}_y$ and value space $\mathbb{V} = \{0,1\}^\theta$ defined as follows:

```
• $\mathsf{Gen}_{\mathbf{C}}\big(1^k\big)$:

  Client:
    1. sample and output $K \xleftarrow{\$} \{0,1\}^k$;

• $\mathsf{Init}_{\mathbf{C},\mathbf{S}}\big(K, \theta\big)$:

  Client:
    1. initialize an empty dictionary $\mathsf{DX} : \{0,1\}^k \to \{0,1\}^\theta$;
    2. initialize an empty multi-map $\mathsf{MM} : \{0,1\}^\theta \to \{0,1\}^{k \times *}$;
    3. compute $K_t := F_K(1)$ and $K_e := F_K(2)$;
    4. output $K := (K_t, K_e)$ and send $\mathsf{EMM} := (\mathsf{DX}, \mathsf{MM})$ to the server;

• $\mathsf{Write}_{\mathbf{C},\mathbf{S}}\big(K, \boldsymbol{\ell}; \mathsf{EMM}, \mathbf{v}, \mathbf{a}\big)$:

  Client:
    1. parse $K$ as $(K_t, K_e)$ and $\boldsymbol{\ell}$ as $(\ell_x, \ell_y)$;
    2. compute $K_{\boldsymbol{\ell},t} := F_{K_t}[\ell_x, \ell_y]$;
    3. compute $K_{\boldsymbol{\ell},e} := F_{K_e}[\ell_x, \ell_y]$;
    4. send $\mathsf{wtk} := (K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e})$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{DX}, \mathsf{MM})$ and $\mathsf{wtk}$ as $(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e})$;
    2. for all $1 \leq i \leq \#\mathbf{v}$,
       (a) compute $\mathrm{ct}_i := \mathsf{SKE.Enc}(K_{\boldsymbol{\ell},e}, v_i)$;
       (b) compute $\mathsf{tag}_{a_i} := F_{K_{\boldsymbol{\ell},t}}(a_i)$;
       (c) set $\mathsf{DX}[\mathsf{tag}_{a_i}] := \mathrm{ct}_i$ and $\mathsf{MM}[v_i] \overset{+}{:=} \mathsf{tag}_{a_i}$;
    3. output $\mathsf{EMM} := (\mathsf{DX}, \mathsf{MM})$;

• $\mathsf{Read}_{\mathbf{C},\mathbf{S}}\big(K, \boldsymbol{\ell}; \mathsf{EMM}, \mathbf{a}\big)$:

  Client:
    1. parse $K$ as $(K_t, K_e)$ and $\boldsymbol{\ell}$ as $(\ell_x, \ell_y)$;
    2. compute $K_{\boldsymbol{\ell},t} := F_{K_t}[\ell_x, \ell_y]$;
    3. compute $K_{\boldsymbol{\ell},e} := F_{K_e}[\ell_x, \ell_y]$;
    4. send $\mathsf{rtk} := (K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e})$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{DX}, \mathsf{MM})$ and $\mathsf{rtk}$ as $(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e})$;
    2. initialize an empty sequence $\mathbf{v}$;
    3. for all $1 \leq i \leq \#\mathbf{a}$,
       (a) compute $\mathsf{tag}_{a_i} := F_{K_{\boldsymbol{\ell},t}}(a_i)$;
       (b) compute $\mathrm{ct}_i := \mathsf{DX}[\mathsf{tag}_{a_i}]$;
       (c) compute $v_i := \mathsf{SKE.Dec}(K_{\boldsymbol{\ell},e}, \mathrm{ct}_i)$
       (d) set $\mathbf{v} := (\mathbf{v}, v_i)$;
    4. output $\mathbf{v}$;
```

Figure 2: Σ_M: a 2-dimensional addressable multi-map encryption scheme (part 1).

```
• $\mathsf{HyperRead}_{\mathbf{C},\mathbf{S}}\big(K, \ell_x; \mathsf{EMM}, \mathbf{L}_y, \mathbf{A}\big)$:

  Client:
    1. parse $K$ as $(K_t, K_e)$;
    2. send $\mathsf{hrtk} := (F_{K_t}(\ell_x), F_{K_e}(\ell_x))$ to the server;

  Server:
    1. parse $\mathsf{hrtk}$ as $(K_{x,t}, K_{x,e})$, $\mathbf{A}$ as $\{\mathbf{a}_{\ell_y}\}_{\ell_y \in \mathbf{L}_y}$ and $\mathsf{EMM}$ as $(\mathsf{DX}, \mathsf{MM})$;
    2. initialize an empty sequence $\mathbf{v}$;
    3. for all $\ell_y \in \mathbf{L}_y$,
       (a) compute $K_{\boldsymbol{\ell},t} := F_{K_{x,t}}(\ell_y)$ and compute $K_{\boldsymbol{\ell},e} := F_{K_{x,e}}(\ell_y)$;
       (b) for all $1 \leq i \leq \#\mathbf{a}_{\ell_y}$,
            i. compute $\mathsf{tag}_{\boldsymbol{\ell},t,i} := F_{K_{\boldsymbol{\ell},t}}(\mathbf{a}_{\ell_y}[i])$;
           ii. compute $\mathrm{ct}_i := \mathsf{DX}\big[\mathsf{tag}_{\boldsymbol{\ell},t,i}\big]$;
          iii. compute $K_{\boldsymbol{\ell},e,i} := F_{K_{\boldsymbol{\ell},e}}(\mathbf{a}_{\ell_y}[i])$;
           iv. compute $v_i := \mathsf{SKE.Dec}(K_{\boldsymbol{\ell},e,i}, \mathrm{ct}_i)$;
            v. set $\mathbf{v} := (\mathbf{v}, v_i)$;
    4. output $\mathbf{v}$;

• $\mathsf{Erase}_{\mathbf{C},\mathbf{S}}\big(v; \mathsf{EMM}\big)$:

  Client:
    1. send $\mathsf{etk} := v$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{DX}, \mathsf{MM})$;
    2. compute $(\mathsf{tag}_1, \ldots, \mathsf{tag}_n) := \mathsf{MM}[v]$
    3. for all $1 \leq i \leq n$, compute $\mathsf{DX} - \mathsf{tag}_i$;
    4. compute $\mathsf{MM} - v$;
    5. output $\mathsf{EMM} := (\mathsf{DX}, \mathsf{MM})$;
```

Figure 3: Σ_M: a 2-dimensional addressable multi-map encryption scheme (part 2).

Leakage profile $\mathcal{L}_M$

The leakage profile $\mathcal{L}_M$ is a stateful functionality that constructs a leakage graph $\mathbf{G}$ from any sequence of init, write and erase operations. Given the t^{th} operation op of a sequence, it outputs a leakage sub-graph $\mathbf{G}_t \subseteq \mathbf{G}$ that captures the leakage of op:

```
• $\mathcal{L}_M(\mathsf{op})$:
    1. if $\mathsf{op} = (\mathsf{init}, \theta)$,
        (a) set $t := 0$ and $\mathbf{G} := (\emptyset, \emptyset)$;
        (b) set $\mathbf{V}_t := \langle 0, \mathsf{init}, \theta \mid \perp \rangle$ and $\mathbf{E}_t := \emptyset$;
    2. else if $\mathsf{op} = (\mathsf{write}, \boldsymbol{\ell}, \mathbf{v}, \mathbf{a})$
        (a) compute $\mathbf{t{+}{+}}$;
        (b) set $m := \#\mathbf{v}$;
        (c) for all $1 \leq i \leq m$,
            i. set $\mathsf{vx}_i := \langle t, \mathsf{write}, v_i \mid \boldsymbol{\ell}, a_i \rangle$
            ii. set $\mathbf{E}_i := \mathsf{vx}_i \times \mathbf{V}(\cdot \mid \boldsymbol{\ell}, a_i)$
        (d) set $\mathbf{V}_t := \{\mathsf{vx}_1, \ldots, \mathsf{vx}_m\}$ and $\mathbf{E}_t := \{\mathbf{E}_1, \ldots, \mathbf{E}_m\}$;
    3. else if $\mathsf{op} = (\mathsf{erase}, v)$
        (a) set $\mathbf{V}_t := \langle t{+}{+}, \mathsf{erase}, v \mid \perp \rangle$;
        (b) set $\mathbf{E}_t := \emptyset$
    4. set $\mathbf{G} := \mathbf{G} + (\mathbf{V}_t, \mathbf{E}_t)$
    5. output $(\mathbf{V}_t, \mathbf{E}_t)$;
```

Figure 4: The leakage profile $\mathcal{L}_M$.

Notice that the only modification to DX and MM made by the $\mathsf{Write}_{\mathbf{S}_2}$ algorithm are that: (1) $\#\mathbf{a}$ tag/ciphertext pairs of the form $(\mathsf{tag}_{\boldsymbol{\ell},a_i}, \mathrm{ct}_i)$ are added to DX; and (2) $(v_i, \mathsf{tag}_{a_i})$ is added to MM. Since the values v_i are given and the ciphertexts ct_i are randomized (see Step 3 of $\mathsf{Write}_{\mathbf{C}_1}$), they are straightforward to simulate. The tags, on the other hand, are not given and are deterministic so they could have appeared in previous Write or Erase executions. These correlations, however, are captured by the leakage graph which will help with the simulation.

We consider two cases. The first is when vx_i is a "stale" operation in the sense that its label and address a_i have appeared in a previous write or erase. In this case, vx_i will have incident edges to all the previous operations [3] that wrote to or erased the same label and address. [4] We can use this information to add a tag/ciphertext pair to DX and a value/tag pair to MM with the previously-used tag and a newly simulated ciphertext. More precisely, for all $\mathsf{vx}_i \in \mathbf{V}_t$ with degree at least 1, let $\mathsf{vx}_i^\times$ be the vertex in $\mathbf{N}_\mathbf{G}(\mathsf{vx}_i)$ with the highest rank; compute $(\mathsf{tag}_i^\times, \mathrm{ct}_i^\times) := \mathsf{DX}^{\mathsf{sim}}[\mathsf{vx}_i^\times]$, [5] $\mathrm{ct}_i \xleftarrow{\$} \{0,1\}^{\gamma(\theta))}$ and set $\mathsf{DX}^{\mathsf{sim}}[\mathsf{vx}_i] := (\mathsf{tag}_i^\times, \mathrm{ct}_i)$, $\mathsf{DX}[\mathsf{tag}_i^\times] := \mathrm{ct}_i$ and $\mathsf{MM}[v_i] \overset{+}{:=} \mathsf{tag}_i^\times$.

The second case is when vx_i is "fresh" in the sense that one of $\boldsymbol{\ell}$ or a_i have never appeared in a previous write or erase operation. In this case, we simulate a new tag/ciphertext pair

[3]Since all vertices have a public rank t, we can order them acoording to when they occurred.

[4]Note that here the leakage is slightly more than what is actually leaked. To see why, consider a sequence of operations where we have two erases in a row, e.g., a write, an erase, an erase and then a write. The second erase vertex would have degree 0.

[5]Note that $\mathrm{ct}_i^\times = \perp$ if $\mathsf{vx}_i^\times$ was an erase operation.

Functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{2D\text{-}IDX}}$

The functionality is parameterized with a leakage profile $\mathcal{L}$ and interacts with n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, a server $\mathbf{S}$ and an ideal adversary $\mathcal{S}$. It stores and manages a two-dimensional dictionary DX using the following operations:

- upon receiving $(\mathsf{init}, \mathsf{cid})$ from a client, initialize and store a dictionary DX and send $\big(\mathsf{cid}, \mathsf{init}, \mathcal{L}(\mathsf{init})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{put}, \mathsf{cid}, \boldsymbol{\ell})$ from a client and (put, v) from $\mathbf{S}$, set $\mathsf{DX}[\boldsymbol{\ell}] := v$ and send $\big(\mathsf{cid}, \mathsf{put}, v, \mathcal{L}(\mathsf{put}, \mathsf{DX}, \boldsymbol{\ell}, v)\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{get}, \boldsymbol{\ell})$ from a client return $v := \mathsf{DX}[\boldsymbol{\ell}]$;
- upon receiving $(\mathsf{cid}, \mathsf{hyperget}, \ell_x)$ from a client and $(\mathsf{hyperget}, \mathbf{L}_y)$ from $\mathbf{S}$, for all $\ell_y \in \mathbf{L}_y$, return $v_y := \mathsf{DX}[\boldsymbol{\ell}]$;
- upon receiving $(\mathsf{cid}, \mathsf{comp})$ from a client send $\big(\mathsf{cid}, \mathsf{comp}, \mathcal{L}(\mathsf{comp}, \mathsf{DX})\big)$ to $\mathcal{S}$;

Figure 5: The ideal 2-dimensional immutable dictionary functionality.

and add it to DX. More precisely, for all $\mathsf{vx}_i \in \mathbf{V}_t$ with degree 0, compute $\mathsf{tag}_i \xleftarrow{\$} \{0,1\}^k$ and $\mathrm{ct}_i \xleftarrow{\$} \{0,1\}^{\gamma(\theta))}$, set $\mathsf{DX}^{\mathsf{sim}}[\mathsf{vx}_i] := (\mathsf{tag}_i, \mathrm{ct}_i)$ and $\mathsf{DX}[\mathsf{tag}_i] := \mathrm{ct}_i$, and return DX.

Notice that the pseudo-randomness of F and the RCPA-security of SKE guarantee that the simulated EMM is computationally indistinguishable from a real EMM.

(simulating the EMM after reads and hyperreads) since these operations cause no modifications to the underlying structures, return DX and MM;

(simulating the EMM after erasures) given the leakage $\mathbf{G}_t := \mathcal{L}_M(\mathsf{erase}, \boldsymbol{\ell}, a)$ set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \big\langle\, t, \mathsf{erase}, v \mid \bot \,\big\rangle \qquad \text{and} \qquad \mathbf{E}_t = \emptyset$$

The only modification to DX made by the $\mathsf{Erase}_{\mathbf{S}_2}$ algorithm is the deletion of tag/ciphertext pairs $(\mathsf{tag}_{a_1}, \mathrm{ct}_1), \ldots, (\mathsf{tag}_{a_n}, \mathrm{ct}_n)$ from DX, where $(a_1, \ldots, a_n)$ are the addresses that store v (for any $\boldsymbol{\ell}$). The only modification to MM made by $\mathsf{Erase}_{\mathbf{S}_2}$ is the deletion of v from MM. Note that the tag is deterministic and could have appeared in previous Write or Erase executions. These correlations, however, are captured by the leakage graph.

We first compute $(\mathsf{tag}_{a_1}, \ldots, \mathsf{tag}_{a_n}) := \mathsf{MM}[v]$ and, for all $1 \le i \le n$, compute $\mathsf{DX} - \mathsf{tag}_{a_i}$. Finally, we compute $\mathsf{MM} - v$.

Notice that the simulated EMM is perfectly indistinguishable from a real EMM.

■

6 Immutable Dictionaries

Our second building block Σ_C is a stateless 2-dimensional immutable dictionary encryption scheme that achieves correctness at the cost of limited query functionality and (in some cases) a slight decrease in query efficiency. It is the most complex of our building blocks because it needs to satisfy several non-standard properties which we discuss further.

Overwrite protection. As explained in the Introduction, Σ_M achieves statelessness by giving up on correctness and, specifically, by not providing overwrite protection. To address this limitation Ω will use an auxiliary encrypted structure EDX_C produced with Σ_C to store information that helps prevent overwrites in EMM_M. Σ_C, however, has to be designed in such a way that it is both stateless and correct.

The simplest way to achieve this is to associate a counter $\mathsf{count}_{\boldsymbol{\ell}}$ with every label $\boldsymbol{\ell}$ in the main encrypted multi-map EMM_M, store the pairs $(\boldsymbol{\ell}, \mathsf{count}_{\boldsymbol{\ell}})$ in a dictionary DX, encrypt DX using a response-revealing dictionary encryption scheme and store the resulting encrypted dictionary, EDX_C, with the main encrypted multi-map EMM_M. To add a label/tuple pair $(\boldsymbol{\ell}, \mathbf{v})$ to EMM_M, the client sends encryptions of $\mathbf{v}$ and a Σ_C get token, gtk_C, for $\boldsymbol{\ell}$ so that the server can query EDX_C, recover $\mathsf{count}_{\boldsymbol{\ell}}$ and store the ciphertexts $\mathbf{ct}$ in EMM_M at addresses $\mathbf{a} = (\mathsf{count}_{\boldsymbol{\ell}} + 1, \ldots, \mathsf{count}_{\boldsymbol{\ell}} + \#\mathbf{ct})$. The server then edits the pair $(\boldsymbol{\ell}, \mathsf{count}_{\boldsymbol{\ell}})$ in EDX_C to $(\boldsymbol{\ell}, \mathsf{count}_{\boldsymbol{\ell}} + \#\mathbf{ct})$.

Snapshot security via immutability. While this approach may seem reasonable, it has a subtle security flaw if implemented naively. The problem is with the last step where the server updates EDX_C with the new counter value. If this is done in-place, then a snapshot adversary will be able to correlate EDX_C put operations—and therefore EMM_M write operations—since every put for a label $\boldsymbol{\ell}$ results in changes at a specific location of EDX.[6] To handle this, we design Σ_C so that edits are not in place so that, in turn, correlations are not revealed. One way to do this is to implement the encrypted dictionary using an encrypted multi-map and to implement dictionary-level edit operations with multi-map-level append operations; for example, changing a pair $(\boldsymbol{\ell}, v)$ stored in the encrypted dictionary to $(\boldsymbol{\ell}, v')$ is implemented by appending the new value, v', to $\boldsymbol{\ell}$'s tuple in the underlying encrypted multi-map. A dictionary-level get operation for $\boldsymbol{\ell}$ can then be implemented by returning the last value of $\boldsymbol{\ell}$'s tuple in the encrypted underlying multi-map. We refer to this value as $\boldsymbol{\ell}$'s *tail*. Note that because an dictionary-level edit is implemented as a multi-map-level append, a snapshot adversary cannot correlate between edit operations. Note that in our instantiation of this approach, the encrypted multi-map is built using an underlying plaintext dictionary so, in summary, we build an encrypted dictionary EDX_C using an encrypted multi-map EMM_C which, in turn, uses a plaintext dictionary DX_C. In the following, it will be useful to keep in mind the different layers of this construction.

Efficient immutability via completeness. Recall that any STE scheme we use as a building block for Ω must be stateless; including the encrypted dictionary EDX_C and its underlying encrypted multi-map. This may seem contradictory, however, since the problem we are trying to solve in the first place is to design a stateless encrypted multi-map. Fortunately, the way we use EDX_C's underlying EMM guarantees that the EMM has a special property which allows us to design a stateless scheme that is correct. Specifically, the underlying multi-map will always

[6]Even if the location of the pairs in EDX_C's underlying structures are randomized, there would still be a consistent string associated to the pair that could be used to correlate.

be *complete*, in the sense that for all labels $\boldsymbol{\ell}$, if $\boldsymbol{\ell}$'s tuple $\mathbf{v}$ includes m values then there does not exist an index $1 \leq i \leq m$ such that $v_i = \perp$.

This guarantee of completeness will allow us to support *gettail* operations on the underlying encrypted multi-map efficiently, where the tail of a label/tuple pair is the last element of the label's tuple. More precisely, we do this using the following variant of binary search. Consider a sequence $S = (v_1, \ldots, v_n, \perp_{n+1}, \ldots, \perp_N)$. Given S, we would like to find the address a such that $v_a \neq \perp$ but $v_{a+1} = \perp$. This problem can be solved in $O(N)$ time with linear scanning but also in $O(\log N)$ time as follows: given S, check if the element at address $N/2$ is $\perp$; if so recur on the "left half" of S otherwise recur on the "right half" of S. The base case occurs when the set holds a single element. Note that this algorithm can only work if S is complete. We describe in Section 6.1 below how to execute this algorithm over the scheme's underlying encrypted multi-map EMM_C.

Compaction. Because EDX_C-level edits are implemented using EMM_C-level *appends*, the structure will grow with every edit. Specifically, it will have size $O(\Sigma_{\boldsymbol{\ell} \in \mathsf{DX}} \#\mathsf{puts}_{\boldsymbol{\ell}})$, where $\boldsymbol{\ell} \mathrel{\hat{=}} (\ell_x, \ell_y)$ and $\#\mathsf{puts}_{\boldsymbol{\ell}}$ is the number of put operations for $\boldsymbol{\ell}$ since initialization (note that here put operations overwrite a pair with an existing label so can also be used to edit). To address this, Σ_C includes a compaction operation that works as follows. For every label $\boldsymbol{\ell}$ in the dictionary, it removes the old versions of $\boldsymbol{\ell}$'s value which, at the EMM_C level, translates to deleting all the elements of $\boldsymbol{\ell}$'s tuple except for its tail. This decreases the size of the structure to $O(\#\mathbf{L}_{\mathsf{DX}})$ but introduce a new challenge; specifically with respect to the gettail operation described above. If no compaction has ever occurred, then the binary search occurs over a sequence that starts at a known address $\mathsf{start} := 0$. But if a compaction has occurred, then the binary search needs to start at an unknown address $\mathsf{start} := a+1$, where a is the address of $\boldsymbol{\ell}$'s tail pre-compaction. To handle this problem, the compaction operation will store an *anchor pair* of the form $(\mathsf{anchorflag} \| \alpha, a)$ in EDX_C, where $\mathsf{anchorflag}$ is a flag used to differentiate anchor pairs from value pairs and α is the address of this particular anchor pair. Note that, at every compaction, α will be incremented by 1. Post-compaction a gettail operation will then need to: (1) retrieve the label's latest anchor pair; (2) retrieve the label's tail a from the anchor pair; and (3) set start to that address. To retrieve the label's latest anchor pair, we will perform a binary search similar to the one described above for gettail.

The plaintext data structure. We are now ready to describe the dictionary structure that Σ_C will encrypt. We refer to it as a 2-dimensional immutable dictionary. Its ideal functionality is given in Figure 5 and it supports the following operations:

- put: takes as input a 2-dimensional label $\boldsymbol{\ell}$ and a value v, and stores the pair $(\boldsymbol{\ell}, v)$; even if a pair with label $\boldsymbol{\ell}$ already exists.[7] We sometimes write this as $\mathsf{DX}[\boldsymbol{\ell}] := v$.

- get: takes as input a 2-dimensional label $\boldsymbol{\ell}$ and returns the value associated with $\boldsymbol{\ell}$ or $\perp$ if a pair with label $\boldsymbol{\ell}$ is not stored. We sometimes write this as $v := \mathsf{DX}[\boldsymbol{\ell}]$.

[7]This guarantees that the put operation can also be used to edit pre-existing pairs which is a slight deviation from the traditional put operation.

- $\mathsf{hyperget}$: takes as input an x-coordinate ℓ_x, a set of y-coordinates $\mathbf{L}_y = \{\ell_{y,1}, \ldots, \ell_{y,n}\}$ and returns, for all $\ell_y \in \mathbf{L}_y$, the values associated with the two dimensional label $\boldsymbol{\ell}$. We sometimes write this as $\mathbf{V} := \mathsf{DX}[\ell_x, \mathbf{L}_y]$.
- comp: takes as input a set of two-dimensional labels $\mathbf{L}$ and reduces the size of the dictionary.

6.1 Construction

The scheme is described in detail in Figures 6 and 7 and works as follows. It makes use of a pseudo-random function F and of a symmetric encryption scheme SKE.

Gen. The client samples and outputs a key $K \xleftarrow{\$} \{0,1\}^k$.

Init. To initialize the structure, Init initializes an empty dictionary DX that will represent the encrypted multi-map which, in turn, represents the (append-only) encrypted dictionary. The encrypted dictionary $\mathsf{EDX} := \mathsf{DX}$ is then sent to the server.

Put. To put a client-provided two-dimensional label $\boldsymbol{\ell} = (\ell_x, \ell_y)$ with a server-provided value v, the client sends to the server a put token $\mathsf{ptk} := F_K[\ell_x, \ell_y]$. The server parses ptk as $K_{\boldsymbol{\ell}}$ and uses it to derive two additional keys $K_{\boldsymbol{\ell},t} := F_{K_{\boldsymbol{\ell}}}(1)$ and $K_{\boldsymbol{\ell},e} := F_{K_{\boldsymbol{\ell}}}(2)$ that it will use to generate tags and encrypt v, respectively. More precisely, the server stores the pair $(\mathsf{tag}_{\boldsymbol{\ell},a+1}, \mathrm{ct})$ in DX, where $\mathsf{tag}_{\boldsymbol{\ell},a+1} := F_{K_{\boldsymbol{\ell},t}}(\mathsf{valueflag}\|(a+1))$ and a is the address of $\boldsymbol{\ell}$'s tail and ct is an encryption of $\mathbf{0}^k\|v$. To generate $\mathsf{tag}_{\boldsymbol{\ell},a+1}$, the server uses a subroutine $\mathsf{GetValueAddres}(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e}, \mathsf{DX})$ which returns the address a of $\boldsymbol{\ell}$'s tail and sets $\mathsf{tag}_{\boldsymbol{\ell},a+1} := F_{K_{\boldsymbol{\ell},t}}(\mathsf{valueflag}\|(a+1))$.

Get. To get a client-provided label $\boldsymbol{\ell} = (\ell_x, \ell_y)$, the client sends to the server a get token $\mathsf{gtk} = F_K[\ell_x, \ell_y]$. The server parses gtk as $K_{\boldsymbol{\ell}}$ and uses this key to derive two additional keys $K_{\boldsymbol{\ell},t} := F_{K_{\boldsymbol{\ell}}}(1)$ and $K_{\boldsymbol{\ell},e} := F_{K_{\boldsymbol{\ell}}}(2)$ which it will use as follows. It then uses a subroutine $\mathsf{GetValue}(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e}, \mathsf{DX})$ which returns $\boldsymbol{\ell}$'s tail.

HyperGet. To hyperget the values associated with a client-provided x-coordinate ℓ_x and a set of server-provided y-coordinates $\mathbf{L}_y = \{\ell_{y,1}, \ldots, \ell_{y,n}\}$, the client sends to the server a hyperget token $\mathsf{hgtk} := F_K(\ell_x)$. The server parses hgtk as K_x and, for all $\ell_{y,i} \in \mathbf{L}_y$, generates two keys $K_{\boldsymbol{\ell}_i,t} := F_{K_x}[\ell_{y,i}, 1]$ and $K_{\boldsymbol{\ell}_i,e} := F_{K_x}[\ell_{y,i}, 2]$ which it uses to retrieve a value v_i as in a get operation. Finally, it returns the set of values $\mathbf{v} := (v_1, \ldots, v_n)$.

Compaction. To compact a set of two-dimensional labels $\mathbb{L}_{\mathsf{DX}}$, the client sends to the server a compaction token $\mathsf{ctk} := (\mathsf{ctk}_{\boldsymbol{\ell}})_{\boldsymbol{\ell} \in \mathbb{L}_{\mathsf{DX}}}$ where $\mathsf{ctk}_{\boldsymbol{\ell}} := F_K[\ell_x, \ell_y]$. Each label $\boldsymbol{\ell}$ is compacted in three phases: (1) the server prepares a new anchor for $\boldsymbol{\ell}$; (2) it finds the tags of $\boldsymbol{\ell}$'s values and inserts them in a priority queue PQ; and (3) it dequeues the tags from PQ and deletes the associated pairs from DX.

Note that the server doesn't actually "see" the labels $\boldsymbol{\ell}$, it only receives compaction tokens for these labels which are pseudo-random strings. For notational convenience and ease of understanding we will describe the server's steps using $\overline{\boldsymbol{\ell}}$ and $\overline{\mathbb{L}}_{\mathsf{DX}}$ to denote that it is a "hidden" label in the sense that the server never actually sees the label in plaintext.

To begin, the server parses ctk as $(K_{\overline{\boldsymbol{\ell}}})_{\overline{\boldsymbol{\ell}} \in \overline{\mathbb{L}}_{\mathsf{DX}}}$. Then, for all $\overline{\boldsymbol{\ell}} \in \overline{\mathbb{L}}_{\mathsf{DX}}$, it generates two keys $K_{\overline{\boldsymbol{\ell}},t} := F_{K_{\overline{\boldsymbol{\ell}}}}(1)$ and $K_{\overline{\boldsymbol{\ell}},e} := F_{K_{\overline{\boldsymbol{\ell}}}}(2)$ and does the following. It uses the keys to find $\overline{\boldsymbol{\ell}}$'s tail and address by using the $\mathsf{GetValue}$ and $\mathsf{GetValueAddres}$ subroutines. At this stage, the server constructs a new anchor as follows: (1) it first finds the latest anchor's address by computing $\alpha := \mathsf{BinarySearch}(K_{\overline{\boldsymbol{\ell}},t}, K_{\overline{\boldsymbol{\ell}},e}, \mathsf{DX}, \mathsf{anchorflag}, 0, 2)$ which is a subroutine that is described in more detail below; (2) it prepares a new anchor ciphertext $\mathrm{ct}_{\overline{\boldsymbol{\ell}}} := \mathsf{SKE.Enc}(K_{\overline{\boldsymbol{\ell}},e}, a_{\overline{\boldsymbol{\ell}}} \| v_{\overline{\boldsymbol{\ell}}})$ and stores the pair $(\mathsf{tag}_{\overline{\boldsymbol{\ell}},\alpha+1}, \mathrm{ct}_{\overline{\boldsymbol{\ell}}})$ in DX, where $\mathsf{tag}_{\overline{\boldsymbol{\ell}},\alpha+1} := F_{K_{\overline{\boldsymbol{\ell}},t}}(\mathsf{anchorflag} \| (\alpha+1))$. For the second phase, the server will compute $\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}}$ and test if $\mathsf{DX}[\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}}]$ exists. If this is the case, it enqueues $\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}}$ to PQ, decrements $a_{\overline{\boldsymbol{\ell}}}$ and iterates until either $\mathsf{DX}[\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}}] = \perp$ or $a = 0$. In the third phase, the server simply dequeues each tag from PQ and deletes its associated pair from DX. The purpose of the priority queue PQ is to insure that that the deletion of pairs in DX is done in a random order in case the underlying dictionary is not history independent.

BinarySearch. The purpose of the $\mathsf{BinarySearch}$ subroutine is to retrieve the address of a label' s last anchor or of its tail. More precisely, $\mathsf{BinarySearch}$ takes as input the keys $K_{\boldsymbol{\ell},t}$ and $K_{\boldsymbol{\ell},e}$, as well as the dictionary DX, a flag that can be set to either $\mathsf{anchorflag}$ or $\mathsf{valueflag}$ and two values start and size. The algorithm executes a form of binary search as described above but in order to do so it needs a start address start and an end address end. If we have access to DX's size, we could set $\mathsf{end} := \mathsf{start} + \mathsf{size}(\mathsf{DX})$ but, unfortunately, we only have its approximate size.[8] Nevertheless, we can set end as follows. We first set size to be the approximate size of DX and $\mathsf{end} := \mathsf{start} + \mathsf{size}$ and check if $\mathsf{DX}[\mathsf{tag}_{\mathsf{end}}] \neq \perp$, where $\mathsf{tag}_{\mathsf{end}} := F_{K_{\boldsymbol{\ell},t}}(\mathsf{end})$. If this is the case, then we double size by settings $\mathsf{size} := 2 \cdot \mathsf{size}$, set $\mathsf{end} := \mathsf{start} + \mathsf{size}$ and check if $\mathsf{DX}[\mathsf{tag}_{\mathsf{end}}] \neq \perp$ and so on until the $\mathsf{DX}[\mathsf{tag}_{\mathsf{end}}] = \perp$. At this stage, size is guaranteed to be an upper bound on DX's real size so we can proceed with a binary search between start and end. The core of the algorithm is to compute $\mathsf{tag}_{\mathsf{pivot}} := F_{K_{\boldsymbol{\ell},t}}(\mathsf{flag} \| (\mathsf{start} + \mathsf{median}))$, where $\mathsf{start} + \mathsf{median}$ halfway between start and end, and test if $\mathsf{DX}[\mathsf{tag}_{\mathsf{pivot}}] \neq \perp$. If so then it iterates on the "second half" of the address space; otherwise on the "first half". Notice that prepending the flag to the address when computing the tags allows us to have separate address spaces for anchors and values.

GetValueAddress & GetValue. The purpose of $\mathsf{GetValueAddres}$ and $\mathsf{GetValue}$ are to retrieve a label's tail and the address of the tail, respectively. $\mathsf{GetValueAddres}$ takes as input $K_{\boldsymbol{\ell},t}$, $K_{\boldsymbol{\ell},e}$ and DX and works by using the $\mathsf{BinarySearch}$ subroutine to find the address of $\boldsymbol{\ell}$'s tail. To do this, $\mathsf{BinarySearch}$ needs to be executed with the two keys $K_{\boldsymbol{\ell},t}$ and $K_{\boldsymbol{\ell},e}$, the dictionary DX, the flag $\mathsf{valueflag}$, a start address start and an estimated size size. The main challenge here is for $\mathsf{GetValue}$ to determine the appropriate start address for the binary search because, as discussed above, start has to be set to a value other than 0 if a compaction occurred. This is handled as follows. The algorithm first computes $w := \mathsf{GetAnchor}(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e}, \mathsf{DX})$. If $w = \perp$ then no anchor exists which means that no compaction has occurred and start can be set to 0. If, on the other

[8] While this might be surprising, in the commercial database system our scheme was designed for, it is impossible to efficiently retrieve the precise size of a database.

hand, $w \neq \perp$, then a compaction has occurred and start should be set to the value s stored in w. Next, the algorithm sets size to the maximum of 2 and $\mathsf{approx\_size}(\mathsf{DX})$. The maximum guarantees that the log computation in $\mathsf{BinarySearch}$ will never be computed on 0. Given these values, the algorithm runs the $\mathsf{BinarySearch}$ subroutine to recover the tail's address a. Now there are three cases: (1) if $a = 0$ and $w = \perp$, then the label $\boldsymbol{\ell}$ was never stored in the dictionary so the algorithm returns $\perp$; (2) if $a = 0$ and $w \neq \perp$, then $\boldsymbol{\ell}$ is in the dictionary, a compaction occurred but no put was executed after the compaction in which case the algorithm returns the address a it recovered from the anchor; (3) if $a \neq 0$, then a is returned. The subroutine $\mathsf{GetValue}$ is very similar to $\mathsf{GetValueAddres}$ except that, in addition to finding the tail's address, the tail value itself is returned.

GetAnchor. The purpose of the $\mathsf{GetAnchor}$ subroutine is to retrieve the contents of the latest anchor. To do this, the algorithm makes a call to the $\mathsf{BinarySearch}$ subroutine with flag $\mathsf{anchorflag}$, a start address of 0 and a size of 2. The start address is always 0 because anchors are never deleted and the size is set to 2 as an optimization since the number of anchors is relatively small.

Efficiency. Σ_C is optimal with respect to communication complexity: all tokens are $O(1)$. All its algorithms are also $O(1)$ with the exception of Put and Get which are

$$O\left(\log\left(c \cdot \#\mathbb{L}_{\mathsf{DX}} + \#\mathsf{puts}_c\right)\right),$$

where c is the number of previous compactions and $\#\mathsf{puts}_c$ is the number of the puts since the last compaction (or initialization if no compaction has occurred), and compaction which is

$$O\left(\#\mathsf{puts}_c + \#\mathbb{L}_{\mathsf{DX}} \cdot \log\left(c \cdot \#\mathbb{L}_{\mathsf{DX}} + \#\mathsf{puts}_c\right)\right).$$

The storage overhead of Σ_C is $O\left(c \cdot \#\mathbb{L}_{\mathsf{DX}} + \#\mathsf{puts}_c\right)$.

6.2 Security Against Snapshot Adversaries

We now analyze the security of Σ_C in the database-level snapshot model. The leakage profile of Σ_C is described in Figure 9 and its security is analyzed in the following Theorem.

Theorem 2. *If F is pseudo-random and SKE is RCPA-secure, then Σ_C as described in Figures 6 and 7 is $\mathcal{L}_C$-secure with respect to atomic database-level multi-snapshots.*

Proof. Consider the simulator $\mathcal{S}$ that simulates $\mathcal{A}$'s view as follows:

(simulating after initialization) given leakage $\mathbf{G}_0 := \mathcal{L}_C(\mathsf{init}, \theta)$ set $\mathbf{G} := \mathbf{G}_0$ and parse $\mathbf{G}_0 = (\mathbf{V}_0, \mathbf{E}_0)$ as

$$\mathbf{V}_0 := \left\langle\, 0, \mathsf{init}, \theta \mid \perp \,\right\rangle \qquad \text{and} \qquad \mathbf{E}_0 := \emptyset.$$

Initialize an empty multi-map $\mathsf{MM}^{\mathsf{sim}}$ and an empty dictionary $\mathsf{DX} : \{0,1\}^k \rightarrow \{0,1\}^{\gamma(k+\theta)}$ and output DX; [9]

[9] Recall that our scheme produces encrypted dictionaries that are represented as encrypted multi-maps built on top of an underlying dictionary.

Let $k, \theta \in \mathbb{N}_{\geq 1}$ and $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. Consider the stateless response-revealing two-dimensional dictionary encryption scheme $\Sigma_C = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Put}_{\mathbf{C},\mathbf{S}}, \mathsf{Get}_{\mathbf{C},\mathbf{S}}, \mathsf{HyperGet}_{\mathbf{C},\mathbf{S}}, \mathsf{Compaction}_{\mathbf{C},\mathbf{S}})$ with label space $\mathbb{L} = \mathbb{L}_x \times \mathbb{L}_y$ and value space $\mathbb{V} = \{0,1\}^\theta$ defined as follows:

- $\mathsf{Gen}_{\mathbf{C},\mathbf{S}}\big(1^k, \big)$:

  Client:

  1. sample a key $K \overset{\$}{\leftarrow} \{0,1\}^k$;
  2. output K;

- $\mathsf{Init}_{\mathbf{C},\mathbf{S}}\big(1^k, \theta\big)$:

  Client:

  1. initialize a dictionary $\mathsf{DX} : \{0,1\}^k \to \{0,1\}^{\gamma(k+\theta)}$;
  2. send $\mathsf{EDX} := \mathsf{DX}$ to the server;

- $\mathsf{Put}_{\mathbf{C},\mathbf{S}}\big(K, \boldsymbol{\ell}; \mathsf{EDX}, v\big)$:

  Client:

  1. send $\mathsf{ptk} := F_K[\ell_x, \ell_y]$ to the server;

  Server:

  1. parse EDX as DX and ptk as $K_{\boldsymbol{\ell}}$;
  2. compute $K_{\boldsymbol{\ell},t} := F_{K_{\boldsymbol{\ell}}}(1)$ and $K_{\boldsymbol{\ell},e} := F_{K_{\boldsymbol{\ell}}}(2)$;
  3. compute $\mathrm{ct} := \mathsf{SKE.Enc}(K_{\boldsymbol{\ell},e}, \mathbf{0}^k \| v)$;
  4. compute $a := \mathsf{GetValueAddres}(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e}, \mathsf{DX})$;
  5. compute $\mathsf{tag}_{\boldsymbol{\ell},a+1} := F_{K_{\boldsymbol{\ell},t}}(\mathsf{valueflag} \| (a+1))$;
  6. set $\mathsf{DX}[\mathsf{tag}_{\boldsymbol{\ell},a+1}] := \mathrm{ct}$;
  7. output $\mathsf{EDX} := \mathsf{DX}$;

- $\mathsf{Get}_{\mathbf{C},\mathbf{S}}\big(K, \boldsymbol{\ell}; \mathsf{EDX}\big)$:

  Client:

  1. send $\mathsf{gtk} := F_K[\ell_x, \ell_y]$ to the server;

  Server:

  1. parse EDX as DX and gtk as $K_{\boldsymbol{\ell}}$;
  2. compute $K_{\boldsymbol{\ell},t} := F_{K_{\boldsymbol{\ell}}}(1)$ and $K_{\boldsymbol{\ell},e} := F_{K_{\boldsymbol{\ell}}}(2)$;
  3. compute $v := \mathsf{GetValue}(K_{\boldsymbol{\ell},t}, K_{\boldsymbol{\ell},e}, \mathsf{DX})$;
  4. output v;

Figure 6: Σ_C: a 2-dimensional immutable dictionary encryption scheme (part 1).

```
• $\mathsf{HyperGet}_{\mathbf{C},\mathbf{S}}\big(K, \ell_x; \mathsf{EDX}, \mathbf{L}_y\big)$:
  Client:
    1. send $\mathsf{hgtk} = F_K(\ell_x)$ to the server;
  Server:
    1. parse $\mathsf{EDX}$ as $\mathsf{DX}$ and $\mathsf{hgtk}$ as $K_x$;
    2. parse $\mathbf{L}_y$ as $\{\ell_{y,1}, \ldots, \ell_{y,n}\}$;
    3. for all $1 \leq i \leq n$,
       (a) compute $K_{\boldsymbol{\ell}_i,t} := F_{K_x}[\ell_{y,i}, 1]$ and $K_{\boldsymbol{\ell}_i,e} := F_{K_x}[\ell_{y,i}, 2]$;
       (b) compute $v := \mathsf{GetValue}(K_{\boldsymbol{\ell}_i,t}, K_{\boldsymbol{\ell}_i,e}, \mathsf{DX})$;
       (c) set $\mathbf{v} := (\mathbf{v}, v)$;
    4. output $\mathbf{v}$;

• $\mathsf{Compaction}_{\mathbf{C},\mathbf{S}}\big(K, \mathbb{L}_{\mathsf{DX}}; \mathsf{EDX}\big)$:
  Client:
    1. for all $\boldsymbol{\ell} \in \mathbb{L}_{\mathsf{DX}}$,
       (a) parse $\boldsymbol{\ell}$ as $(\ell_x, \ell_y)$;
       (b) set $\mathsf{ctk}_{\boldsymbol{\ell}} := F_K[\ell_x, \ell_y]$;
    2. send $\mathsf{ctk} := (\mathsf{ctk}_{\boldsymbol{\ell}})_{\boldsymbol{\ell} \in \mathbb{L}_{\mathsf{DX}}}$ to the server;
  Server:
    1. parse $\mathsf{EDX}$ as $\mathsf{DX}$ and $\mathsf{ctk}$ as $(K_{\overline{\boldsymbol{\ell}}})_{\overline{\boldsymbol{\ell}} \in \mathbb{L}_{\mathsf{DX}}}$;
    2. initialize a priority queue $\mathsf{PQ}$;
    3. for all $\overline{\boldsymbol{\ell}} \in \mathbb{L}_{\mathsf{DX}}$,
       (a) compute $K_{\overline{\boldsymbol{\ell}},t} := F_{K_{\overline{\boldsymbol{\ell}}}}(1)$ and $K_{\overline{\boldsymbol{\ell}},e} := F_{K_{\overline{\boldsymbol{\ell}}}}(2)$;
       (b) compute $v_{\overline{\boldsymbol{\ell}}} := \mathsf{GetValue}(K_{\overline{\boldsymbol{\ell}},t}, K_{\overline{\boldsymbol{\ell}},e}, \mathsf{DX})$;
       (c) compute $a_{\overline{\boldsymbol{\ell}}} := \mathsf{GetValueAddres}(K_{\overline{\boldsymbol{\ell}},t}, K_{\overline{\boldsymbol{\ell}},e}, \mathsf{DX})$;
       (d) compute $\alpha_{\overline{\boldsymbol{\ell}}} := \mathsf{BinarySearch}(K_{\overline{\boldsymbol{\ell}},t}, K_{\overline{\boldsymbol{\ell}},e}, \mathsf{DX}, \mathsf{anchorflag}, 0, 2)$;
       (e) compute $\mathrm{ct}_{\overline{\boldsymbol{\ell}}} := \mathsf{SKE.Enc}(K_{\overline{\boldsymbol{\ell}},e}, a_{\overline{\boldsymbol{\ell}}} \| v_{\overline{\boldsymbol{\ell}}})$;
       (f) compute $\mathsf{tag}_{\overline{\boldsymbol{\ell}},\alpha+1} := F_{K_{\overline{\boldsymbol{\ell}},t}}\big(\mathsf{anchorflag} \| (\alpha_{\overline{\boldsymbol{\ell}}} + 1)\big)$;
       (g) set $\mathsf{DX}[\mathsf{tag}_{\overline{\boldsymbol{\ell}},\alpha+1}] := \mathrm{ct}_{\overline{\boldsymbol{\ell}}}$;
       (h) while $a_{\overline{\boldsymbol{\ell}}} > 0$,
           i. compute $\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}} := F_{K_{\overline{\boldsymbol{\ell}},t}}(\mathsf{valueflag} \| a_{\overline{\boldsymbol{\ell}}})$;
           ii. if $\mathsf{DX}[\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}}] \neq \perp$, compute $\mathsf{PQ.enqueue}(\mathsf{tag}_{a_{\overline{\boldsymbol{\ell}}}})$;
           iii. else exit;
           iv. set $a_{\overline{\boldsymbol{\ell}}} := a_{\overline{\boldsymbol{\ell}}} - 1$;
    4. while $\mathsf{PQ.peek} \neq \perp$,
       (a) compute $\mathsf{tag} := \mathsf{PQ.dequeue}$;
       (b) compute $\mathsf{DX} - \mathsf{tag}$;
    5. output $\mathsf{EDX} := \mathsf{DX}$.
```

Figure 7: Σ_C: a 2-dimensional immutable dictionary encryption scheme (part 2).

```
• $\mathsf{GetValue}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX})$:
    1. compute $w := \mathsf{GetAnchor}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX})$;
    2. if $w = \perp$, set $\mathsf{start} := 0$;
    3. else, parse $w$ as $a_s \| s$ and set $\mathsf{start} := a_s$;
    4. set $\mathsf{size} := \max(2, \mathsf{approx\_size}(\mathsf{DX}))$;
    5. compute $a := \mathsf{BinarySearch}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX}, \mathsf{valueflag}, \mathsf{start}, \mathsf{size})$;
    6. if $a = 0$ and $w = \perp$, set $\mathsf{out} := \perp$;
    7. else if $a = 0$ and $w \neq \perp$, set $\mathsf{out} := s$;
    8. else,
        (a) compute $\mathsf{tag}_{\ell,a} := F_{K_{\ell,t}}(\mathsf{valueflag} \| a)$;
        (b) compute $\mathrm{ct}_a := \mathsf{DX}[\mathsf{tag}_{\ell,a}]$;
        (c) compute $\mathbf{0}^k \| v := \mathsf{SKE.Dec}(K_{\ell,e}, \mathrm{ct}_a)$ and set $\mathsf{out} := v$;
    9. output $\mathsf{out}$

• $\mathsf{GetValueAddres}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX})$:
    1. compute $w := \mathsf{GetAnchor}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX})$;
    2. if $w = \perp$, set $\mathsf{start} := 0$;
    3. else, parse $w$ as $a_s \| s$ and set $\mathsf{start} := a_s$;
    4. set $\mathsf{size} := \max(2, \mathsf{approx\_size}(\mathsf{DX}))$;
    5. compute $a := \mathsf{BinarySearch}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX}, \mathsf{valueflag}, \mathsf{start}, \mathsf{size})$;
    6. if $a = 0$ and $w = \perp$, set $\mathsf{out} := 0$;
    7. else if $a = 0$ and $w \neq \perp$, set $\mathsf{out} := a_s$;
    8. else set $\mathsf{out} := a$;
    9. output $\mathsf{out}$;

• $\mathsf{GetAnchor}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX})$:
    1. set $\mathsf{start} := 0$ and $\mathsf{size} := 2$;
    2. compute $\alpha := \mathsf{BinarySearch}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX}, \mathsf{anchorflag}, \mathsf{start}, \mathsf{size})$;
    3. if $\alpha \neq 0$,
        (a) compute $\mathsf{tag}_\alpha := F_{K_{\ell,t}}(\mathsf{anchorflag} \| \alpha)$;
        (b) compute $\mathrm{ct}_\alpha := \mathsf{DX}[\mathsf{tag}_\alpha]$;
        (c) compute $a_s \| v := \mathsf{SKE.Dec}(K_{\ell,e}, \mathrm{ct}_\alpha)$;
        (d) set $\mathsf{out} := a_s \| v$;
    4. else set $\mathsf{out} := \perp$;
    5. output $\mathsf{out}$;

• $\mathsf{BinarySearch}(K_{\ell,t}, K_{\ell,e}, \mathsf{DX}, \mathsf{flag}, \mathsf{start}, \mathsf{size})$:
    1. set $\mathsf{end} := \mathsf{start} + \mathsf{size}$;
    2. compute $\mathsf{tag}_{\mathsf{end}} := F_{K_{\ell,t}}(\mathsf{flag} \| \mathsf{end})$;
    3. while $\mathsf{DX}[\mathsf{tag}_{\mathsf{end}}] \neq \perp$,
        (a) set $\mathsf{size} := 2 \cdot \mathsf{size}$;
        (b) set $\mathsf{end} := \mathsf{start} + \mathsf{size}$;
        (c) compute $\mathsf{tag}_{\mathsf{end}} := F_{K_{\ell,t}}(\mathsf{flag} \| \mathsf{end})$;
    4. set $\mathsf{min} := 1$ and $\mathsf{max} := \mathsf{size}$;
    5. for $1 \leq i \leq \lceil \log \mathsf{size} \rceil$,
        (a) set $\mathsf{median} := \mathsf{min} + \lceil (\mathsf{max} - \mathsf{min})/2 \rceil$;
        (b) compute $\mathsf{tag}_{\mathsf{pivot}} := F_{K_{\ell,t}}(\mathsf{flag} \| (\mathsf{start} + \mathsf{median}))$;
        (c) if $\mathsf{DX}[\mathsf{tag}_{\mathsf{pivot}}] \neq \perp$,
            i. set $\mathsf{min} := \mathsf{median}$;
            ii. if $i = \lceil \log \mathsf{max} \rceil$, set $\mathsf{addr} := \mathsf{start} + \mathsf{median}$;
        (d) else,
            i. set $\mathsf{max} := \mathsf{median}$;
            ii. if $i = \lceil \log \mathsf{size} \rceil$ and $\mathsf{min} = 1$,
                A. if $\mathsf{DX}[F_{K_{\ell,t}}(\mathsf{flag} \| (\mathsf{start} + 1))] \neq \perp$, set $\mathsf{addr} := \mathsf{start} + 1$;
            iii. else if $i := \lceil \log(\mathsf{size}) \rceil$ and $\mathsf{min} \neq 1$, set $\mathsf{addr} := \mathsf{start} + \mathsf{min}$;
    6. output $\mathsf{addr}$;
```

Figure 8: The Σ_C subroutines.

Leakage profile $\mathcal{L}_C$

The leakage profile $\mathcal{L}_C$ is a stateful functionality that constructs a leakage graph $\mathbf{G}$ from any sequence of init, put and comp operations. Given the t^{th} operation op of a sequence, it outputs a leakage sub-graph $\mathbf{G}_t \subseteq \mathbf{G}$ that captures the leakage of op:

```
$\mathcal{L}_C(\mathsf{op})$:
  1. if $\mathsf{op} = (\mathsf{init}, \theta)$,
     (a) set $t := 0$ and $\mathbf{G} := (\emptyset, \emptyset)$;
     (b) initialize an empty set $S$;
     (c) set $\mathbf{V}_t := \langle\, 0, \mathsf{init}, \theta \mid \perp \rangle$;
     (d) set $\mathbf{E}_t := \emptyset$;
  2. else if $\mathsf{op} = (\mathsf{put}, \boldsymbol{\ell}, v)$
     (a) set $\mathbf{L} := \mathbf{L} \cup \boldsymbol{\ell}$;
     (b) set $\mathbf{V}_t := \langle\, t\text{++}, \mathsf{put} \mid \perp \rangle$ and $\mathbf{E}_t := \emptyset$;
  3. else if $\mathsf{op} = \big(\mathsf{comp}, \mathbb{L}_{\mathsf{DX}}\big)$,
     (a) set $\mathbf{V}_t := \langle\, t\text{++}, \mathsf{comp}, \#\mathbf{L} \mid \perp \rangle$ and $\mathbf{E}_t := \emptyset$;
     (b) set $\mathbf{L} := \emptyset$;
  4. set $\mathbf{G} := \mathbf{G} + (\mathbf{V}_t, \mathbf{E}_t)$
  5. output $(\mathbf{V}_t, \mathbf{E}_t)$;
```

Figure 9: The leakage profile $\mathcal{L}_C$.

(simulating EDX after puts) given $\mathbf{G}_t := \mathcal{L}_C(\mathsf{put}, \boldsymbol{\ell}, v)$ set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \big\langle\, t, \mathsf{put} \mid \perp \big\rangle \qquad \text{and} \qquad \mathbf{E}_t = \emptyset.$$

The only modification to DX made by the $\mathsf{Put}_{\mathbf{S}_2}$ algorithm is the addition of a tag/ciphertext pair of the form $(\mathsf{tag}_{a+1}, \mathrm{ct})$. Notice that this change occurs with certainty in the snapshot model since snapshots are only provided after operations—including compactions—complete. The ciphertext ct is randomized (see Step 3 of $\mathsf{Put}_{\mathbf{S}_2}$) and therefore straightforward to simulate. While the tag is deterministic, it is computed by evaluating the PRF on an increasing sequence so tags will never appear more than once in a Put execution.

To simulate, we compute $\mathsf{tag} \overset{\$}{\leftarrow} \{0,1\}^k$ and $\mathrm{ct} \overset{\$}{\leftarrow} \{0,1\}^{\gamma(k+\theta))}$, set $\mathsf{MM}^{\mathsf{sim}}[\mathsf{vx}] := (\mathsf{tag}, \mathrm{ct})$ and $\mathsf{DX}[\mathsf{tag}] := \mathrm{ct}$ and return DX.

Notice that the pseudo-randomness of F and the RCPA-security of SKE guarantee that the simulated EDX is computationally indistinguishable from a real EDX.

(simulating EDX after gets and hypergets) since these operations cause no modifications to the underlying dictionary, return DX;

(simulating the EDX after compactions) given $\mathbf{G}_t := \mathcal{L}_C(\mathsf{comp}, \mathbb{L}_{\mathsf{DX}})$ set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \big\langle\, t, \mathsf{comp}, \#\mathbf{L} \mid \perp \big\rangle \qquad \text{and} \qquad \mathbf{E}_t = \emptyset.$$

$\mathsf{Compaction}_{\mathbf{S}_2}$ makes the following modifications to DX. First, for all $\boldsymbol{\ell} \in \mathbb{L}_{\mathsf{DX}}$, it adds an anchor pair of the form $(\mathsf{tag}_{\overline{\boldsymbol{\ell}}, \alpha+1}, \mathrm{ct})$. The ciphertext is randomized and therefore straightforward to simulate. While the tag is deterministic, it is computed by evaluating

Functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{SET}}$

The functionality is parameterized with a leakage profile $\mathcal{L}$ and interacts with n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, a server $\mathbf{S}$ and an ideal adversary $\mathcal{S}$. It stores and manages a set SET using the following operations:

- upon receiving $(\mathsf{cid}, \mathsf{init})$ from a client, initialize and store a set SET and send $\big(\mathsf{cid}, \mathsf{init}, \mathcal{L}(\mathsf{init})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{ins}, v)$ from a client set $\mathsf{SET} = \mathsf{SET} \cup v$ and send $(\mathsf{cid}, \mathsf{ins}, \mathcal{L}(\mathsf{ins}, \mathsf{SET}, v))$;
- upon receiving $(\mathsf{cid}, \mathsf{enum})$ from a client return $\{v\}_{v \in \mathsf{SET}}$;

Figure 10: The ideal enumerable set functionality.

a PRF on an increasing sequence so tags will never appear more than once. The second modification is that it deletes all pairs with tags in PQ. Note, however, that PQ stores the tags of the values added since the last compaction and this information is provided in the vertices $\mathbf{V}$ of the leakage graph $\mathbf{G}$.

To simulate, for all $1 \leq i \leq \#\mathbf{L}$, compute $\mathsf{tag}_i \overset{\$}{\leftarrow} \{0,1\}^k$ and $\mathrm{ct}_i \overset{\$}{\leftarrow} \{0,1\}^{\gamma(k+\theta))}$ and set $\mathsf{DX}[\mathsf{tag}_i] := \mathrm{ct}_i$. Finally, for all $\mathsf{vx} \in \mathbf{V}(\,\mathsf{put} \mid \cdot\,) \bigcap \mathbf{V}[r+1, t]$, compute $(\mathsf{tag}, \mathrm{ct}) := \mathsf{MM}^{\mathsf{sim}}[\mathsf{vx}]$ and $\mathsf{DX} - (\mathsf{tag}, \mathrm{ct})$, where r is the rank of the latest compaction.

Notice that the pseudo-randomness of F and the RCPA-security of SKE guarantee that the simulated EDX is computationally indistinguishable from a real EDX.

■

7 Enumerable Sets

Σ_S is a set encryption scheme whose ideal functionality is given in Figure 10. It encrypts sets that support the following operations:

- **insert**: takes as input an element and stores it in the set;
- **enum**: enumerates all the elements in the set.

Construction. The scheme Σ_S is simple and described in detail in Figure 11. An encrypted set consists of symmetrically-encrypted elements, an insert token consists of the encryption of the inserted element and enumeration consists of decrypting all the ciphertexts in the encrypted set and listing the plaintexts.

Efficiency. Σ_S is optimal with respect to communication complexity: insert and enumeration tokens are $O(1)$. The scheme is also optimal with respect to server-side computation since inserts are $O(1)$ and enumeration is $O(\#\mathsf{SET})$. Client-side operations are also optimal since computing insert and enumeration tokens are $O(1)$. Finally, the scheme is also optimal with respect to storage complexity since the size of ESET is $O(\#\mathsf{SET})$.

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme. Consider the stateless response-revealing set encryption scheme $\Sigma_S = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Insert}_{\mathbf{C},\mathbf{S}}, \mathsf{Enum}_{\mathbf{C},\mathbf{S}})$ defined as follows:

```
• $\mathsf{Gen}_{\mathbf{C},\mathbf{S}}(1^k)$:
  Client:
    1. sample a key $K \overset{\$}{\leftarrow} \{0,1\}^k$;
    2. output $K$.
• $\mathsf{Init}_{\mathbf{C},\mathbf{S}}(1^k)$:
  Client:
    1. initialize an empty set $\mathsf{SET}$;
    2. output $\mathsf{ESET} := \mathsf{SET}$;
• $\mathsf{Insert}_{\mathbf{C},\mathbf{S}}(K, v; \mathsf{ESET})$:
  Client
    1. compute $\mathrm{ct} := \mathsf{Enc}(K, v)$;
    2. send $\mathsf{itk} := \mathrm{ct}$ to the server;
  Server:
    1. parse $\mathsf{ESET}$ as $\mathsf{SET}$ and $\mathsf{itk}$ as ct;
    2. set $\mathsf{SET} := \mathsf{SET} \cup \mathrm{ct}$;
    3. output $\mathsf{ESET} := \mathsf{SET}$.
• $\mathsf{Enum}_{\mathbf{C},\mathbf{S}}(K; \mathsf{ESET})$:
  Client:
    1. send $\mathsf{etk} := K$ to the server;
  Server:
    1. parse $\mathsf{ESET}$ as $\mathsf{SET}$;
    2. initialize a set $\mathbf{R}$;
    3. for all $\mathrm{ct} \in \mathsf{SET}$,
       (a) compute $v := \mathsf{SKE.Dec}(K, \mathrm{ct})$;
       (b) set $\mathbf{R} := \mathbf{R} \cup v$;
    4. output $\mathbf{R}$.
```

Figure 11: Σ_S: a stateless enumerable encrypted set scheme.

Leakage profile $\mathcal{L}_S$

The leakage profile $\mathcal{L}_S$ is a stateful functionality that constructs a leakage graph $\mathbf{G}$ from any sequence of init and ins operations. Given the t^{th} operation op of a sequence, it outputs a leakage sub-graph $\mathbf{G}_t \subseteq \mathbf{G}$ that captures the leakage of op:

```
• $\mathcal{L}_S(\mathsf{op})$:
    1. if $\mathsf{op} = \mathsf{init}$,
        (a) set $t := 0$ and $\mathbf{G} := (\emptyset, \emptyset)$;
        (b) set $\mathbf{V}_t := \langle\, 0, \mathsf{init} \mid \perp \,\rangle$ and $\mathbf{E}_t := \emptyset$;
    2. else if $\mathsf{op} = (\mathsf{ins}, v)$,
        (a) set $\mathbf{V}_t := \langle\, t{+}{+}, \mathsf{ins}, |v| \mid \perp \,\rangle$ and $\mathbf{E}_t := \emptyset$;
    3. set $\mathbf{G} := \mathbf{G} + (\mathbf{V}_t, \mathbf{E}_t)$;
    4. output $(\mathbf{V}_t, \mathbf{E}_t)$;
```

Figure 12: The leakage profile $\mathcal{L}_S$.

7.1 Security Against Snapshot Adversaries

We now analyze the security of Σ_S. The leakage profile of Σ_P is described in Figure 12 and its security is analyzed in the following Theorem.

Theorem 3. *If* SKE *is RCPA-secure, then* Σ_S *as described in Figure 11 is* $\mathcal{L}_S$*-secure with respect to atomic database-level multi-snapshots.*

Proof. Consider the simulator $\mathcal{S}$ that simulates $\mathcal{A}$'s view as follows:

(simulating after initialization) given leakage $\mathbf{G}_0 := \mathcal{L}_S(\mathsf{init})$ set $\mathbf{G} := \mathbf{G}_0$. Initialize an empty dictionary DX_{st} and an empty set SET and output SET.

(simulating the ESET after insert) given $\mathbf{G}_t := \mathcal{L}_S\big(\mathsf{ins}, v\big)$ set $\mathbf{G} = \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G} = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \big\langle\, t, \mathsf{ins}, |v| \mid \perp \,\big\rangle \qquad \text{and} \qquad \mathbf{E}_t = \emptyset.$$

The only modification to SET made by the $\mathsf{Insert}_{\mathsf{S}_2}$ algorithm is the insertion of a ciphertext ct. To simulate, we compute $\mathrm{ct} \overset{\$}{\leftarrow} \{0,1\}^{\gamma(|v|)}$ and set and return $\mathsf{SET} := \mathsf{SET} \cup \mathrm{ct}$.

Notice that the RCPA-security of SKE guarantees that the simulated ESET is computationally indistinguishable from a real ESET.

(simulating the ESET after enums) since this operation causes no modifications to the underlying dictionary, return SET.

■

8 Testable Multi-Maps

Σ_R is a multi-map encryption scheme whose ideal functionality is given in Figure 13. It encrypts multi-maps that support the following operations:

Functionality $\mathcal{F}_{\mathsf{TMM}}^{\mathcal{L}}$

The functionality is parameterized with a leakage profile $\mathcal{L}$ and interacts with n clients $\mathbf{C}_1, \dots, \mathbf{C}_n$, a server $\mathbf{S}$ and an ideal adversary $\mathcal{S}$. It stores and manages a multi-map MM using the following operations:

- upon receiving $(\mathsf{cid}, \mathsf{init})$ from a client, initialize and store a multi-map MM and send $\big(\mathsf{cid}, \mathsf{init}, \mathcal{L}(\mathsf{init})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{app}, \ell)$ from a client, and (app, v) from $\mathbf{S}$, set $\mathsf{MM}[\ell] \overset{+}{:=} v$ and send $\big(\mathsf{cid}, \mathsf{ins}, v, \mathcal{L}(\mathsf{ins}, \ell, v)\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{tst}, \ell)$ from a client, and (tst, v) from $\mathbf{S}$, return true if $v \in \mathsf{MM}[\ell]$ and false otherwise;
- upon receiving $(\mathsf{cid}, \mathsf{Get}, \ell)$ from a client, return $\mathsf{MM}[\ell]$;
- upon receiving $(\mathsf{cid}, \mathsf{ers}, v)$ from a client, remove v from all tuples in $\mathsf{MM}^{-1}[v]$ and send $\big(\mathsf{cid}, \mathsf{ers}, \mathcal{L}(\mathsf{ers}, v)\big)$ to $\mathcal{S}$;

Figure 13: The ideal testable multi-map functionality.

- **append**: takes as input a label/value pair (ℓ, v) and adds the value v to ℓ's tuple. If ℓ does not exist in the structure it adds the label/tuple pair (ℓ, v).
- **test**: takes as input a label ℓ and a value v and outputs true if ℓ is in the structure and if v is in ℓ's tuple. It outputs false otherwise.
- **get**: takes as input a label ℓ and returns its tuple.
- **erase**: takes as input an element v and removes v from all the tuples in which it is held.

8.1 Construction

The scheme is described in detail in Figures 14 and 15 and works as follows. It makes use of a pseudo-random function F and of a symmetric encryption scheme SKE.

Gen. The client samples and outputs a key $K \overset{\$}{\leftarrow} \{0,1\}^k$.

Init. To initialize the structure, Init initializes an empty set SET that will represent the encrypted set. The encrypted set $\mathsf{EMM} := \mathsf{SET}$ is then sent to the server.

Append. To append a server-provided value v to a client-provided label ℓ, the client computes a key $K_\ell := F_K(\ell)$ that it uses to encrypt a k-bit zero string. More precisely, it computes $\mathrm{ct} := \mathsf{SKE.Enc}(K_\ell, \mathbf{0}^k)$. The client then sends an append token $\mathsf{atk} := \mathrm{ct}$ to the server who adds the pair (ct, v) to SET.

Test. To test if a server-provided value v is in the tuple of a client-provided label ℓ, the client sends a test token $\mathsf{ttk} := F_K(\ell)$ to the server. The server parses EMM as SET and ttk as a key K_ℓ. If there exists a pair $(\mathrm{ct}, v') \in \mathsf{SET}$ such that $v' = v$ and that ct decrypts to $\mathbf{0}^k$ using key K_ℓ, it outputs true. Otherwise it outputs false.

Get. To retrieve the tuple associated to a client-provided label ℓ, the client sends a get token $\mathsf{gtk} := F_K(\ell)$ to the server. The server parses EMM as SET and ttk as a key K_ℓ and instantiates an empty result set $\mathbf{R}$. For all pairs $(\mathrm{ct}, v') \in \mathsf{SET}$, the server adds v' to $\mathbf{R}$ if ct decrypts to $\mathbf{0}^k$ under key K_ℓ. Finally it returns $\mathbf{R}$ to the client.

Erase To erase client-prpvided value v, the client sends an erase token $\mathsf{etk} := v$ to the server. The server removes any pair of the form $(\cdot, v)$ from SET.

Efficiency. Σ_R is optimal with respect to communication complexity: append, test, get and erase tokens are $O(1)$. The server-side computation is $O(1)$ for append and $O(\#\mathsf{MM})$ for test, get and erase. Client-side operations are also optimal since computing append, test, get and erase tokens is $O(1)$. Finally, the scheme is also optimal with respect to storage complexity since the size of EMM is $O(\#\mathsf{MM})$.

8.2 Security Against Snapshot Adversaries

We now analyze the security of Σ_R. The leakage profile of Σ_R is described in Figure 16 and its security is analyzed in the following Theorem.

Theorem 4. *If* SKE *is RCPA-secure, then* Σ_S *as described in Figure 11 is* $\mathcal{L}_R$*-secure with respect to atomic database-level multi-snapshots.*

Proof. Consider the simulator $\mathcal{S}$ that simulates $\mathcal{A}$'s view as follows:

(simulating after initialization) given leakage $\mathbf{G}_0 := \mathcal{L}_R(\mathsf{init})$ set $\mathbf{G} := \mathbf{G}_0$. Initialize an empty set SET and output SET.

(simulating the EMM after appends) given $\mathbf{G}_t := \mathcal{L}_R\big(\mathsf{app}, \ell, v\big)$ set $\mathbf{G} = \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G} = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \big\langle\, t, \mathsf{app}, v \mid \bot \,\big\rangle \qquad \text{and} \qquad \mathbf{E}_t = \emptyset.$$

The only modification to SET made by the $\mathsf{Append}_{\mathbf{S}_2}$ algorithm is the insertion of a pair (ct, v) where v is provided by the leakage. To simulate, compute $\mathrm{ct} \overset{\$}{\leftarrow} \{0,1\}^{\gamma(k)}$ and set and return $\mathsf{SET} := \mathsf{SET} \cup \{(\mathrm{ct}, v)\}$.

The RCPA-security of SKE guarantees that the simulated EMM is computationally indistinguishable from a real EMM.

(simulating the EMM after tests) since this operation causes no modifications to the underlying set, return SET.

(simulating the EMM after gets) since this operation causes no modifications to the underlying set, return SET.

(simulating the EMM after erasures) given the leakage $\mathbf{G}_t := \mathcal{L}_R(\mathsf{erase}, v)$ set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t = \big\langle\, t, \mathsf{erase}, v \mid \bot \,\big\rangle \qquad \text{and} \qquad \mathbf{E}_t = \emptyset.$$

Let $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme. Consider the multi-map encryption scheme $\Sigma_R = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Append}_{\mathbf{C},\mathbf{S}}, \mathsf{Test}_{\mathbf{C},\mathbf{S}}, \mathsf{Get}_{\mathbf{C},\mathbf{S}}, \mathsf{Erase}_{\mathbf{C},\mathbf{S}})$ defined as follows:

```
• $\mathsf{Gen}_{\mathbf{C},\mathbf{S}}\big(1^k\big)$:
  Client:
    1. sample $K \overset{\$}{\leftarrow} \{0,1\}^k$;
    2. output $K$;
• $\mathsf{Init}_{\mathbf{C},\mathbf{S}}\big(1^k\big)$:
  Client:
    1. initialize an empty set $\mathsf{SET}$;
    2. output $\mathsf{EMM} := \mathsf{SET}$;
• $\mathsf{Append}_{\mathbf{C},\mathbf{S}}\big(K, \ell; \mathsf{EMM}, v\big)$:
  Client
    1. compute $K_\ell := F_K(\ell)$;
    2. compute $\mathrm{ct} := \mathsf{Enc}(K_\ell, \mathbf{0}^k)$
    3. send $\mathsf{atk} := \mathrm{ct}$ to the server;
  Server:
    1. parse $\mathsf{EMM}$ as $\mathsf{SET}$ and $\mathsf{atk}$ as ct;
    2. set $\mathsf{SET} := \mathsf{SET} \cup (\mathrm{ct}, y)$;
    3. output $\mathsf{EMM} := \mathsf{SET}$.
• $\mathsf{Test}_{\mathbf{C},\mathbf{S}}\big(K, \ell; \mathsf{EMM}, v\big)$:
  Client:
    1. compute $K_\ell := F_K(\ell)$;
    2. send $\mathsf{ttk} := K_\ell$ to the server;
  Server:
    1. parse $\mathsf{EMM}$ as $\mathsf{SET}$ and $\mathsf{ttk}$ as $K_\ell$;
    2. for all $(\mathrm{ct}, v') \in \mathsf{SET}$,
       (a) compute $m := \mathsf{SKE.Dec}(K_\ell, \mathrm{ct})$;
       (b) if $v' = v$ and $m = \mathbf{0}^k$, output $\mathsf{true}$;
    3. output $\mathsf{false}$;
```

Figure 14: Σ_R: a stateless encrypted set scheme (part 1).

```
• $\mathsf{Get}_{\mathbf{C},\mathbf{S}}(K, \ell; \mathsf{EMM})$:

  Client:
    1. compute $K_\ell := F_K(\ell)$;
    2. send $\mathsf{gtk} := K_\ell$ to server;

  Server:
    1. parse $\mathsf{EMM}$ as $\mathsf{SET}$ and $\mathsf{gtk}$ as $K_\ell$;
    2. initialize a set $\mathbf{R}$;
    3. for all $(\mathrm{ct}, v') \in \mathsf{SET}$,
       (a) compute $m := \mathsf{SKE.Dec}(K_\ell, \mathrm{ct})$;
       (b) if $m = \mathbf{0}^k$, set $\mathbf{R} := \mathbf{R} \cup v'$;
    4. output $\mathbf{R}$;

• $\mathsf{Erase}_{\mathbf{C},\mathbf{S}}(v; \mathsf{EMM})$:

  Client:
    1. send $\mathsf{etk} := v$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $\mathsf{SET}$ and $\mathsf{etk}$ as $v$;
    2. for all $(\mathrm{ct}, v') \in \mathsf{SET}$,
       (a) if $v' = v$, set $\mathbf{R} := \mathbf{R} \setminus (\mathrm{ct}, v')$;
    3. output $\mathsf{EMM} := \mathsf{SET}$.
```

Figure 15: Σ_R: a stateless encrypted set scheme (part 2).

The only modification to SET made by the $\mathsf{Erase}_{\mathbf{S}_2}$ algorithm is the deletion of all pairs of the form $(\cdot, v)$. The value v is provided as part of the leakage so, to simulate, remove all pairs of the form $(\cdot, v)$ from SET.

Notice that the simulated EMM is perfectly indistinguishable from a real EMM.

■

9 A Stateless Multi-Map Encryption Scheme

Ω is our main stateless and concurrent multi-map encryption scheme. Its ideal functionality is described in Figure 17 and its high-level structure was already described in Section 1. It supports the following operations:

- put: takes as input a label ℓ and a tuple $\mathbf{v}$ and adds the pair $(\ell, \mathbf{v})$ to the multi-map if ℓ is not in the multi-map or appends $\mathbf{v}$ to ℓ's existing tuple if ℓ is already in the multi-map;

- get: takes as input a label ℓ and returns the tuple associated with ℓ or $\perp$ if no such label exists;

- count: takes as input a label ℓ and returns the size of the tuple associated with ℓ or 0 if no such label exists;

- test: takes as input a label ℓ and a value v and returns true if v belongs to the tuple associated with ℓ or false otherwise;

Leakage profile $\mathcal{L}_R$

The leakage profile $\mathcal{L}_R$ is a stateful functionality that constructs a leakage graph $\mathbf{G}$ from any sequence of init, ins and erase operations. Given the t^{th} operation op of a sequence, it outputs a leakage sub-graph $\mathbf{G}_t \subseteq \mathbf{G}$ that captures the leakage of op:

```
• $\mathcal{L}_R(\mathsf{op})$:
    1. if $\mathsf{op} = \mathsf{init}$,
        (a) set $t := 0$ and $\mathbf{G} := (\emptyset, \emptyset)$;
        (b) set $\mathbf{V}_t := \langle 0, \mathsf{init} \mid \perp \rangle$ and $\mathbf{E}_t := \emptyset$;
    2. else if $\mathsf{op} = (\mathsf{app}, \ell, v)$,
        (a) set $\mathbf{V}_t := \langle t\text{++}, \mathsf{app}, v \mid \perp \rangle$ and $\mathbf{E}_t := \emptyset$;
    3. else if $\mathsf{op} = (\mathsf{erase}, v)$
        (a) set $\mathbf{V}_t := \langle t\text{++}, \mathsf{erase}, v \mid \perp \rangle$;
        (b) set $\mathbf{E}_t := \emptyset$
    4. set $\mathbf{G} := \mathbf{G} + (\mathbf{V}_t, \mathbf{E}_t)$;
    5. output $(\mathbf{V}_t, \mathbf{E}_t)$;
```

Figure 16: The leakage profile $\mathcal{L}_R$.

- erase: takes as input a value v and removes all tuples that contain the value v from the multi-map;
- compaction: reduces the size of the multi-map.

9.1 Construction

Recall that the scheme makes use of an addressable two-dimensional multi-map encryption scheme Σ_M, an immutable two-dimensional dictionary encryption scheme Σ_C, an enumerable set encryption scheme Σ_S, and a membership set encryption scheme Σ_R. It consists of eight protocols $\Omega = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Put}, \mathsf{Get}, \mathsf{Count}, \mathsf{Test}, \mathsf{Erase}, \mathsf{Compaction})$ which are all described in Figures 18, 19 and 20 and works as follows.

Gen. The client samples and outputs a key $K \overset{\$}{\leftarrow} \{0,1\}^k$.

Init. To initialize the structure, Init first computes four keys K_M, K_C, K_S and K_R such that

$$K_M := F_K[1,1] \quad K_C := F_K[1,2] \quad K_S := F_K[1,3] \quad \text{and} \quad K_R := F_K[1,4]$$

which it then gives as input to the Init protocols of Σ_M, Σ_C, Σ_S and Σ_R, respectively. In particular, the Init protocol of Σ_M and Σ_C takes an additional input θ and k, respectively, where $\theta = \log \#\mathbb{V}$ and k is the security parameter. The output of the init protocols are various encrypted data structures that together compose the encrypted multi-map EMM. More precisely, the Init protocol outputs to the client the set of keys $K := \big(K_M, K_C, K_S, K_R\big)$ and the server receives the encrypted multi-map $\mathsf{EMM} := \big(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R\big)$.

Functionality $\mathcal{F}_{\mathsf{MM}}^{\mathcal{L}}$

The functionality is parameterized with a leakage profile $\mathcal{L}$ and interacts with n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, a server $\mathbf{S}$ and an ideal adversary $\mathcal{S}$. It stores and manages a multi-map MM using the following operations:

- upon receiving $(\mathsf{cid}, \mathsf{init})$ from a client, initialize and store a multi-map MM and send $\big(\mathsf{cid}, \mathsf{init}, \mathcal{L}(\mathsf{init})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{put}, \ell)$ from a client, and $(\mathsf{put}, \mathbf{v})$ from $\mathbf{S}$, set $\mathsf{MM}[\ell] := \mathbf{v}$ and send $\big(\mathsf{cid}, \mathsf{put}, \mathbf{v}, \mathcal{L}(\mathsf{put}, \mathsf{MM}, \ell, \mathbf{v})\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{get}, \ell)$ from a client, return $\mathbf{v} := \mathsf{MM}[\ell]$ to the client;
- upon receiving $(\mathsf{cid}, \mathsf{count}, \ell)$ from a client, return $\mathsf{count} := \#\mathsf{MM}[\ell]$ to the client;
- upon receiving $(\mathsf{cid}, \mathsf{tst}, \ell)$ from a client, and (tst, v) from $\mathbf{S}$, return to the client true if $v \in \mathsf{MM}[\ell]$ and false otherwise;
- upon receiving $(\mathsf{cid}, \mathsf{ers}, v)$ from a client, set $\mathsf{MM} - (\ell, v)$ for all $\ell \in \mathbb{L}_{\mathsf{MM}}$ and send $\big(\mathsf{cid}, \mathsf{ers}, \mathcal{L}(\mathsf{erase}, \mathsf{MM}, v)\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{comp})$ from a client send $\big(\mathsf{cid}, \mathsf{comp}, \mathcal{L}(\mathsf{comp}, \mathsf{MM})\big)$ to $\mathcal{S}$;

Figure 17: The ideal multi-map functionality.

Put. To put a client-provided label ℓ with contention factor p and a server-provided tuple $\mathbf{v}$, the client first samples a partition number $u \overset{\$}{\leftarrow} \{0, \cdots, p\}$. The label ℓ along with the partition number u constitute a two-dimensional label $\boldsymbol{\ell} := (\ell, u)$. The client then computes several tokens as follows. First, it computes a Σ_M write token

$$\mathsf{wtk}_M \leftarrow \Sigma_M.\mathsf{Write}_{\mathbf{C}_1}(K_M, \boldsymbol{\ell}).$$

It then generates Σ_C get, put and compaction tokens

$$\mathsf{gtk}_C \leftarrow \Sigma_C.\mathsf{Get}_{\mathbf{C}_1}(K_C, \boldsymbol{\ell}) \quad \mathsf{ptk}_C \leftarrow \Sigma_C.\mathsf{Put}_{\mathbf{C}_1}(K_C, \boldsymbol{\ell}) \quad \text{and} \quad \mathsf{ctk}_C \leftarrow \Sigma_C.\mathsf{Compaction}_{\mathbf{C}_1}(K_C, \boldsymbol{\ell}).$$

It then computes two insertion tokens

$$\mathsf{atk}_R \leftarrow \Sigma_R.\mathsf{Append}_{\mathbf{C}_1}(K_R, \boldsymbol{\ell}) \quad \text{and} \quad \mathsf{itk}_S \leftarrow \Sigma_S.\mathsf{Insert}_{\mathbf{C}_1}(K_S, \mathsf{gtk}_C \| \mathsf{ctk}_C).$$

The client finally sends to the server a put token $\mathsf{ptk} := (\mathsf{wtk}_M, \mathsf{gtk}_C, \mathsf{ptk}_C, \mathsf{itk}_S, \mathsf{atk}_R)$. Given the put token ptk, the server does the following. First, it retrieves the current counter value count associated with $\boldsymbol{\ell}$ by computing $c \leftarrow \Sigma_C.\mathsf{Get}_{\mathbf{S}_2}(\mathsf{EDX}_C, \mathsf{gtk}_C)$ and setting $\mathsf{count} := c$ if $c \neq \perp$ and setting $\mathsf{count} := 1$ otherwise. Note that the counter value corresponds to the number of values associated with $\boldsymbol{\ell}$ that have ever been added to the encrypted multi-map. Given count and the Σ_M write token wtk_M, the server writes the tuple $\mathbf{v}$ at the addresses $\{\mathsf{count}, \cdots, \mathsf{count} + \#\mathbf{v} - 1\}$ such that

$$\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Write}_{\mathbf{S}_2}(\mathsf{EMM}_M, \mathsf{wtk}_M, \mathbf{v}, \{\mathsf{count}, \ldots, \mathsf{count} + \#\mathbf{v} - 1\}).$$

Given that the server made use of $\#\mathbf{v}$ new addresses, it puts an updated counter in EDX_C by computing $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Put}_{\mathbf{S}_2}(\mathsf{EDX}_C, \mathsf{ptk}_C, \mathsf{count} + \#\mathbf{v})$. The server then updates the encrypted set structures by computing $\mathsf{ESET}_S \leftarrow \Sigma_S.\mathsf{Insert}_{\mathbf{S}_2}(\mathsf{ESET}_S, \mathsf{itk}_S)$ and $\mathsf{EMM}_R \leftarrow \Sigma_R.\mathsf{Append}_{\mathbf{S}_2}(\mathsf{EMM}_R, \mathsf{atk}_R, v)$, for every value $v \in \mathbf{v}$. Finally, it outputs the updated encrypted multi-map $\mathsf{EMM} := \big(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R\big)$.

Let $\Sigma_M = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Write}_{\mathbf{C},\mathbf{S}}, \mathsf{Read}_{\mathbf{C},\mathbf{S}}, \mathsf{HyperRead}_{\mathbf{C},\mathbf{S}}), \Sigma_C = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Put}_{\mathbf{C},\mathbf{S}}, \mathsf{Get}_{\mathbf{C},\mathbf{S}}, \mathsf{HyperGet}_{\mathbf{C},\mathbf{S}}, \mathsf{Compaction}_{\mathbf{C},\mathbf{S}})$, $\Sigma_S = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Insert}_{\mathbf{C},\mathbf{S}}, \mathsf{Enum}_{\mathbf{C},\mathbf{S}})$ and $\Sigma_R = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Append}_{\mathbf{C},\mathbf{S}}, \mathsf{Test}_{\mathbf{C},\mathbf{S}}, \mathsf{Get}_{\mathbf{C},\mathbf{S}})$ be the schemes described in Figures 2, 3 and 6, 7 and 11 and 14 and 15 respectively. Let $p \in \mathbb{N}$ be the maximum number of partitions. Consider the stateless response-hiding multi-map encryption scheme $\Omega = (\mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Put}_{\mathbf{C},\mathbf{S}}, \mathsf{Get}_{\mathbf{C},\mathbf{S}}, \mathsf{Erase}_{\mathbf{C},\mathbf{S}}, \mathsf{Count}_{\mathbf{C},\mathbf{S}}, \mathsf{Test}_{\mathbf{C},\mathbf{S}}, \mathsf{Erase}_{\mathbf{C},\mathbf{S}}, \mathsf{Compaction}_{\mathbf{C},\mathbf{S}})$ with label space $\mathbb{L} = \{0,1\}^\star$ and value space $\mathbb{V} = \{0,1\}^\theta$ defined as follows:

- $\mathsf{Gen}_{\mathbf{C},\mathbf{S}}(1^k)$:

  Client:

  ```
  1. sample $K \xleftarrow{\$} \{0,1\}^k$;
  2. output $K$;
  ```

- $\mathsf{Init}_{\mathbf{C},\mathbf{S}}(K, \lambda, \theta)$:

  Client:

  ```
  1. compute
         $K_M := F_K[1,1] \quad K_C := F_K[1,2] \quad K_S := F_K[1,3] \quad \text{and} \quad K_R := F_K[1,4]$
  2. compute $\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Init}_{\mathbf{C}_1}(K_M, \theta)$;
  3. compute $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Init}_{\mathbf{C}_1}(K_C, \theta)$;
  4. compute $\mathsf{ESET}_S \leftarrow \Sigma_S.\mathsf{Init}_{\mathbf{C}_1}(K_S)$;
  5. compute $\mathsf{EMM}_R \leftarrow \Sigma_R.\mathsf{Init}_{\mathbf{C}_1}(K_R)$;
  6. output $K := (K_M, K_C, K_S, K_R)$ and $\mathsf{EMM} = (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$;
  ```

- $\mathsf{Put}_{\mathbf{C},\mathbf{S}}(K, p, \ell; \mathbf{v}, \mathsf{EMM})$:

  Client:

  ```
  1. parse $K$ as $(K_M, K_C, K_D, K_S)$;
  2. sample $u \xleftarrow{\$} \{0, \cdots, p\}$;
  3. set $\boldsymbol{\ell} \overset{\circ}{=} (\ell, u)$;
  4. compute $\mathsf{wtk}_M \leftarrow \Sigma_M.\mathsf{Write}_{\mathbf{C}_1}(K_M, \boldsymbol{\ell})$;
  5. compute $\mathsf{gtk}_C \leftarrow \Sigma_C.\mathsf{Get}_{\mathbf{C}_1}(K_C, \boldsymbol{\ell})$;
  6. compute $\mathsf{ptk}_C \leftarrow \Sigma_C.\mathsf{Put}_{\mathbf{C}_1}(K_C, \boldsymbol{\ell})$;
  7. compute $\mathsf{ctk}_C \leftarrow \Sigma_C.\mathsf{Compaction}_{\mathbf{C}_1}(K_C, \boldsymbol{\ell})$;
  8. compute $\mathsf{atk}_R \leftarrow \Sigma_R.\mathsf{Append}_{\mathbf{C}_1}(K_R, \boldsymbol{\ell})$;
  9. compute $\mathsf{itk}_S \leftarrow \Sigma_S.\mathsf{Insert}_{\mathbf{C}_1}(K_S, \mathsf{gtk}_C \| \mathsf{ctk}_C)$;
  10. send $\mathsf{ptk} := (\mathsf{wtk}_M, \mathsf{gtk}_C, \mathsf{ptk}_C, \mathsf{itk}_S, \mathsf{atk}_R)$ to the server;
  ```

  Server:

  ```
  1. parse $\mathsf{EMM}$ as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$;
  2. parse $\mathsf{ptk}$ as $(\mathsf{wtk}_M, \mathsf{gtk}_C, \mathsf{ptk}_C, \mathsf{itk}_S, \mathsf{atk}_R)$;
  3. compute $c \leftarrow \Sigma_C.\mathsf{Get}_{\mathbf{S}_2}(\mathsf{EDX}_C, \mathsf{gtk}_C)$;
  4. if $c \neq \perp$ set $\mathsf{count} := c$ else set $\mathsf{count} := 1$;
  5. compute $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Put}_{\mathbf{S}_2}(\mathsf{EDX}_C, \mathsf{ptk}_C, \mathsf{count} + \#\mathbf{v})$;
  6. compute $\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Write}_{\mathbf{S}_2}(\mathsf{EMM}_M, \mathsf{wtk}_M, \mathbf{v}, \{\mathsf{count}, \ldots, \mathsf{count} + \#\mathbf{v} - 1\})$;
  7. for all $0 \leq i \leq \#\mathbf{v} - 1$, set $\mathsf{EMM}_R \leftarrow \Sigma_R.\mathsf{Append}_{\mathbf{S}_2}(\mathsf{EMM}_R, \mathsf{atk}_R, v_i)$;
  8. compute $\mathsf{ESET}_S \leftarrow \Sigma_S.\mathsf{Insert}_{\mathbf{S}_2}(\mathsf{ESET}_S, \mathsf{itk}_S)$;
  9. output $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.
  ```

Figure 18: Ω: a stateless multi-map encryption scheme (part 1).

```
• $\mathsf{Get}_{\mathbf{C},\mathbf{S}}\big(K, p, \ell; \mathsf{EMM}\big)$:

  Client:
    1. parse $K$ as $(K_M, K_C, K_D, K_S, K_R)$;
    2. compute $\mathsf{hrtk}_M \leftarrow \Sigma_M.\mathsf{HyperRead}_{\mathbf{C}_1}(K_M, \ell)$;
    3. compute $\mathsf{hgtk}_C \leftarrow \Sigma_C.\mathsf{HyperGet}_{\mathbf{C}_1}(K_C, \ell)$;
    4. compute $\mathsf{gtk}_R \leftarrow \Sigma_R.\mathsf{Get}_{\mathbf{C}_1}(K_R, \ell)$;
    5. output $\mathsf{gtk} = \big(\mathsf{hrtk}_M, \mathsf{hgtk}_C, \mathsf{gtk}_R, p\big)$;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$ and $\mathsf{gtk}$ as $(\mathsf{hrtk}_M, \mathsf{hgtk}_C, \mathsf{gtk}_R, p)$;
    2. compute $(\mathsf{count}_0, \ldots, \mathsf{count}_p) \leftarrow \Sigma_C.\mathsf{HyperGet}_{\mathbf{S}_2}\,(\mathsf{EDX}_C, \mathsf{hgtk}_C, \{0, \ldots, p\})$;
    3. let $\mathbf{A} \overset{\circ}{=} \big\{\{1, \ldots, \mathsf{count}_u\}\big\}_{u \in \{0,\cdots,p\}}$ and $\mathsf{count} := \sum_{i=0}^{p} \mathsf{count}_i$;
    4. if $\mathsf{count} > \mathtt{limit}$, compute $\mathbf{v} \leftarrow \Sigma_R.\mathsf{Get}_{\mathbf{S}_2}\big(\mathsf{EMM}_R, \mathsf{gtk}_R\big)$;
    5. else, compute $\mathbf{v} \leftarrow \Sigma_M.\mathsf{HyperRead}_{\mathbf{S}_2}\big(\mathsf{EMM}_M, \mathsf{hrtk}_M, \{0, \ldots, p\}, \mathbf{A}\big)$;
    6. output $\mathbf{v}$;

• $\mathsf{Count}_{\mathbf{C},\mathbf{S}}\big(K, p, \ell; \mathsf{EMM}\big)$:

  Client:
    1. parse $K$ as $(K_M, K_C, K_S, K_R)$;
    2. compute $\mathsf{hgtk}_C \leftarrow \Sigma_C.\mathsf{HyperGet}_{\mathbf{C}_1}(K_C, \ell)$;
    3. output $\mathsf{ctk} = \big(\mathsf{hgtk}_C, p\big)$;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$ and $\mathsf{gtk}$ as $(\mathsf{hgtk}_C, p)$;
    2. compute $(\mathsf{count}_0, \ldots, \mathsf{count}_p) \leftarrow \Sigma_C.\mathsf{HyperGet}_{\mathbf{S}_2}\,(\mathsf{EDX}_C, \mathsf{hgtk}_C, \{0, \ldots, p\})$;
    3. compute $\mathsf{count} := \sum_{i=0}^{p} \mathsf{count}_i$;
    4. output $\mathsf{count}$;

• $\mathsf{Test}_{\mathbf{C},\mathbf{S}}\big(K, \ell; \mathsf{EMM}, v\big)$:

  Client:
    1. parse $K$ as $(K_M, K_C, K_S, K_R)$;
    2. compute $\mathsf{ttk}_R \leftarrow \Sigma_R.\mathsf{Test}_{\mathbf{C}_1}(K_R, \ell)$;
    3. send $\mathsf{ttk} := \mathsf{ttk}_R$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$ and $\mathsf{ttk}$ as $\mathsf{ttk}_R$;
    2. compute $b \leftarrow \Sigma_R.\mathsf{Test}_{\mathbf{S}_2}(\mathsf{EMM}_R, v, \mathsf{ttk}_R)$;
    3. output $b$;
```

Figure 19: Ω: a stateless multi-map encryption scheme (part 2).

```
• $\mathsf{Erase}_{\mathbf{C},\mathbf{S}}\big(v; \mathsf{EMM}\big)$:

  Client:
    1. compute $\mathsf{etk}_M \leftarrow \Sigma_M.\mathsf{Erase}_{\mathbf{C}_1}(v)$;
    2. compute $\mathsf{etk}_R \leftarrow \Sigma_R.\mathsf{Erase}_{\mathbf{C}_1}(v)$;
    3. send $\mathsf{etk} := (\mathsf{etk}_M, \mathsf{etk}_R)$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$ and $\mathsf{etk}$ as $(\mathsf{etk}_M, \mathsf{etk}_R)$;
    2. compute $\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_M, \mathsf{etk}_M)$;
    3. compute $\mathsf{EMM}_R \leftarrow \Sigma_R.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_R, \mathsf{etk}_R)$;
    4. output $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.

• $\mathsf{Compaction}_{\mathbf{C},\mathbf{S}}\big(K; \mathsf{EMM}\big)$:

  Client:
    1. parse $K$ as $(K_M, K_C, K_S, K_R)$;
    2. compute $K'_S \leftarrow \Sigma_S.\mathsf{Gen}(1^k)$;
    3. compute $\mathsf{ESET}'_S \leftarrow \Sigma_S.\mathsf{Init}(1^k)$;
    4. compute $\mathsf{etk} \leftarrow \Sigma_S.\mathsf{Enum}_{\mathbf{C}_1}(K_S)$;
    5. set $K := (K_M, K_C, K'_S, K_R)$;
    6. send $\mathsf{ctk} := (\mathsf{etk}, \mathsf{ESET}'_S)$ to the server;

  Server:
    1. parse $\mathsf{EMM}$ as $(\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$;
    2. parse $\mathsf{ctk}$ as $(\mathsf{etk}, \mathsf{ESET}'_S)$ and $\mathsf{etk}$ as $K_S$;
    3. compute $P \leftarrow \Sigma_S.\mathsf{Enum}(K_S, \mathsf{ESET}_S)$;
    4. parse $P$ as $\big(\mathsf{gtk}_{C,i} \| \mathsf{ctk}_{C,i}\big)_{1 \leq i \leq n}$;
    5. set $\mathsf{ctk}_C := (\mathsf{ctk}_{C,i})_{1 \leq i \leq n}$;
    6. compute $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Compaction}_{\mathbf{S}_2}(\mathsf{EDX}_C, \mathsf{ctk}_C)$;
    7. output $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}'_S, \mathsf{EMM}_R)$.
```

Figure 20: Ω: a stateless multi-map encryption scheme (part 3).

Get. To get the tuple associated with label ℓ the client does the following. It computes a Σ_M hyperread token $\mathsf{hrtk}_M \leftarrow \Sigma_M.\mathsf{HyperRead}_{\mathbf{C}_1}(K_M, \ell)$, a Σ_C hyperget token and a Σ_R get token $\mathsf{gtk}_R \leftarrow \Sigma_R.\mathsf{Get}_{\mathbf{C}_1}(K_R, \ell)$ and sends to the server a get token $\mathsf{gtk} := \big(\mathsf{hrtk}_M, \mathsf{hgtk}_C, \mathsf{gtk}_R, p\big)$ where p is the maximum number of partitions. Given gtk, the server first retrieves the counter values count_i from EMM_C associated with the labels $\boldsymbol{\ell}_i := (\ell, i)$, for $0 \leq i \leq p$. It computes $\mathsf{count} := \sum_{i=0}^{p} \mathsf{count}_i$ and if $\mathsf{count} \leq \mathtt{limit}$, it uses the hyperread token hrtk_M to retrieve the values $\mathbf{v}$ at addresses $\mathbf{A}$ from EMM_M by computing $\mathbf{v} \leftarrow \Sigma_M.\mathsf{HyperRead}_{\mathbf{S}_2}\big(\mathsf{EMM}_M, \mathsf{hrtk}_M, \{0, \ldots, p\}, \mathbf{A}\big)$, where

$$\mathbf{A} \overset{\circ}{=} \big\{\{1, \ldots, \mathsf{count}_u\}\big\}_{u \in \{0, \cdots, p\}}.$$

However, if $\mathsf{count} > \mathtt{limit}$, then the server performs a linear membership test over all the values that were ever stored in EMM. In particular, it executes the Σ_R get algorithm $\mathbf{v} \leftarrow \Sigma_R.\mathsf{Get}\mathbf{S}_2\big(\mathsf{EMM}_R, \mathsf{ttk}_R\big)$.

Notice that the Get protocol retrieves the values $\mathbf{v}$ differently depending on whether $\mathsf{count} \leq \mathtt{limit}$ or not. The reason for this is to get around a limitation of the underlying DBMS. More precisely, in our implementation of OST the dictionary DX that underlies the encrypted multi-map EMM_M is built on top of a document database management system (DBMS). How exactly this is done is outside the scope of this work but what is important to note is that the underlying DBMS will return an error if a query response includes more than $\mathtt{limit}$ items. To get around this, Get operates in two modes: if $\mathsf{count} \leq \mathtt{limit}$ it retrieves $\mathbf{v}$ directly from EMM_M; otherwise it retrieves $\mathbf{v}$ from EMM_R. Note that EMM_R's Get algorithm is linear but $\mathtt{limit}$ is very high in practice so the likelihood of doing linear work is low. Furthermore, if count is larger than $\mathtt{limit}$ then it is likely that the optimal get time is close to linear.

Count. To get the count associated with a label ℓ, the client first computes a hyperget token $\mathsf{hgtk}_C \leftarrow \Sigma_C.\mathsf{HyperGet}_{\mathbf{C}_1}(K_C, \ell)$ and then sends to the server a count token $\mathsf{ctk} = \big(\mathsf{hgtk}_C, p\big)$ where p is the contention factor. Given ctk, the server executes the Σ_C get algorithm to retrieve all the counter values $\{\mathsf{count}_0, \cdots, \mathsf{count}_p\}$ associated with the labels $\boldsymbol{\ell}_i := (\ell, i)$ for $i \in \{0, \cdots, p\}$. Finally it returns the sum of all counter values, count, such that $\mathsf{count} := \sum_{i=0}^{p} \mathsf{count}_i$.

Test. To test if the pair composed of a client-provided label ℓ and a server-provided value v exists in the multi-map, the client computes a Σ_R test token $\mathsf{ttk}_R \leftarrow \Sigma_R.\mathsf{Test}_{\mathbf{C}_1}(K_R, \ell)$ which it then sends to the server. Given the test token, the server computes and returns $b \leftarrow \Sigma_R.\mathsf{Test}_{\mathbf{S}_2}(\mathsf{EMM}_R, v, \mathsf{ttk}_R)$.

Erase. To erase a value from the multi-map, the client computes a Σ_M erase token $\mathsf{etk}_M \leftarrow \Sigma_M.\mathsf{Erase}_{\mathbf{C}_1}(v)$ and a Σ_R erase token $\mathsf{etk}_R \leftarrow \Sigma_R.\mathsf{Erase}_{\mathbf{C}_1}(v)$. The client then sends to the server an erase token $\mathsf{etk} := (\mathsf{etk}_M, \mathsf{etk}_R)$. Given etk, the server updates EMM_M and EMM_R by computing the Σ_M and Σ_R erase protocols $\mathsf{EMM}_M \leftarrow \Sigma_M.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_M, \mathsf{etk}_M)$ and $\mathsf{EMM}_R \leftarrow \Sigma_R.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_R, \mathsf{etk}_R)$. The server finally outputs the updated encrypted multi-map $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.

Compaction. To compact the encrypted multi-map EMM, the client first generates a new key K_S and a new encrypted set ESET'_S by running Σ's Gen and Init protocols, respectively. It also computes a Σ_S enumeration token $\mathsf{etk} \leftarrow \Sigma_S.\mathsf{Enum}_{\mathbf{C}_1}(K_S)$ which it then sends to the server along with ESET'_S. The server first parses the enumeration token etk as K_S and executes the Σ_S enumeration algorithm $P \leftarrow \Sigma_S.\mathsf{Enum}(K_S, \mathsf{ESET}_S)$. The set P is composed of all the Σ_C get and compaction tokens that were inserted for every put operation performed against the encrypted multi-map. The server parses P as $\big(\mathsf{gtk}_{C,i} \| \mathsf{ctk}_{C,i}\big)_{1\leq i\leq n}$ and compacts the encrypted dictionary EDX_C by computing $\mathsf{EDX}_C \leftarrow \Sigma_C.\mathsf{Compaction}_{\mathbf{S}_2}(\mathsf{EDX}_C, \mathsf{ctk}_C)$ where $\mathsf{ctk}_C := (\mathsf{ctk}_{C,i})_{1\leq i\leq n}$. Finally, the server outputs the updated encrypted multi-map $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}'_S, \mathsf{EMM}_R)$.

Efficiency. Ω is optimal with respect to communication complexity: the get, count, test, erase and compaction tokens are $O(1)$ whereas the put token is in $O(\#\mathbf{v})$. With respect to server-side computation, the Put protocol is

$$O\bigg(\log\big(c \cdot p \cdot \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c\big) + \#\mathbf{v}\bigg),$$

where c is the number of previous compactions and $\#\mathsf{puts}_c$ is the number of the puts since the last compaction (or initialization if no compaction has occurred). The Get protocol's computation complexity depends on the frequency of the label ℓ, i.e., the number of values associated with ℓ. If the frequency of label ℓ is below `limit`, then the complexity is

$$O\bigg(p \cdot \log\big(c \cdot p \cdot \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c\big) + \#\mathsf{MM}[\ell] + \#\mathsf{del}_\ell\bigg),$$

where $\#\mathsf{del}_\ell$ is the number of values associated to ℓ that were deleted; otherwise it is

$$O\bigg(p \cdot \log\big(c \cdot p \cdot \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c\big) + \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]\bigg).$$

Recall that we are concerned with snapshot security so the fact that the running time is a function of frequency is not a concern. The count complexity is

$$O\bigg(p \cdot \log\big(c \cdot p \cdot \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c\big)\bigg).$$

The test and erase complexity is $O(\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell])$. Finally, the compaction complexity is

$$O\bigg(\#\mathsf{puts}_c + p \cdot \#\mathbb{L}_{\mathsf{MM}} \cdot \log\big(c \cdot p \cdot \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c\big)\bigg).$$

The storage overhead of Ω is

$$O\bigg(c \cdot p \cdot \#\mathbb{L}_{\mathsf{MM}} + \#\mathsf{puts}_c + \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]\bigg).$$

The round of all protocols is 1.

9.2 Security Against Snapshot Adversaries

We now analyze the security of Ω. The leakage profile of Ω is described in Figure 21 and its security is analyzed in the following Theorem.

Theorem 5. *If Σ_M is $\mathcal{L}_M$-secure, Σ_C is $\mathcal{L}_C$-secure, Σ_S is $\mathcal{L}_S$-secure and Σ_R is $\mathcal{L}_R$-secure, then Ω as described in Figures 18, 19 and 20 is $\mathcal{L}_\Omega$-secure with respect to atomic database-level multi-snapshots.*

Proof. Let $\mathcal{S}_M$, $\mathcal{S}_C$, $\mathcal{S}_S$ and $\mathcal{S}_R$ be the simulators guaranteed to exist by the adaptive security of Σ_M, Σ_C, Σ_S and Σ_R, respectively. Consider the simulator $\mathcal{S}$ that simulates $\mathcal{A}$'s view as follows:

(simulating after initialization) given leakage $\mathbf{G}_0 := \mathcal{L}_\Omega(\mathsf{init}, \theta)$ set $\mathbf{G} := \mathbf{G}_0$ and parse $\mathbf{G}_0 = (\mathbf{V}_0, \mathbf{E}_0)$ as

$$\mathbf{V}_0 := \Big\{ \big\langle\, \Omega, 0, \Sigma_M, 0, \mathsf{init}, \theta \mid \perp \big\rangle, \\ \big\langle\, \Omega, 0, \Sigma_C, 0, \mathsf{init}, k \mid \perp \big\rangle, \\ \big\langle\, \Omega, 0, \Sigma_S, 0, \mathsf{init} \mid \perp \big\rangle, \\ \big\langle\, \Omega, 0, \Sigma_R, 0, \mathsf{init} \mid \perp \big\rangle \Big\}$$

and $\mathbf{E}_0 = \emptyset$. Compute

1. $\mathsf{EMM}_M \leftarrow \mathcal{S}_M\big(\big\langle\, \Sigma_M, 0, \mathsf{init}, \theta \mid \perp \big\rangle, \emptyset\big)$,
2. $\mathsf{EDX}_C \leftarrow \mathcal{S}_C\big(\big\langle\, \Sigma_C, 0, \mathsf{init}, k \mid \perp \big\rangle, \emptyset\big)$,
3. $\mathsf{ESET}_S \leftarrow \mathcal{S}_S\big(\big\langle\, \Sigma_S, 0, \mathsf{init} \mid \perp \big\rangle, \emptyset\big)$,
4. $\mathsf{EMM}_R \leftarrow \mathcal{S}_R\big(\big\langle\, \Sigma_R, 0, \mathsf{init} \mid \perp \big\rangle, \emptyset\big)$,

and output $\mathsf{EMM} = (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.

The computational indistinguishability of the simulated EMM from a real EMM follows from a standard hybrid argument and the adaptive security of Σ_M, Σ_C, Σ_S and Σ_R.

(simulating the EMM after puts) given $\mathbf{G}_t := \mathcal{L}_\Omega(\mathsf{put}, p, \ell, \mathbf{v})$, parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t := \Big\{ \big\langle\, \Omega, t, \Sigma_C, t_C, \mathsf{put} \mid \boldsymbol{\ell}, \mathsf{count} + m \,\big\rangle, \\ \big\langle\, \Omega, t, \Sigma_M, t_M, \mathsf{write}, v_i \mid \boldsymbol{\ell}, \mathsf{count} + i \,\big\rangle_{i \in [m]}, \\ \big\langle\, \Omega, t, \Sigma_S, t_S, \mathsf{ins}, 2k \mid \perp \big\rangle, \\ \big\langle\, \Omega, t, \Sigma_R, t_{R,i}, \mathsf{app}, v_i \mid \perp \big\rangle_{i \in [m]} \Big\}$$

and $\mathbf{E}_t := \emptyset$. Compute

1. $\mathsf{EDX}_C \leftarrow \mathcal{S}_C\big(\big\langle\, \Sigma_C, t_C, \mathsf{put} \mid \boldsymbol{\ell}, \mathsf{count} + m \,\big\rangle, \emptyset\big)$,
2. $\mathsf{EMM}_M \leftarrow \mathcal{S}_M\Big(\big\langle\, \Sigma_M, t_M, t, \mathsf{write}, v_i \mid \boldsymbol{\ell}, \mathsf{count} + i \,\big\rangle_{i \in [m]}, \emptyset\Big)$,
3. $\mathsf{ESET}_S \leftarrow \mathcal{S}_S\big(\big\langle\, \Sigma_S, t_S, \mathsf{ins}, 2k \mid \perp \big\rangle, \emptyset\big)$,

4. $\mathsf{EMM}_R \leftarrow \mathcal{S}_S\left(\left\langle\, \Sigma_R, t_R, t, \mathsf{app}, v_i \mid \bot \,\right\rangle_{i\in[m]}, \emptyset\right)$

and output $\mathsf{EMM} = (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.

The computational indistinguishability of the simulated EMM from a real EMM follows from a standard hybrid argument and the adaptive security of Σ_M, Σ_C, Σ_S and Σ_R.

(simulating the EMM after gets) since this operation causes no modifications to the underlying multi-map, return EMM.

(simulating the EMM after counts) since this operation causes no modifications to the underlying multi-map, return EMM.

(simulating the EMM after tests) since this operation causes no modifications to the underlying multi-map, return EMM.

(simulating the EMM after erasures) given leakage $\mathbf{G}_t := \mathcal{L}_\Omega(\mathsf{erase}, v)$, parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t := \left\{\left\langle\, \Omega, t, \Sigma_M, t_M, \mathsf{erase}, v \mid \bot \,\right\rangle, \left\langle\, \Omega, t, \Sigma_R, t_R, \mathsf{erase}, v \mid \bot \,\right\rangle\right\}$$

and $\mathbf{E}_t := \emptyset$.

Compute

1. $\mathsf{EMM}_M \leftarrow \mathcal{S}_M\left(\left\langle\, \Sigma_M, t_M, \mathsf{erase}, v \mid \bot \,\right\rangle, \emptyset\right)$;
2. $\mathsf{EMM}_R \leftarrow \mathcal{S}_R\left(\left\langle\, \Sigma_R, t_R, \mathsf{erase}, v \mid \bot \,\right\rangle, \emptyset\right)$;

and output $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.

The computational indistinguishability of the simulated EMM from a real EMM follows from a standard hybrid argument and the adaptive security of Σ_M, Σ_C, Σ_S and Σ_R.

(simulating the EMM after compactions) given leakage $\mathbf{G}_t := \mathcal{L}_\Omega(\mathsf{comp})$, parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t := \left\{\left\langle\, \Omega, t, \Sigma_S, 0, \mathsf{init} \mid \bot \,\right\rangle, \left\langle\, \Omega, t, \Sigma_C, t_C, \mathsf{comp}, \#P \mid \bot \,\right\rangle\right\}$$

and $\mathbf{E}_t := \emptyset$.

Compute

1. $\mathsf{EDX}_C \leftarrow \mathcal{S}_C\left(\left\langle\, \Sigma_C, t_C, \mathsf{comp}, \#P \mid \bot \,\right\rangle, \emptyset\right)$;
2. $\mathsf{ESET}_S \leftarrow \mathcal{S}_S\left(\left\langle\, \Sigma_S, 0, \mathsf{init} \mid \bot \,\right\rangle, \emptyset\right)$;

and output $\mathsf{EMM} := (\mathsf{EMM}_M, \mathsf{EDX}_C, \mathsf{ESET}_S, \mathsf{EMM}_R)$.

The computational indistinguishability of the simulated EMM from a real EMM follows from a standard hybrid argument and the adaptive security of Σ_M, Σ_C, Σ_S and Σ_R.

■

10 The OST Database Encryption Scheme

We can now describe our final construction, OST which is a stateless document database encryption scheme that makes use of Ω as a building block. Its ideal functionality is described in Figure 22 and we describe it in detail in Figures 23, 24 and 25. In our description of OST below we assume that the set of fields $\mathbf{F}$ contained in the database is fixed and that each field $f \in \mathbf{F}$ has a contention factor $p_f \in \mathbb{N}_{\geq 0}$.

Init. To initialize an encrypted document database Init starts by computing two keys $K_f := F_{K_M}[f, 1]$ and $K'_f := F_{K_M}[f, 2]$ for all fields $f \in \mathbf{F}$. It then initializes an encrypted multi-map for every field $f \in \mathbf{F}$ by computing $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Init}_{\mathbf{C}_1}(K_f, \theta)$. It then initializes an empty document database DDB with fields $\mathbf{F}$ and sets the key $K := (K_f, K'_f)_{f\in\mathbf{F}}$, a state $st := (p_f)_{f\in\mathbf{F}}$ and the encrypted database $\mathsf{EDB} := (\mathsf{DDB}, (\mathsf{EMM}_f)_{f\in\mathbf{F}})$.

Insert. To insert a document $\mathbf{D} = (f, v)_{f\in\mathbf{F}}$, the server will sample a document identifier id uniformly at random and, for all fields $f \in \mathbf{F}$, add id to v's tuple in EMM_f and set $\mathbf{ED}.f$ to an encryption of f's value v, where $\mathbf{ED}$ is an encrypted version of $\mathbf{D}$ prepared and sent by the client. To enable this, the client sends $\mathbf{ED} := (f, \mathrm{ct}_f)_{f\in\mathbf{F}}$, where ct_f is an encryption of f's value and an insert token $\mathsf{itk} := (\mathsf{ptk}_f)_{f\in\mathbf{F}}$, where ptk_f is an Ω put token computed using $\Omega.\mathsf{Put}_{\mathbf{C}_1}(K_f, p_f, v)$. The server uses ptk_f to add id to v's tuple in EMM_f and updates the encrypted document $\mathbf{ED}$ by setting the $\mathsf{\_id}$ field to id. Finally, the server outputs the updated encrypted document database EDB.

Find. To find the documents that match filter, the client does the following. If filter is an exact match filter, i.e., has the form $f = v$, it computes an Ω get token $\mathsf{gtk} \leftarrow \Omega.\mathsf{Get}_{\mathbf{C}_1}(K_f, p_f, v)$ and sends a find token $\mathsf{ftk} := (\mathsf{exactflag}, \mathsf{gtk})$ to the server. If, on the other hand, filter is a conjunctive filter, i.e., has the form $f_1 = v_1 \bigwedge \cdots \bigwedge f_m = v_m$, it computes, for each clause $f_i = v_i$, three tokens $\mathsf{gtk}_i \leftarrow \Omega.\mathsf{Get}_{\mathbf{C}_1}(K_{f_i}), v_i)$, $\mathsf{ttk}_i \leftarrow \Omega.\mathsf{Test}_{\mathbf{C}_1}(K_f, v_i)$ and $\mathsf{cttk}_i \leftarrow \Omega.\mathsf{count}_{\mathbf{C}_1}(K_{f_i}, v_i)$ and sends $\mathsf{ftk} := (\mathsf{conjflag}, \mathsf{cjtk})$, where $\mathsf{cjtk} := (f_i, \mathsf{gtk}_i, \mathsf{ttk}_i, \mathsf{cttk}_i)_{1\leq i\leq m}$ to the server.

Given ftk, the server parses it as $(\mathsf{flag}, \mathsf{tk})$. If the find query is for an exact filter, i.e., if $\mathsf{flag} = \mathsf{exactflag}$, the server parses tk as gtk and recovers the IDs of the documents that match the filter by computing $\mathbf{ids} \leftarrow \Omega.\mathsf{Get}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{gtk})$. It then retrieves these documents from DDB and returns them to the client. If, on the other hand, $\mathsf{filter} = \mathsf{conjfilter}$ the server parses tk as $(f_i, \mathsf{gtk}_i, \mathsf{ttk}_i, \mathsf{cttk}_i)_{1\leq i\leq m}$ and, for all $1 \leq i \leq m$, computes the frequencies of each field f_i by computing $\mathsf{count}_i \leftarrow \Omega.\mathsf{Counts}_{\mathbf{S}_2}(\mathsf{EMM}_{f_i}, \mathsf{cttk}_i)$. If some $\mathsf{count}_i = 0$, then returns $\emptyset$, otherwise it finds the field $f^\star$ with the smallest frequency and recovers the IDs of the documents that match $f^\star$'s clause by computing $\mathbf{ids} \leftarrow \Omega.\mathsf{Get}_{\mathbf{S}_2}(\mathsf{EMM}_{f^\star}, \mathsf{gtk}_{f^\star})$. Then, for all $\mathsf{id} \in \mathbf{ids}$, it tests if id is in the result set of the non-$f^\star$ clauses by computing $b \leftarrow \Omega.\mathsf{Test}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{ttk}_f, \mathsf{id})$. If this is not the case, i.e., if $b = 0$, then id is removed from $\mathbf{ids}$. The documents with document IDs in $\mathbf{ids}$ are then retrieved from DDB and returned to the client who decrypts the ciphertexts associated to each field f using the key K'_f.

DeleteOne. To delete one document that matches a filter filter, the client sends an Ω find token $\mathsf{gtk} \leftarrow \Omega.\mathsf{Find}_{\mathbf{C}_1}(K, \mathsf{filter})$ to the server who uses it to recover the IDs of the documents that match the filter by computing $\mathbf{ids} \leftarrow \Omega.\mathsf{Find}_{\mathbf{S}_2}(\mathsf{EDB}, \mathsf{ftk})$. The server then selects a document ID uniformly at random from $\mathbf{ids}$ such that $\mathsf{id} \overset{\$}{\leftarrow} \mathbf{ids}$. Finally, the server erases id from all the encrypted multi-maps; that is, for all $f \in \mathbf{F}$, it computes $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{id})$.

UpdateOne. To update one document that matches a filter filter with an action $\mathsf{action} = (f, v)$, the server will need to: (1) pick a document uniformly at random from the set of documents that match the filter; (2) replace the content of field f with an encryption of the new value v; (3) remove id from EMM_f; and (4) put id in v's tuple in EMM_f. To enable this, the client sends to the server an update token

$$\mathsf{utk} := (\mathsf{ftk}, f, \mathrm{ct}_f, \mathsf{ptk}_f),$$

where $\mathsf{ftk} \leftarrow \mathsf{OST}.\mathsf{Find}_{\mathbf{S}_s}(\mathsf{EDB}, \mathsf{filter})$ is an OST find token for filter, $\mathrm{ct}_f := \mathsf{SKE}.\mathsf{Enc}(K'_f, v)$ is an encryption of v, and $\mathsf{ptk} \leftarrow \Omega.\mathsf{Put}_{\mathbf{C}_1}(K_f, p_f, v)$ is an Ω put token for v. The server uses the find token ftk to recover the IDs of the documents that match the filter by computing $\mathbf{ids} \leftarrow \Omega.\mathsf{Find}_{\mathbf{S}_2}(\mathsf{EDB}, \mathsf{ftk})$. The server then selects a document ID uniformly at random from $\mathbf{ids}$ such that $\mathsf{id} \overset{\$}{\leftarrow} \mathbf{ids}$, and it performs the following. It updates the content of the field f of document $\mathsf{DDB}[\_\mathsf{id} = \mathsf{id}]$ with the new encrypted value ct_f; it erases id from the EMM_f by computing $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{id})$; and puts the new value in EMM_f by computing $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Put}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{ptk}_f, \mathsf{id})$. Finally, the server outputs the updated encrypted document database EDB.

Compaction. To compact, the client sends an Ω compaction token for every field $f \in \mathbf{F}$. That is, it sends $\mathsf{ctk} := (\mathsf{ctk}_f)_{f \in \mathbf{F}}$, where $\mathsf{ctk}_f \leftarrow \Omega.\mathsf{Compaction}_{\mathbf{C}_1}(K_f)$. Given ctk, the server simply compacts the encrypted multi-map EMM_f of every field.

Efficiency. In the following efficiency analysis, we will make the simplification assumption that all operations performed against the underlying document database are constant time. In particular, the operations include: (1) updating the value of a field, (2) inserting a document into the database, (3) retrieving one document using the document ID, and (4) deleting a document based on its document ID.[10]

All the protocols of OST are non-interactive and therefore have constant round complexity. In terms of communication complexity, OST is optimal as the Insert is $O(\#\mathbf{D})$, Find is $O(\#\mathsf{filter} + \#\mathsf{DDB}[\mathsf{filter}])$, $\mathsf{UpdateOne}$ is $O(1)$, $\mathsf{DeleteOne}$ is $O(1)$, and $\mathsf{Compaction}$ is $O(1)$. The storage overhead of OST is equal to $O\big(\#\mathsf{DDB} + \mathsf{aux}\big)$ where

$$\mathsf{aux} := \sum_{f \in \mathbf{F}} \big(c \cdot p_f \cdot \#\mathbb{S}_f + \#\mathsf{InsUp}_{c,f}\big),$$

and $\mathbb{S}_f$ is the support of field f and $\mathsf{InsUp}_{c,f}$ is the number of inserts and updates since the last compaction (or initialization if no compaction has occurred).In terms of server-side computation,

[10]Note that in practice, these operations are not constant time as they depend on the underlying plaintext data structure. The plaintext database we are considering makes use of B-trees as the underlying data structure and all of these operations will take a logarithmic time in the size of the structure. The reason of removing this cost is to better focus on the overhead incurred by OST only.

the Insert protocol takes

$$O\bigg(\#\mathbf{F}\cdot\log(\mathsf{aux})\bigg),$$

The Find, $\mathsf{DeleteOne}$ and $\mathsf{UpdateOne}$ have the same computation complexity so we only describe the cost of finds. In the case of an exact search, i.e., $\mathsf{filter}\equiv f=v$, and when the frequency of the value v is smaller than `limit`, then the computation complexity of a Find is equal to

$$O\bigg(p\cdot\log(\mathsf{aux})+\#\mathsf{DDB}[f=v]+\#\mathsf{delUp}_{f,v}\bigg),$$

where $\#\mathsf{delUp}_{f,v}$ is the number of times the field/value (f,v) has been deleted or updated; otherwise it is equal to

$$O\bigg(p\cdot\log(\mathsf{aux})+\#\mathsf{DDB}\bigg).$$

In the case of a conjunction, i.e., $\mathsf{filter}\equiv f_1=v_1\bigwedge\cdots\bigwedge f_m=v_m$, and when the smallest frequency is smaller then `limit`, then the computation complexity is equal to

$$O\bigg(m\cdot\#\mathsf{DDB}[f^\star=v]+\#\mathsf{delUp}_{f^\star,v}+\sum_{i=1}^{m}p_{f_i}\cdot\log(\mathsf{aux})\bigg),$$

where $f^\star$ is the field in the conjunction filter with the smallest frequency; otherwise it is equal to

$$O\bigg(m\cdot\#\mathsf{DDB}+\sum_{i=1}^{m}p_{f_i}\cdot\log(\mathsf{aux})\bigg).$$

10.1 Security against Snapshot Adversaries

We now analyze the security of OST. The leakage profile of OST is described in Figure 26 adn its security is analyzed in the following Theorem.

Theorem 6. *If Ω is $\mathcal{L}_\Omega$-secure and SKE is CPA-secure, then OST as described in Figures 23, 24 and 25 is $\mathcal{L}_{\mathsf{OST}}$-secure with respect to atomic database-level multi-snapshots.*

Proof. Let $\mathcal{S}_\Omega$ and $\mathcal{S}_{\mathsf{SKE}}$ be the simulators guaranteed to exist by the $\mathcal{L}_\Omega$-security of Ω and the CPA-security of SKE. Consider the simulator $\mathcal{S}_{\mathsf{OST}}$ that simualtes $\mathcal{A}$'s view as follows:

(simulating after initialization) given leakage $\mathbf{G}_0:=\mathcal{L}_{\mathsf{OST}}(\mathsf{init},\theta,\mathbf{F},\mathbf{P})$ set $\mathbf{G}:=\mathbf{G}_0$ and parse $\mathbf{G}_0=(\mathbf{V}_0,\mathbf{E}_0)$ as

$$\begin{aligned}\mathbf{V}_0=\Bigg\{\bigg\{&\big\langle\,\mathsf{OST},0,\Omega,0,\Sigma_M,0,\mathsf{init},\theta,f\mid\bot\,\big\rangle,\\ &\big\langle\,\mathsf{OST},0,\Omega,0,\Sigma_C,0,\mathsf{init},\theta,f\mid\bot\,\big\rangle,\\ &\big\langle\,\mathsf{OST},0,\Omega,0,\Sigma_S,0,\mathsf{init},f\mid\bot\,\big\rangle,\\ &\big\langle\,\mathsf{OST},0,\Omega,0,\Sigma_R,0,\mathsf{init},f\mid\bot\,\big\rangle\bigg\}_{f\in\mathbf{F}},\\ &\big\langle\,\mathsf{OST},0,\Omega,0,\Delta,0,\mathsf{init},\mathbf{F},\mathbf{P}\mid\bot\,\big\rangle\Bigg\}\end{aligned}$$

and $\mathbf{E}_0 := \emptyset$;

For all $f \in \mathbf{F}$, compute

$$\mathsf{EMM}_f \leftarrow \mathcal{S}_\Omega\bigg(\Big\{\big\langle\, \Omega, 0, \Sigma_M, 0, \mathsf{init}, \theta, f \mid \bot \,\big\rangle, \big\langle\, \Omega, 0, \Sigma_C, 0, \mathsf{init}, \theta, f \mid \bot \,\big\rangle, \big\langle\, \Omega, 0, \Sigma_S, 0, \mathsf{init}, f \mid \bot \,\big\rangle, \big\langle\, \Omega, 0, \Sigma_R, 0, \mathsf{init}, f \mid \bot \,\big\rangle\Big\}, \emptyset\bigg).$$

Initialize an empty document database DDB with fields $\mathbf{F}$ and output $\mathsf{EDB} := \big(\mathsf{DDB}, (\mathsf{EMM}_f)_f\big)$.

The computational indistinguishability of the simulated EMM_f from a real EMM_f, for all $f \in \mathbf{F}$, follows from a standard hybrid argument and the adaptive security of Ω.

(simulating after inserts) given $\mathbf{G}_t := \mathcal{L}_{\mathsf{OST}}(\mathsf{insert}, \mathbf{D})$, set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t := \Bigg\{\bigg\{\big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{put}, f \mid \bot \,\big\rangle, \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{write}, \mathsf{id}, f \mid \bot \,\big\rangle, \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{ins}, 2k, f \mid \bot \,\big\rangle, \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{app}, \mathsf{id}, f \mid \bot \,\big\rangle\bigg\}_{f \in \mathbf{F}}, \big\langle\, \mathsf{OST}, t, \Delta, t_\Delta, \mathsf{insert}, |v_f|, f \mid \bot \,\big\rangle_{f \in \mathbf{F}}\Bigg\}$$

and $\mathbf{E}_t = \emptyset$.

For all $f \in \mathbf{F}$, compute

$$\mathsf{EMM}_f \leftarrow \mathcal{S}_\Omega\bigg(\Big\{\big\langle\, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{put}, f \mid \bot \,\big\rangle, \big\langle\, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{write}, \mathsf{id}, f \mid \bot \,\big\rangle, \big\langle\, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{ins}, 2k, f \mid \bot \,\big\rangle, \big\langle\, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{app}, \mathsf{id}, f \mid \bot \,\big\rangle\Big\}, \emptyset\bigg).$$

To simulate an encrypted document, we perform the following steps. First, initialize an empty document $\mathbf{D}$ and sample uniformly at random a document ID, id, from $\{0,1\}^k$. For every $f \in \mathbf{F}$, compute $\mathrm{ct}_f \leftarrow \mathcal{S}_{\mathsf{SKE}}(1^k, |v_f|)$ and set $\mathbf{D}.f := \mathrm{ct}_f$. Finally, add $\mathbf{D}$ to DDB and output $\mathsf{EDB} := \big(\mathsf{DDB}, (\mathsf{EMM}_f)_f\big)$.

The computational indistinguishability of the simulated EMM_f from a real EMM_f, for all $f \in \mathbf{F}$, follows from a standard hybrid argument and the adaptive security of Ω. Moreover, the computational indistinguishability of the simulated document $\mathbf{D}$ from a real document, follows from a standard hybrid argument and the CPA-security of SKE.

(simulating after finds) since this operation causes no modifications to the underlying database, return EDB.

(simulating after deleteOnes) given leakage $\mathbf{G}_t := \mathcal{L}_{\mathsf{OST}}(\mathsf{deleteOne}, \mathsf{filter})$, set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t := \Big\{ \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle, \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle \Big\}_{f \in \mathbf{F}}$$

and $\mathbf{E}_t := \emptyset$;

For all $f \in \mathbf{F}$, compute

$$\mathsf{EMM}_f \leftarrow \mathcal{S}_\Omega \left(\left\{ \big\langle\, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle, \big\langle\, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle \right\}, \emptyset \right),$$

and set $\mathsf{DDB} - \mathsf{id}$. Output the updated EDB.

The computational indistinguishability of the simulated EMM_f from a real EMM_f, for all $f \in \mathbf{F}$, follows from a standard hybrid argument and the adaptive security of Ω.

(simulating after updateOnes) given leakage $\mathbf{G}_t := \mathcal{L}_{\mathsf{OST}}(\mathsf{updateOne}, \mathsf{filter}, \mathsf{action})$, set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\begin{aligned} \mathbf{V}_t := \Bigg\{ \Big\{ & \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle, \\ & \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle, \\ & \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{put}, f \mid \perp \big\rangle, \\ & \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{write}, \mathsf{id}, f \mid \perp \big\rangle, \\ & \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{ins}, 2k, f \mid \perp \big\rangle, \\ & \big\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{app}, \mathsf{id}, f \mid \perp \big\rangle \Big\}_{f \in \mathbf{F}}, \\ & \big\langle\, \mathsf{OST}, t, \Delta, t_\Delta, \mathsf{updateOne}, |v|, f \mid \perp \big\rangle \Bigg\} \end{aligned}$$

and $\mathbf{E}_t := \emptyset$;

For all $f \in \mathbf{F}$, compute

$$\mathsf{EMM}_f \leftarrow \mathcal{S}_\Omega \left(\left\{ \big\langle\, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle, \big\langle\, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle \right\}, \emptyset \right).$$

and

$$\begin{aligned} \mathsf{EMM}_f \leftarrow \mathcal{S}_\Omega \Big(& \big\langle\, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{put}, f \mid \perp \big\rangle, \\ & \big\langle\, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}, \mathsf{write}, \mathsf{id}, f \mid \perp \big\rangle, \\ & \big\langle\, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{ins}, 2k, f \mid \perp \big\rangle, \\ & \big\langle\, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}, \mathsf{app}, \mathsf{id}, f \mid \perp \big\rangle, \emptyset \Big); \end{aligned}$$

Let $\mathbf{D} := \mathsf{DDB}[\_\mathsf{id} = \mathsf{id}]$, compute $\mathrm{ct} \leftarrow \mathcal{S}_{\mathsf{SKE}}(1^k, |v|)$ and set $\mathbf{D}.f := \mathrm{ct}$. Output the updated EDB.

The computational indistinguishability of the simulated EMM_f from a real EMM_f, for all $f \in \mathbf{F}$, follows from a standard hybrid argument and the adaptive security of Ω. Moreover, the computational indistinguishability of the simulated ciphertext for field f in $\mathbf{D}$ from a real document, follows from a standard hybrid argument and the CPA-security of SKE.

(simulating after compactions) given leakage $\mathbf{G}_t := \mathcal{L}_{\mathsf{OST}}(\mathsf{comp})$, set $\mathbf{G} := \mathbf{G} + \mathbf{G}_t$ and parse $\mathbf{G}_t = (\mathbf{V}_t, \mathbf{E}_t)$ as

$$\mathbf{V}_t := \left\{ \left\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{Init} \mid \perp \right\rangle, \left\langle\, \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{comp}, \mathsf{pcount} \mid \perp \right\rangle \right\}_{f \in \mathbf{F}}$$

and $\mathbf{E}_t := \emptyset$.

For all $f \in \mathbf{F}$, compute

$$\mathsf{EMM}_f \leftarrow \mathcal{S}_\Omega \left(\left\{ \left\langle\, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}, \mathsf{Init} \mid \perp \right\rangle, \left\langle\, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}, \mathsf{comp}, \mathsf{pcount} \mid \perp \right\rangle \right\}, \emptyset \right).$$

Output the updated EDB.

The computational indistinguishability of the simulated EMM_f from a real EMM_f, for all $f \in \mathbf{F}$, follows from a standard hybrid argument and the adaptive security of Ω.

■

Acknowledgements

The authors would like thank Mark Benvenuto, Spencer Jackson, Shreyas Kalyan, Erwin Pe, Judah Schvimer and Andy Schwerin for critical feedback throughout the design stages of this work. The authors would also like to thank Archita Agarwal, David Cash, Zach Espiritu and Marilyn George for their helpful comments on the design and analysis of OST.

Leakage profile $\mathcal{L}_\Omega$

```
• $\mathcal{L}_\Omega(\mathsf{op})$:
  1. if $\mathsf{op} = (\mathsf{init}, \theta)$,
     (a) set $t := 0$, $t_M := 0$, $t_C := 0$, $t_R := 0$ and $t_S := 0$;
     (b) initialize an empty set $P$;
     (c) set
         $$\begin{aligned}\mathbf{V}_t := \Big\{ & \big\langle\, \Omega, 0, \Sigma_M, 0, \mathsf{init}, \theta \mid \perp \big\rangle, \\ & \big\langle\, \Omega, 0, \Sigma_C, 0, \mathsf{init}, k \mid \perp \big\rangle, \\ & \big\langle\, \Omega, 0, \Sigma_R, t_R, \mathsf{init} \mid \perp \big\rangle, \\ & \big\langle\, \Omega, 0, \Sigma_S, 0, \mathsf{init} \mid \perp \big\rangle \Big\}\end{aligned}$$
         and $\mathbf{E}_t := \emptyset$;
  2. else if $\mathsf{op} = (\mathsf{put}, p, \ell, \mathbf{v})$
     (a) compute $t$++ and $t_M$++;
     (b) set $m \overset{\circ}{=} \#\mathbf{v}$;
     (c) sample $u \overset{\$}{\leftarrow} \{0, \ldots, p\}$;
     (d) let $\boldsymbol{\ell} \overset{\circ}{=} (\ell, u)$;
     (e) set $P := P \cup \boldsymbol{\ell}$;
     (f) set $\mathsf{count} := \#\mathbf{V}(\, \Sigma_M, \mathsf{write} \mid \boldsymbol{\ell}\,)$;
     (g) set
         $$\begin{aligned}\mathbf{V}_t := \Big\{ & \big\langle\, \Omega, t, \Sigma_C, t_C\text{++}, \mathsf{put} \mid \perp \big\rangle, \\ & \big\langle\, \Omega, t, \Sigma_M, t_M, \mathsf{write}, v_i \mid \boldsymbol{\ell}, \mathsf{count} + i \,\big\rangle_{i \in [m]}, \\ & \big\langle\, \Omega, t, \Sigma_S, t_S\text{++}, \mathsf{ins}, 2k \mid \perp \big\rangle, \\ & \big\langle\, \Omega, t, \Sigma_R, t_R\text{++}, \mathsf{app}, v_i \mid \perp \big\rangle_{i \in [m]} \Big\}\end{aligned}$$
         and $\mathbf{E}_t := \emptyset$;
  3. else if $\mathsf{op} = (\mathsf{erase}, v)$
     (a) compute $t$++;
     (b) set
         $$\mathbf{V}_t := \Big\{ \big\langle\, \Omega, t, \Sigma_M, t_M\text{++}, \mathsf{erase}, v \mid \perp \big\rangle, \big\langle\, \Omega, t, \Sigma_R, t_R\text{++}, \mathsf{erase}, v \mid \perp \big\rangle \Big\}$$
         and $\mathbf{E}_t := \emptyset$;
  4. else if $\mathsf{op} = \mathsf{comp}$,
     (a) compute $t$++;
     (b) set
         $$\mathbf{V}_t := \Big\{ \big\langle\, \Omega, t, \Sigma_S, 0, \mathsf{init} \mid \perp \big\rangle, \big\langle\, \Omega, t, \Sigma_C, t_C\text{++}, \mathsf{comp}, \#P \mid \perp \big\rangle \Big\}$$
         and $\mathbf{E}_t := \emptyset$;
     (c) set $t_S := 0$ and $P := \emptyset$;
  5. output $(\mathbf{V}_t, \mathbf{E}_t)$;
```

Figure 21: The leakage profile $\mathcal{L}_\Omega$.

Functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{DDB}}$

The functionality is parameterized with a leakage profile $\mathcal{L}$ and interacts with n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, a server $\mathbf{S}$ and an ideal adversary $\mathcal{S}$. It stores and manages a document database DDB using the following operations:

- upon receiving $(\mathsf{cid}, \mathsf{init})$ from a client, initialize and store a document database DDB and send $\big(\mathsf{cid}, \mathsf{init}, \mathcal{L}\big(\mathsf{init}\big)\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{insert}, \mathbf{D})$ from a client, set $\mathsf{DDB} := \mathsf{DDB} \cup \mathbf{D}$ and send $\big(\mathsf{cid}, \mathsf{insert}, \mathcal{L}\big(\mathsf{insert}, \mathbf{D}\big)\big)$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{find}, \mathsf{filter})$ from a client, return $\mathbf{R} := \mathsf{DDB}[\mathsf{filter}]$ to the client and send $(\mathsf{cid}, \mathsf{find}, \mathcal{L}(\mathsf{find}, \mathsf{filter}))$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{deleteOne}, \mathsf{filter})$ from a client, compute $\mathbf{R} := \mathsf{DDB}[\mathsf{filter}]$ and $\mathsf{DDB} - \mathbf{D}$ where $\mathbf{D} \overset{\$}{\leftarrow} \mathbf{R}$. Send $(\mathsf{cid}, \mathsf{deleteOne}, \mathcal{L}(\mathsf{deleteOne}, \mathsf{filter}))$ to $\mathcal{S}$;
- upon receiving $(\mathsf{cid}, \mathsf{updateOne}, \mathsf{filter}, \mathsf{action})$ from a client, compute $\mathbf{R} := \mathsf{DDB}[\mathsf{filter}]$ and set $\mathbf{D}.f := v$ where $\mathbf{D} \overset{\$}{\leftarrow} \mathbf{R}$. Send $(\mathsf{cid}, \mathsf{updateOne}, \mathcal{L}(\mathsf{updateOne}, \mathsf{filter}, \mathsf{action}))$ to $\mathcal{S}$.
- upon receiving $(\mathsf{cid}, \mathsf{compaction})$ from a client, send $(\mathsf{cid}, \mathsf{compaction}, \mathcal{L}(\mathsf{compaction}))$ to $\mathcal{S}$;

Figure 22: The ideal document database functionality.

Let $\Omega = (\mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Put}_{\mathbf{C},\mathbf{S}}, \mathsf{Get}_{\mathbf{C},\mathbf{S}}, \mathsf{Count}_{\mathbf{C},\mathbf{S}}, \mathsf{Test}_{\mathbf{C},\mathbf{S}}, \mathsf{Erase}_{\mathbf{C},\mathbf{S}}, \mathsf{Compaction}_{\mathbf{C},\mathbf{S}})$ be a stateless, response-revealing dynamic multi-map encryption scheme. Let $\mathbf{F}$ be the set of fields in the database and $\mathbf{P}$ be the contention factors of each field. Consider the stateless document database encryption scheme $\mathsf{OST} = (\mathsf{Gen}_{\mathbf{C},\mathbf{S}}, \mathsf{Init}_{\mathbf{C},\mathbf{S}}, \mathsf{Insert}_{\mathbf{C},\mathbf{S}}, \mathsf{Find}_{\mathbf{C},\mathbf{S}}, \mathsf{Delete}_{\mathbf{C},\mathbf{S}}, \mathsf{Update}_{\mathbf{C},\mathbf{S}}, \mathsf{Compaction}_{\mathbf{C},\mathbf{S}})$ defined as follows:

```
• $\mathsf{Gen}_{\mathbf{C},\mathbf{S}}(1^k)$:
  Client:
    1. sample $K_M \xleftarrow{\$} \{0,1\}^k$;
    2. output $K := K_M$;

• $\mathsf{Init}_{\mathbf{C},\mathbf{S}}(K, \theta, \mathbf{F}, \mathbf{P})$:
  Client:
    1. parse $\mathbf{P}$ as $(p_f)_{f\in\mathbf{F}}$;
    2. for all $f \in \mathbf{F}$,
       (a) compute $K_f := F_{K_M}[f,1]$ and $K'_f := F_{K_M}[f,2]$;
       (b) compute $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Init}_{\mathbf{C}_1}(K_f, \lambda, \theta)$;
    3. initialize an empty document database $\mathsf{DDB}$ with fields $\mathbf{F}$;
    4. output $K := (K_f, K'_f)_{f\in\mathbf{F}}$ and $st := (p_f)_{f\in\mathbf{F}}$
    5. send $\mathsf{EDB} := \big(\mathsf{DDB}, (\mathsf{EMM}_f)_{f\in\mathbf{F}}\big)$ to the server;

• $\mathsf{Insert}_{\mathbf{C},\mathbf{S}}(K, st, \mathbf{D}; \mathsf{EDB})$:
  Client:
    1. parse $K$ as $(K_f, K'_f)_{f\in\mathbf{F}}$ and $st$ as $(p_f)_{f\in\mathbf{F}}$;
    2. parse $\mathbf{D}$ as $(f, v)_{f\in\mathbf{F}}$;
    3. for all $f \in \mathbf{F}$,
       (a) compute $\mathsf{ptk}_f \leftarrow \Omega.\mathsf{Put}_{\mathbf{C}_1}(K_f, p_f, v)$;
       (b) compute $\mathrm{ct}_f := \mathsf{SKE.Enc}(K'_f, v)$;
    4. send $\mathbf{ED} := (f, \mathrm{ct}_f)_{f\in\mathbf{F}}$ and $\mathsf{itk} := (\mathsf{ptk}_f)_{f\in\mathbf{F}}$ to the server;
  Server:
    1. parse $\mathsf{EDB}$ as $\big(\mathsf{DDB}, (\mathsf{EMM}_f)_{f\in\mathbf{F}}\big)$;
    2. for all $f \in \mathbf{F}$,
       (a) compute $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Put}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{ptk}_f, \mathsf{id})$;
    3. sample $\mathsf{id} \xleftarrow{\$} \{0,1\}^k$;
    4. set $\mathbf{ED}$ to $\big((\_\mathsf{id}, \mathsf{id}), (f, \mathrm{ct}_f)_{f\in\mathbf{F}}\big)$;
    5. set $\mathsf{DDB} := \mathsf{DDB} \cup \mathbf{ED}$;
    6. output $\mathsf{EDB} := \big(\mathsf{DDB}, (\mathsf{EMM}_f)_{f\in\mathbf{F}}\big)$;
```

Figure 23: OST: a stateless database encryption scheme (part 1).

```
• $\mathsf{Find}_{\mathbf{C},\mathbf{S}}\big(K, st, \mathsf{filter}; \mathsf{EDB}\big)$:

  Client:
    1. parse $K$ as $(K_f, K'_f)_{f\in\mathbf{F}}$;
    2. if $\mathsf{filter} \equiv f = v$,
       (a) compute $\mathsf{gtk} \leftarrow \Omega.\mathsf{Get}_{\mathbf{C}_1}(K_f, p_f, v)$;
       (b) set $\mathsf{ftk} := (\mathsf{exactflag}, \mathsf{gtk})$ to the server;
    3. if $\mathsf{filter} \equiv f_1 = v_1 \bigwedge \cdots \bigwedge f_m = v_m$,
       (a) for all $1 \le i \le m$,
             i. compute $\mathsf{gtk}_i \leftarrow \Omega.\mathsf{Get}_{\mathbf{C}_1}(K_{f_i}, v_i)$;
            ii. compute $\mathsf{ttk}_i \leftarrow \Omega.\mathsf{Test}_{\mathbf{C}_1}(K_{f_i}, v_i)$;
           iii. compute $\mathsf{cttk}_i \leftarrow \Omega.\mathsf{Count}_{\mathbf{C}_1}(K_{f_i}, v_i)$;
       (b) set $\mathsf{cjtk} := (f_i, \mathsf{gtk}_i, \mathsf{ttk}_i, \mathsf{cttk}_i)_{1\le i\le m}$ to the server;
       (c) set $\mathsf{ftk} := (\mathsf{conjflag}, \mathsf{cjtk})$ to the server;
    4. send $\mathsf{ftk}$ to the server;

  Server:
    1. parse $\mathsf{EDB}$ as $\big(\mathsf{DDB}, \big(\mathsf{EMM}_f\big)_{f\in\mathbf{F}}\big)$;
    2. parse $\mathsf{ftk}$ as $(\mathsf{flag}, \mathsf{tk})$;
    3. if $\mathsf{flag} = \mathsf{exactflag}$,
       (a) parse $\mathsf{tk}$ as $\mathsf{gtk}$;
       (b) initialize an empty set $\mathbf{R}$;
       (c) compute $\mathbf{ids} \leftarrow \Omega.\mathsf{Get}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{gtk})$;
       (d) compute $\mathbf{R} := \mathsf{DDB}\left[\bigwedge_{\mathsf{id}\in\mathbf{ids}} \_\mathsf{id} = \mathsf{id}\right]$;
       (e) send $\mathbf{R}$ to client;
    4. else if $\mathsf{flag} = \mathsf{conjflag}$,
       (a) parse $\mathsf{tk}$ as $(f_i, \mathsf{gtk}_i, \mathsf{ttk}_i, \mathsf{cttk}_i)_{1\le i\le m}$;
       (b) initialize an empty set $\mathbf{R}$;
       (c) for all $1 \le i \le m$,
             i. compute $\mathsf{count}_{f_i} \leftarrow \Omega.\mathsf{Count}_{\mathbf{S}_2}(\mathsf{EMM}_{f_i}, \mathsf{cttk}_i)$;
            ii. if $\mathsf{count}_{f_i} = 0$, then output $\mathbf{R} := \emptyset$;
       (d) set $f^\star := \arg\min_{i\in[m]}(\mathsf{count}_{f_i})$;
       (e) compute $\mathbf{ids} \leftarrow \Omega.\mathsf{Get}_{\mathbf{S}_2}(\mathsf{EMM}_{f^\star}, \mathsf{gtk}_{f^\star})$;
       (f) for all $\mathsf{id} \in \mathbf{ids}$,
             i. for all $f \in \{f_1, \ldots, f_m\} \setminus \{f^\star\}$,
                A. compute $b \leftarrow \Omega.\mathsf{Test}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{ttk}_f, \mathsf{id})$;
                B. if $b = \mathsf{false}$, set $\mathbf{ids} := \mathbf{ids} \setminus \{\mathsf{id}\}$ and exit loop;
       (g) compute $\mathbf{R} := \mathsf{DDB}\left[\bigwedge_{\mathsf{id}\in\mathbf{ids}} \_\mathsf{id} = \mathsf{id}\right]$;
       (h) send $\mathbf{R}$ to the client;

  Client:
    1. initialize an empty set $\mathbf{R}'$;
    2. for every $\mathbf{ED} \in \mathbf{R}$,
       (a) parse $\mathbf{ED}$ as $((\_\mathsf{id}, \mathsf{id}), (f, \mathrm{ct}_f)_{f\in\mathbf{F}})$;
       (b) for all $f \in \mathbf{F}$,
             i. compute $v := \mathsf{SKE.Dec}(K'_f, \mathrm{ct}_f)$;
       (c) set $\mathbf{D}$ as $((\_\mathsf{id}, \mathsf{id}), (f, v)_{f\in\mathbf{F}})$ and add $\mathbf{D}$ to $\mathbf{R}'$;
    3. output $\mathbf{R}'$;
```

Figure 24: OST: a stateless database encryption scheme (part 2).

```
• $\mathsf{DeleteOne}_{\mathbf{C},\mathbf{S}}\big(K, st, \mathsf{filter}; \mathsf{EDB}\big)$:

  Client:
    1. compute $\mathsf{ftk} \leftarrow \Omega.\mathsf{Find}_{\mathbf{C}_1}(K, \mathsf{filter})$;
    2. send $\mathsf{dtk} := \mathsf{ftk}$ to the server;

  Server:
    1. parse $\mathsf{EDB}$ as $\big(\mathsf{DDB}, \big(\mathsf{EMM}_f\big)_{f\in\mathbf{EF}}\big)$ and $\mathsf{dtk}$ as $\mathsf{ftk}$;
    2. compute $\mathbf{ids} \leftarrow \mathsf{OST}.\mathsf{Find}_{\mathbf{S}_2}(\mathsf{EDB}, \mathsf{ftk})$;
    3. set $\mathsf{id} \overset{\$}{\leftarrow} \mathbf{ids}$;
    4. compute $\mathsf{DDB} - \mathsf{id}$;
    5. for all $f \in \mathbf{F}$, compute $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{id})$;
    6. output $\mathsf{EDB}$.

• $\mathsf{UpdateOne}_{\mathbf{C},\mathbf{S}}\big(K, st, \mathsf{filter}, \mathsf{action}; \mathsf{EDB}\big)$:

  Client:
    1. parse $K$ as $(K_f, K'_f)_{f\in\mathbf{F}}$ and $st$ as $(p_f)_{f\in\mathbf{F}}$;
    2. parse $\mathsf{action}$ as $(f, v)$;
    3. compute $\mathsf{ptk}_f \leftarrow \Omega.\mathsf{Put}_{\mathbf{C}_1}(K_f, p_f, v)$;
    4. compute $\mathrm{ct}_f := \mathsf{SKE}.\mathsf{Enc}(K'_f, v)$;
    5. compute $\mathsf{ftk} \leftarrow \mathsf{OST}.\mathsf{Find}_{\mathbf{C}_1}(K, \mathsf{filter})$;
    6. set $\mathsf{utk} := (\mathsf{ftk}, f, \mathrm{ct}_f, \mathsf{ptk}_f)$;
    7. send $\mathsf{utk}$ to the server;

  Server:
    1. parse $\mathsf{EDB}$ as $\big(\mathsf{DDB}, \big(\mathsf{EMM}_f\big)_{f\in\mathbf{EF}}\big)$;
    2. parse $\mathsf{utk}$ as $(\mathsf{ftk}, f, \mathrm{ct}_f, \mathsf{ptk}_f)$;
    3. compute $\mathbf{ids} \leftarrow \mathsf{OST}.\mathsf{Find}_{\mathbf{S}_2}(\mathsf{EDB}, \mathsf{ftk})$;
    4. set $\mathsf{id} \overset{\$}{\leftarrow} \mathbf{ids}$;
    5. compute $\mathbf{D} := \mathsf{DDB}[\_\mathsf{id} = \mathsf{id}]$;
    6. set $\mathbf{D}.f := \mathrm{ct}_f$;
    7. compute $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Erase}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{id})$;
    8. compute $\mathsf{EMM}_f \leftarrow \Omega.\mathsf{Put}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{ptk}_f, \mathsf{id})$;
    9. output $\mathsf{EDB} = \big(\mathsf{DDB}, \big(\mathsf{EMM}_f\big)_{f\in\mathbf{F}}\big)$.

• $\mathsf{Compaction}_{\mathbf{C},\mathbf{S}}\big(K; \mathsf{EDB}\big)$:

  Client:
    1. parse $K$ as $(K_f, K'_f)_{f\in\mathbf{F}}$;
    2. for all $f \in \mathbf{F}$, compute $\mathsf{ctk}_f \leftarrow \Omega.\mathsf{Compaction}_{\mathbf{C}_1}(K_f)$;
    3. output $\mathsf{ctk} := \big(\mathsf{ctk}_f\big)_{f\in\mathbf{F}}$;

  Server
    1. parse $\mathsf{EDB}$ as $\big(\mathsf{DDB}, \big(\mathsf{EMM}_f\big)_{f\in\mathbf{F}}\big)$;
    2. for all $f \in \mathbf{F}$, compute $\mathsf{EMM}'_f \leftarrow \Omega.\mathsf{Compaction}_{\mathbf{S}_2}(\mathsf{EMM}_f, \mathsf{ctk}_f)$;
    3. output the updated $\mathsf{EDB}$;
```

Figure 25: OST: a stateless database encryption scheme (part 3).

Leakage profile $\mathcal{L}_{\mathsf{OST}}$

- $\mathcal{L}_{\mathsf{OST}}(\mathsf{op})$:
  1. if $\mathsf{op} = (\mathsf{init}, \theta, \mathbf{F}, \mathbf{P})$,
     (a) set $t := 0$ and $t_{\mathsf{DDB}} := 0$;
     (b) for all $f \in \mathbf{F}$, set $t_{\Omega,f} := 0$, $t_{M,f} := 0$, $t_{C,f} := 0$, $t_{S,f} := 0$, $t_{R,f} := 0$ and $\mathsf{pcount}_f := 0$;
     (c) initialize an empty database DDB;
     (d) set

$$\mathbf{V}_t := \Bigg\{ \bigg\{ \big\langle \mathsf{OST}, 0, \Omega, 0, \Sigma_M, 0, \mathsf{init}, \theta, f \mid \perp \big\rangle, \\ \big\langle \mathsf{OST}, 0, \Omega, 0, \Sigma_C, 0, \mathsf{init}, \theta, f \mid \perp \big\rangle, \\ \big\langle \mathsf{OST}, 0, \Omega, 0, \Sigma_S, 0, \mathsf{init}, f \mid \perp \big\rangle, \\ \big\langle \mathsf{OST}, 0, \Omega, 0, \Sigma_R, 0, \mathsf{init}, f \mid \perp \big\rangle \bigg\}_{f \in \mathbf{F}}, \\ \big\langle \mathsf{OST}, 0, \Omega, 0, \Delta, 0, \mathsf{init}, \mathbf{F}, \mathbf{P} \mid \perp \big\rangle \Bigg\}$$

     and $\mathbf{E}_t := \emptyset$;
  2. else if $\mathsf{op} = (\mathsf{insert}, \mathbf{D})$
     (a) compute t++ and for all $f \in \mathbf{F}$, compute $t_{\Omega,f}$++ and pcount++;
     (b) parse $\mathbf{D}$ as $(f, v_f)_{f \in \mathbf{F}}$ and add $\mathbf{D}$ to DDB;
     (c) compute $\mathsf{id} \overset{\$}{\leftarrow} \{0,1\}^k$;
     (d) set

$$\mathbf{V}_t := \Bigg\{ \bigg\{ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}\text{++}, \mathsf{put}, f \mid \perp \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}\text{++}, \mathsf{write}, \mathsf{id}, f \mid \perp \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}\text{++}, \mathsf{ins}, 2k, f \mid \perp \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}\text{++}, \mathsf{app}, \mathsf{id}, f \mid \perp \big\rangle \bigg\}_{f \in \mathbf{F}}, \\ \big\langle \mathsf{OST}, t, \Delta, t_\Delta\text{++}, \mathsf{insert}, |v_f|, f \mid \perp \big\rangle_{f \in \mathbf{F}} \Bigg\}$$

     and $\mathbf{E}_t := \emptyset$;
  3. else if $\mathsf{op} = (\mathsf{deleteOne}, \mathsf{filter})$
     (a) compute t++ and for all $f \in \mathbf{F}$, $t_{\Omega,f}$++;
     (b) compute $\mathsf{id} \overset{\$}{\leftarrow} \mathsf{DDB}[\mathsf{filter}]$;
     (c) set

$$\mathbf{V}_t := \Big\{ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}\text{++}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle, \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}\text{++}, \mathsf{erase}, \mathsf{id} \mid \perp \big\rangle \Big\}_{f \in \mathbf{F}}$$

     and $\mathbf{E}_t := \emptyset$;

Figure 26: The leakage profile $\mathcal{L}_{\mathsf{OST}}$.

Leakage profile $\mathcal{L}_{\mathsf{OST}}$ (cont.)

```
4. else if $\mathsf{op} = (\mathsf{updateOne}, \mathsf{filter}, \mathsf{action})$
    (a) compute $t$++ and for all $f \in \mathbf{F}$, compute $t_{\Omega,f}$++ and $\mathsf{pcount}$++;
    (b) compute $\mathsf{id} \xleftarrow{\$} \mathsf{DDB}[\mathsf{filter}]$ and parse $\mathsf{action}$ as $(f, v)$;
    (c) set
```

$$\mathbf{V}_t := \Bigg\{ \Bigg\{ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}{+}{+}, \mathsf{erase}, \mathsf{id} \mid \bot \big\rangle \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}{+}{+}, \mathsf{erase}, \mathsf{id} \mid \bot \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}{+}{+}, \mathsf{put}, f \mid \bot \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_M, t_{M,f}{+}{+}, \mathsf{write}, \mathsf{id}, f \mid \bot \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}{+}{+}, \mathsf{ins}, 2k, f \mid \bot \big\rangle, \\ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_R, t_{R,f}{+}{+}, \mathsf{app}, \mathsf{id}, f \mid \bot \big\rangle \Bigg\}_{f \in \mathbf{F}}, \\ \big\langle \mathsf{OST}, t, \Delta, t_\Delta{+}{+}, \mathsf{updateOne}, |v|, f \mid \bot \big\rangle \Bigg\}$$

```
        and $\mathbf{E}_t := \emptyset$;
5. else if $\mathsf{op} = \mathsf{comp}$,
    (a) compute $t$++ and for all $f \in \mathbf{F}$, $t_{\Omega,f}$++;
    (b) set
```

$$\mathbf{V}_t := \Big\{ \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_S, t_{S,f}{+}{+}, \mathsf{Init} \mid \bot \big\rangle, \big\langle \mathsf{OST}, t, \Omega, t_{\Omega,f}, \Sigma_C, t_{C,f}{+}{+}, \mathsf{comp}, \mathsf{pcount} \mid \bot \big\rangle \Big\}_{f \in \mathbf{F}}$$

```
        and $\mathbf{E}_t := \emptyset$;
     1. for all $f \in \mathbf{F}$, set $\mathsf{pcount}_f := 0$;
6. output $(\mathbf{V}_t, \mathbf{E}_t)$;
```

Figure 27: The leakage profile $\mathcal{L}_{\mathsf{OST}}$.

References

[1] R. Ada Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2004.

[3] G. Amjad, S. Kamara, and T. Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.

[4] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[5] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *ACM Symposium on Theory of Computing (STOC '16)*, STOC '16, pages 1101–1114, New York, NY, USA, 2016. ACM.

[6] G. Asharov, G. Segev, and I. Shahaf. Tight tradeoffs in searchable symmetric encryption. In *Annual International Cryptology Conference*, pages 407–436. Springer, 2018.

[7] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[8] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS 2008)*, pages 257–266. ACM, 2008.

[9] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment*, 11(11):1715–1728, 2018.

[10] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

[11] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[12] D. Boneh, G. di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT '04*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004.

[13] R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.

[14] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM Conference on Computer and Communications Security (CCS '17)*, 2017.

[15] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1), 2000.

[16] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Communications and Computer Security (CCS '15)*, pages 668–679. ACM, 2015.

[17] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[18] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[19] D. Cash, R. Ng, and A. Rivkin. Improved structured encryption for sql databases via hybrid indexing. In *Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II*, pages 480–510. Springer, 2021.

[20] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[21] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[22] M. Chase and S. Kamara. Structured encryption and controlled disclosure. Technical Report 2011/010.pdf, IACR Cryptology ePrint Archive, 2010.

[23] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[24] I. Demertzis, D. Papadopoulos, and C. Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *Advances in Cryptology - CRYPTO '18*, pages 371–406. Springer, 2018.

[25] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data*, pages 185–198. ACM, 2016.

[26] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *ACM International Conference on Management of Data (SIGMOD '17)*, SIGMOD '17, pages 1053–1067, New York, NY, USA, 2017. ACM.

[27] F. B. Durak, T. M. DuBuisson, and D. Cash. What else is revealed by order-revealing encryption? In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

[28] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science*, volume 9327, pages 123–145, 2015.

[29] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: a scalable private dbms. In *IEEE Symposium on Security and Privacy*, pages 395–410. IEEE, 2015.

[30] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016*, pages 563–592, 2016.

[31] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[32] M. George, S. Kamra, and T. Moataz. Structured encryption and dynamic leakage suppression. In *Advances in Cryptology - EUROCRYPT 2021*, 2021.

[33] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[34] P. Grofig, I. Hang, M. Härterich, F. Kerschbaum, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Privacy by encrypted databases. In *Privacy Technologies and Policy: Second Annual Privacy Forum, APF 2014, Athens, Greece, May 20-21, 2014. Proceedings 2*, pages 56–69. Springer, 2014.

[35] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 315–331. ACM, 2018.

[36] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *Workshop on Hot Topics in Operating Systems (HotOS '17)*, pages 162–168, New York, NY, USA, 2017. ACM.

[37] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002.

[38] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *ACM Conference on Computer and Communications Security (CCS '14)*, CCS '14, pages 310–320, New York, NY, USA, 2014. ACM.

[39] J. Jaeger and N. Tyagi. Handling adaptive compromise for practical encryption schemes. In *Advances in Cryptology - CRYPTO '20*, pages 3–32. Springer, 2020.

[40] C. Jutla and S. Patranabis. Efficient searchable symmetric encryption for join queries. In *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part III*, pages 304–333. Springer, 2023.

[41] S. Kamara, A. Kati, T. Moataz, T. Schneider, A. Treiber, and M. Yonli. Sok: Cryptanalysis of encrypted search with leaker – a framework for leakage attack evaluation on real-world data. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 90–108, 2022.

[42] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology - EUROCRYPT '17*, 2017.

[43] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.

[44] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakae suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[45] S. Kamara, T. Moataz, S. Zdonik, and Z. Zhao. Opx: An optimal relational database encryption scheme. Technical report, IACR ePrint Cryptography Archive, 2020.

[46] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.

[47] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[48] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[49] G. Kellaris, G. Kollios, K. Nissim, and A. O. Neill. Generic attacks on secure outsourced databases. In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

[50] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1223–1240. IEEE, 2020.

[51] I. Miers and P. Mohassel. Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. Cryptology ePrint Archive, Report 2016/830, 2016. http://eprint.iacr.org/2016/830.

[52] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 644–655. ACM, 2015.

[53] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[54] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

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