© IFIP International Federation for Information Processing 2021
K. Barker, K. Ghazinour (eds.), *Data and Applications Security and Privacy XXXV*, Lecture Notes in Computer Science 12840
https://doi.org/10.1007/978-3-030-81242-3_4

Simple Storage-Saving Structure for Volume-Hiding Encrypted Multi-maps

(A Slot in Need is a Slot Indeed)

Jiafan Wang[1] ✉ and Sherman S. M. Chow[1] ✉

(1) Department of Information Engineering, Chinese University of Hong Kong, Shatin, N.T., Hong Kong

✉ **Jiafan Wang**
Email: wj016@ie.cuhk.edu.hk

✉ **Sherman S. M. Chow (Corresponding author)**
Email: sherman@ie.cuhk.edu.hk

Abstract

Severe consequences in volume leakage (subject to the conditions required by specific attacks) stimulate a new research direction (Eurocrypt 2019) of volume-hiding structured encryption (STE), particularly encrypted multi-maps (EMM), in which all queries should share the same (as the largest) response size unless the scheme is lossy. Meanwhile, note that the responses are originated from the actual ciphertexts outsourced to the server. Conventional wisdom suggests that the ciphertexts (to be accessed by the server while answering a query) should also contain many dummy results to make a query look uniform with others. Supporting updates is also natural; however, attaching dummy results to a query also complicates the operation and leakage of updates, which excludes many advanced data structures, e.g., cuckoo hashing (CCS 2019). This paper proposes a space-efficient EMM without storing any dummy ciphertext, which is volume hiding against passive adversaries (SP 2021) and compatible with dynamic extensions. Its crux structure is a hash ring, which is famous for load balancing but rarely appears in any STE. Efficiency-wise, our scheme beats the state-of-the-art (Eurocrypt 2019, CCS 2019), maintaining the necessary communication overhead and downsizing the server storage to be linear in the number of values in the EMM, while ruling out any data loss due to truncations or differential privacy.

Keywords Volume hiding – Encrypted multi-maps – Consistent hashing – Structured encryption – Dynamic symmetric searchable encryption

Sherman S. M. Chow is supported by General Research Fund (Project Numbers: CUHK 14210217 and CUHK 14209918) from Research Grant Council.

1 Introduction

Structured encryption (STE), introduced by Chase and Kamara [6], enables a client to outsource an encrypted data structure to an untrusted cloud server for later private queries. Encrypted multi-map (EMM) is an important instance of STE. A multi-map (MM) usually contains a set of keys, each associated with a tuple of values. Treating each key as a keyword and the values as (identifiers of the) documents containing the keyword, EMM realizes searchable symmetric encryption (SSE) for keyword searches. It serves as the core of SSE for more complex queries (*e.g.*, range [7, 8, 30], graph [24, 25], skyline [31], and SQL [13]) and improved security for dynamism (*e.g.*, forward and backward privacy [3, 4, 25]). Many EMMs are well-known to be lightweight and efficient for millions of records.

An STE is deemed secure with respect to a given leakage profile if its operations reveal nothing beyond well-defined functions capturing the leakage of the private input data. Leakages allow STE to be efficient, but whether they are “benign” enough is still under study. Though most leakage-abuse attacks can be challenging to realize in practice, *e.g.*, require knowing a large portion of data and some queried keys [5, 12] or typical distributions of client queries [10, 18, 22], they stimulate the research of minimizing the leakage of STE in the first place.

This work focuses on the design of volume-hiding EMMs, a notion recently proposed by Kamara and Moataz [14] and further formalized by Patel *et al.* [27]. The volume refers to *the number of associated values* of any key. A very recent work [21] demonstrates that a passive attacker solely observing EMM accesses can still exploit the volume leakage of range queries to reconstruct the private plaintext. A volume-hiding EMM can hide the *response length* of a query, which effectively mitigates the damage caused by volume-abusing attacks [2, 10, 18, 21, 22]. In this work, we propose volume-hiding EMMs against such adversaries, which compare favorably with state-of-the-art [14, 27] in efficiency.

1.1 Related Work – Many Dummies in Storage and Communication

For a multi-map MM, let ℓ be *the maximum volume*, *m* be *the number of distinct keys*, and *n* be *the total number of values over all keys*. Usually, $m \cdot \ell \gg n$. For volume hiding, the response volume is maintained to be at least $\Theta(\ell)$.

The naïve padding approach pads up to $(\ell - 1)$ dummy values to any key with a volume less than ℓ, which increases the server-side storage to $m \cdot \ell$. Another solution is oblivious RAMs (ORAM) [9]. Among the oblivious accesses to each associated value, up to ℓ extra fake accesses are made. While ORAM has been made practically efficient, it inherently incurs large communication overheads. Both approaches are undesirable in practice, motivating cleverer constructions.

The first construction VLH [14] reduces the storage overhead of the naïve approach roughly by half under specific parameters, yet still far from the ideal goal of $\Theta(n)$. For each key, it determines its volume via a pseudorandom function (PRF). It pads dummies when the PRF-derived volume is larger than the real one and truncates values when it is smaller. It is thus lossy, which is often undesirable. Although the number of truncated keys is relatively small under the Zipf's distribution, no data loss guarantee is given for a general MM.

The second construction AVLH [14] achieves $\Theta(n)$ storage. For each key, it chooses a set of ℓ related bins uniformly at random. Each value is placed into a distinct bin related to its associating key. All bins are padded with dummies to the maximum bin size after all the *n* values have been arranged. According to the balls-into-bins analysis, the bin size would be $\Omega(\log n)$, resulting in $\Theta(\ell \cdot \log n)$ communication and computation overheads for any query. AVLH further optimizes the server storage from $\Theta(n)$ to $\Theta(n - \sqrt{m} \cdot \mathsf{polylog}(m)))$ for *concentrated* MM with many values associated with a large number of keys. It ensures that these values only appear once among all bins. However, this variant requires the hardness of the densest subgraph problem [20], which is not thoroughly studied in literature and hard to determine related parameters for concrete security.

Patel *et al.* [27] criticized $\mathsf{(A)VLH}$ [14] for lossiness, large storage and query overheads, or reliance on a less-studied assumption. Observed that minimizing storage overhead is a typical hashing problem: placing *n* items to $\Theta(n)$ locations that can be looked up by probing a small number of locations, they proposed two schemes exploiting "cuckoo hashing with a stash" [20], which store each value in one of two hash slots or a client stash, and abort if the stash overflows.

The first scheme dprfMM outsources a hash table of length $(2 + \alpha)n$, which contains those key-value pairs survived from cuckoo hashing evictions and some dummies. A query token consists of 2ℓ hash slots possibly related to the queried key. The server returns 2ℓ ciphertexts retrieved from these slots, each containing a desirable value, a value of other keys, or a dummy. The client combines the value tuple of the queried key from both the local stash and the server response. The token size could be reduced by delegatable PRF [19] with extra computation.

The second construction dpMM [27] outsources a cuckoo hash table storing the volume of each key, in addition to the one used in dprfMM. It leads to the $(2 + \alpha)(n + m)$ storage overhead. To query, it first takes the client one roundtrip to fetch the volume of the queried key. With the noise from the Laplacian distribution, the client issues a token consisting of hash slots, whose size is the volume of the queried key plus the (possibly negative) noise and an extra adjustable parameter. The following query steps remain the same as dprfMM. With a small amount of perturbation over the response length, the query communication of dpMM is close to the optimal case, which outperforms any other schemes, but only under the relaxed (differential) privacy guarantee with potential data loss.

Our construction, achieving the known best communication overhead (*i.e.*, ℓ) under the consideration of volume hiding against passive adversaries [21], avoids any locally stored MM (*cf.*, the stash from cuckoo hashing in dprfMM and dpMM [27]) or any loss of values (*cf.*, due to differential privacy in dpMM [27] or PRF-based transform in VLH [14]), while further improving the server storage by wiping out dummies (*cf.*, dummy padding required in the state of the art [14, 27]).

1.2 Volume-Hiding EMM with "Volume-Affecting" Updates

To make an EMM scheme dynamic, existing dynamic structures [1, 3, 16, 23–25] may not be applicable to volume-hiding EMMs given the more rigorous leakage control. As discussed, the security and efficiency of volume-hiding EMMs are impacted by the maximum volume of the input MM, which may be changed by even a single update. To hide the key with the (real) maximum volume, the client must expensively introduce a dummy for each distinct key.

Indeed, the only two dynamic volume-hiding EMMs [14], $\mathsf{VLH}^{\mathsf{d}}$ and

$\mathsf{AVLH}^{\mathsf{d}}$, only support a limited form of updates (*cf.*, [25]). They are extended from VLH and AVLH, respectively. $\mathsf{AVLH}^{\mathsf{d}}$ only supports edition, which modifies the value of a key-value tuple, but not the key. $\mathsf{VLH}^{\mathsf{d}}$ further supports "addition" and "deletion." Addition can only add a key that never exists in the MM. In other words, adding one more value to an existing key, which affects ℓ, is not possible. Similarly, $\mathsf{VLH}^{\mathsf{d}}$ can only delete all values associated with a key. All these updates are relatively straightforward, mostly without affecting the maximum volume. Furthermore, both constructions require the client to get back the tuple to be edited by querying before (re-)uploading the updated tuple. Finally, they lack forward privacy and backward privacy [3, 4], which have become the *de facto* standard of dynamic STE due to the adaptive injection attack [33].

One may attempt to upgrade dpMM and dprfMM [27] with dynamic updates. It will involve insertions to the underlying cuckoo hashing table, and the chain of evictions it may incur becomes more complicated in the EMM setting. Firstly, the server cannot determine whether the slot to be inserted is empty unless the client decides to help by an extra round of communication. Also, the eviction chain will affect multiple slots. To avoid extra leakage, such as the occupancy information, one needs to make sure insertions for all updates result in the same eviction time, which significantly burdens the computation complexity. This showcases that designing a volume-hiding EMM that is compatible with updates in the first place appears to be a better approach, which also steers our EMM design.

1.3 Our Contributions and a Technical Overview

We observe that most volume-hiding EMMs hide the volume of a key using the values of other keys and dummies. Dummies carrying no meaningful information are the culprits of the significantly larger storage overhead. Our idea is to wipe out these dummies and realize volume hiding *solely* with meaningful value tuples. Beyond the storage of tuples themselves, we can further free the server from storing a global data structure filled with many dummies here and there (*cf.*, the entire hash table outsourced by dprfMM / dpMM [27] of size $(2+\alpha)n$ or $(2+\alpha)(n+m)$ for a positive constant α). Our volume-hiding EMM thus beats the best-known server storage (from $(2+\alpha)n$ to $2n$) with an even lower communication overhead (from 2ℓ to ℓ). The query overhead remains asymptotically the same as AVLH for a general (*i.e.*, not necessarily concentrated) MM [14]. Notably, our scheme neither stores any local (partial) MMs nor suffers from any data loss due to truncations or differential privacy. Table 1 summarizes the comparison[1].

As the trade-off for the above benefits, we consider passive adversaries who can observe EMM accesses of the data owner but cannot force arbitrary accesses of their wishes, capturing realistic scenarios of cloud storage applications. Nevertheless, solely storing meaningful tuples for supporting volume-hiding queries is still non-trivial. To avoid leakages of the underlying MM incurred by merely outsourcing the EMM, the responses for each query should be randomly distributed across the storage space. However, when the responses of a query can only be padded with those of another query, it appears that the client needs a sophisticated mapping, say, by solving a combinatorics and optimization problem, to carefully “plan ahead” what are the responses to be “borrowed” as dummies, which is essentially a rephrasing of the EMM problem we aim to solve.

Table 1. Comparison of volume-hiding encrypted multi-maps

Scheme	**Communication (response size)**	**Computation (for query)**	**Server storage (encrypted multi-maps)**	**No**	**No**
Naïve padding	ℓ	ℓ	$m \cdot \ell$	✓	✓
VLH [14]	$\Theta(\ell)$	$\Theta(\ell)$	$\Theta(m \cdot \ell)$	✗	✓
AVLH [14] (General MM)	$\Theta(\ell \cdot \log n)$	$\Theta(\ell \cdot \log n)$	$\Theta(n)$	✓	✓
AVLH [14] (Concentrated)	$\Theta(\ell \cdot \frac{n}{\mathsf{polylog}(m)})$	$\Theta(\ell \cdot \frac{n}{\mathsf{polylog}(m)})$	$\Theta(n - \sqrt{m} \cdot \mathsf{polylog}(m))$	✓	✓
dprfMM [27]	2ℓ	2ℓ	$(2+\alpha)n$	✓	✗
dpMM [27]	$2\ell(\mathsf{key}) + \omega(\log \lambda')$	$2\ell(\mathsf{key}) + \omega(\log \lambda')$	$(2+\alpha)(n+m)$	✗	✗

Our $\mathcal{S}^4$	ℓ	$\Theta(\ell \cdot \log n)$	$2n$	✓	✓

(Legends) m: the number of distinct keys, n: the total number of values across all keys
ℓ: the maximum volume across all keys, $\ell(\mathsf{key})$: the volume of key

λ': a security parameter for differential privacy, α: a positive constant

"No stash" means no need for the client to locally store any partial multi-map.
"No loss" means no loss of data, *e.g.*, due to differential privacy or truncations.

As a simple storage-saving structure, our EMM is named $\mathcal{S}^4$. It is based on a variant of hash ring [17], a technique that we first bring to the context of STE (SSE included) to our knowledge. The ring stores the encrypted contents of the original MM (notably without dummies). The query for a key specifies ring slots associated with it. Intuitively, any query should contain the same amount of ring slots, or metaphorically speaking, shelters for the key-value pairs. We still need to ensure that, when a shelter is not in real need, *i.e.*, when a key associates with less than ℓ values, the shelter pointed to by the key is storing the real value of another key in need. Our strategy of using the hash ring makes this possible.

We also explore how to update volume-hiding EMMs efficiently. We borrow an approach from the database community [7] of setting up a new instance for updates and periodically merging them, which matches the feature of some real-world systems (*e.g.*, commercial database Vertica [26]) that handles updates efficiently in batch. We further consider forward and backward privacy, resulting in the first dynamic volume-hiding EMM with such privacy guarantees. A concurrent work of Gui *et al.* [11] proposed a scheme that is forward and backward private while *mitigating* the volume leakage. Nevertheless, its bucketization technique exposes an approximate volume of each query even to a passive adversary.

2 Preliminary

2.1 Notation, Basic Structure and Primitive

Notation. Let λ be the security parameter and all algorithms take λ implicitly as input. $\mathsf{negl}(\lambda)$ is a negligible function in λ. PPT stands for probabilistic polynomial-time. For a set *X*, $x \leftarrow_{\$} X$ samples an element *x* uniformly from *X*. For an algorithm *A*, $x \leftarrow A$ means *x* is an output of *A*. || denotes string concatenation. [*n*] denotes the set $\{1, \ldots, n\}$. $[a\,..\,b]$ denotes $\{a, a+1, \ldots, b\}$.

Basic Data Structures. We recall operations of arrays, dictionaries, and multi-maps used in this work. An array A of the capacity *n* is a sequence of *n* values indexed by integer identifiers from 0 to $n-1$. For an index *i* and a value *v*, we set the *i*-th value via $\mathsf{A}[i] \leftarrow v$, and get the *i*-th value via $v \leftarrow \mathsf{A}[i]$.

A dictionary DX is a collection of key-value pairs. For key in the key space $\mathcal{K}$ and a value *v*, $v \leftarrow \mathsf{DX}[\mathsf{key}]$ (resp. $\mathsf{DX}[\mathsf{key}] \leftarrow v$) represents getting (resp. putting) *v* from (resp. into) $\mathsf{DX}[\mathsf{key}]$. A multi-map MM is a collection of key-tuple pairs $\{(\mathsf{key}, \vec{v})\}$. One could get/put tuples associated with a specific key in MM with similar notions as DX. Specifically, we denote the number of values in the tuple $\vec{v}$ by $[|\vec{v}|]$. For $i \in [|\vec{v}|]$, one could get/put the *i*-th value of $\vec{v}$ with handle $\vec{v}[i]$.

Basic Cryptographic Tools. Pseudorandom functions (PRF) are polynomial-time computable functions that cannot be distinguished from a truly random function by any PPT adversary. The symmetric-key encryption scheme (SKE) used in this work is *random-ciphertext-secure against chosen-plaintext attacks* (RCPA), which requires ciphertexts to be computationally indistinguishable from random even if the PPT adversary adaptively accesses the encryption oracle. It can be obtained from the standard PRF-based SKE or AES in counter mode.

2.2 Structured Encryption for Multi-maps

A structured encryption scheme STE encrypts a data structure such that the client could privately query its content. It can be either response-revealing (revealing the query response in plaintext) or response-hiding (keeping it private). An STE is non-interactive if the response can be fetched in a single roundtrip.

We focus on non-interactive and response-hiding STE for MMs, *i.e.*, EMMs.

Definition 1 (Non-Interactive Structured Encryption for MMs). An STE scheme for MM $\Sigma = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Reply}, \mathsf{Result})$ is defined as follows.

- $(K, \mathsf{st}, \mathsf{EMM}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{MM})$ is a probabilistic algorithm executed by the client that takes as input the security parameter λ and the input multi-map MM. It outputs a private key *K*, a state st, and the encrypted multi-map EMM. The client stores (K, st) while sending EMM to the server.
- $\mathsf{tk} \leftarrow \mathsf{Query}(K, \mathsf{st}, \mathsf{key})$ is a (possibly) probabilistic algorithm executed by the client that takes as input the private key *K*, the state st, and the key from the key universe $\mathcal{K}$ of MM. It outputs a query token tk to be sent to the server.
- $\mathsf{res} \leftarrow \mathsf{Reply}(\mathsf{tk}, \mathsf{EMM})$ is a (possibly) probabilistic algorithm executed by the server that takes as input the query token tk and the

encrypted multi-map EMM. It outputs the response res to the client.

- $\vec{v} \leftarrow \mathsf{Result}(K, \mathsf{key}, \mathsf{res})$ is a deterministic algorithm executed by the client that takes as input the private key K and the queried key which produced tk, and the response res due to tk. It outputs a value tuple $\vec{v}$ associated with key.
-

Security is captured under the real/ideal simulation paradigm with leakage functions $\mathcal{L} = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q})$. It guarantees that the encrypted multi-maps structure EMM from Setup reveals nothing beyond the setup leakage $\mathcal{L}_\mathsf{S}$, and the query procedure over EMM reveals nothing beyond the query leakage $\mathcal{L}_\mathsf{Q}$. More precisely, Reply is the only algorithm during the query procedure $(\mathsf{Query}, \mathsf{Reply}, \mathsf{Result})$ that is executed by the server and possibly leaks information to it.

Definition 2 (Adaptive Security of Structured Encryption for MMs). Let $\Sigma = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Reply}, \mathsf{Result})$ be a non-interactive response-hiding structured encryption scheme for multi-maps. We say that Σ is $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q})$-adaptively-secure, if for all PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

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

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

$\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda)$: $\mathcal{A}$ generates an input multi-map MM and receives EMM from the challenger, where $(K, \mathsf{st}, \mathsf{EMM}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{MM})$. Then $\mathcal{A}$ adaptively makes a polynomial number of queries. For each queried key, the challenger executes $(\mathsf{Query}, \mathsf{Reply})$ with $\mathcal{A}$, and $\mathcal{A}$ receives

$\mathsf{tk} \leftarrow \mathsf{Query}(K, \mathsf{st}, \mathsf{key})$.

Ideal$_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda)$ $\mathcal{A}$ generates an input multi-map MM and receives EMM from the simulator $\mathcal{S}$, where EMM is generated using only $\mathcal{L}_\mathsf{S}$. Then $\mathcal{A}$ adaptively makes a polynomial number of queries. For each queried key, the simulator $\mathcal{S}$ executes $(\mathsf{Query}, \mathsf{Reply})$ with $\mathcal{A}$ by generating tk using only $\mathcal{L}_\mathsf{Q}$.

In both experiments, $\mathcal{A}$ returns a bit $b \in \{0, 1\}$ eventually. Looking ahead, our construction can be proven secure under the adaptive definition, but an active adversary, in principle, can manipulate the leakages to be defined below.

2.3 Typical Leakage Functions

We recall typical leakage functions summarized for encrypted multi-maps [15].

- *Query equality pattern* qeq, usually referred to as the search pattern of SSE, reports whether two queries are issued for the same key or not. Formally, for a sequence of *t* non-cryptographic keys $\mathsf{qeq}(\mathsf{key}_1, \ldots, \mathsf{key}_t) = M$, where *M* is a $t \times t$ matrix such that $M[i][j] = 1$ if $\mathsf{key}_i = \mathsf{key}_j$ for $i, j \in [t]$.
- *Response length pattern* rlen reports the number of values associated with an input key. Formally, for a multi-map $\mathsf{MM} = \{(\mathsf{key}_i, \vec{v}_i)\}_{\mathsf{key}_i \in \mathcal{K}}$ with key space $\mathcal{K}$, $\mathsf{rlen}(\mathsf{MM}, \mathsf{key}_i) = \ell(\mathsf{key}_i) = |\vec{v}_i|$. This is the leakage we aim to avoid.
- *Maximum response length pattern* mrlen reports the maximum number of values associated with any key. Formally, for a multi-map

MM with key space $\mathcal{K}$, $\mathsf{mrlen}(\mathsf{MM}) = \ell = \max_{\mathsf{key} \in \mathcal{K}} \ell(\mathsf{key})$, where $\ell(\mathsf{key})$ is the number of values associated with key.

– *Data size pattern* dsize reports the total number of values over all keys. Formally, for a multi-map MM with key space $\mathcal{K}$, $\mathsf{dsize}(\mathsf{MM}) = n = \sum_{\mathsf{key} \in \mathcal{K}} \ell(\mathsf{key})$.

We further introduce the access intersection pattern aintx as follows.

– *Access intersection pattern* aintx reports the intersection of EMM slots accessed by two queries. Formally, for a sequence of *t* keys, $\mathsf{aintx}(\mathsf{EMM}, \mathsf{key}_1, \ldots, \mathsf{key}_t) = M'$, where M' is a $t \times t$ matrix such that $M'[i][j]$ contains the common slots of EMM accessed when querying key_i and key_j for $i, j \in [t]$.

2.4 Volume Hiding Against Passive Adversaries

Volume hiding requires that the number of values associated with any single key remains private. Patel *et al.* [27] defined it as a property of the leakage functions in a game where the active adversary receives the setup and query leakage of a chosen multi-map. We instead consider passive adversaries who can observe all EMM accesses but cannot actively perform adversarially chosen accesses. Furthermore, they have no knowledge of the plaintext multi-map used to generate EMM. An EMM that is volume-hiding against passive adversaries means they cannot determine the actual volume of any key in the multi-map.

An active adversary has a high level of "control" of the data owner who keeps explicitly authorizing many queries, which an actively malicious cloud can exploit toward exposing the entire multi-map. Our model is weaker; however, it remains meaningful in a realistic setting that the cloud server cannot actively sabotage the data owner or would not risk being caught. This also matches the required strength of the adversary in the very recent volume-abusing attacks [21].

2.5 Hash Ring or Consistent Hashing

Hash ring [17] is best known as a solution to the load-balancing problem (*e.g.*, Chord [29] for peer-to-peer protocols). To assign a set of objects to some servers, the hash function, taking either (the identifier of) an object or a server as input, outputs *s* bits as a slot-identifier in an identifier ring modulo 2^s (*i.e.*, 0 to $2^s - 1$). For large *s*, the probability of the collision of identifiers is negligible. Each object will be assigned to the server that is placed into the first slot clockwise from it.

---

3 Space-Efficient Volume-Hiding Multi-map Encryption

3.1 Overview

A classical hash-ring application (Fig. 1a) assigns a set of objects to some servers by placing both objects and servers to some ring slots. The object is then assigned to the next server that appears on the ring in clockwise order, also dubbed as a *successor* of the object slot. *Successor discovery* thus refers to finding the next occupied slot. In our scenario (Fig. 1b), we pick a slot (red node) for each *value index of a key* via hashing. For each interval bounded by red nodes, a slot for storing *the key-value tuple* (blue node) is *randomly* picked.

For any key with a volume less than the maximum volume ℓ, we hash each *non-existent (fake) value index* (up to ℓ) to a slot (red ring) that lies in a distinct interval bounded by the slots of two key-value tuples (blue nodes). It means no other fake value index (red ring) or real value index of the same key (red node) is allowed in the interval (but value indexes for any other key are fine). We may need to re-sample these slots to place them accordingly, yet our analysis shows it is unlikely to happen. To finish the EMM setup, the client outsources the ciphertext of each key-value tuple and its ring slot location (blue node). Notably, the server does not need to store the hash ring (*cf.*, the entire hash table is outsourced by dprfMM / dpMM [27]) or any dummies! Our construction thus requires less storage than any existing solutions.

To query for a key, the client issues a token of ℓ ring slots related to it. The server then carries out successor discovery for each slot from the token and returns the ciphertext stored in the successor. Due to the slot arrangement in the setup, the ciphertexts surely contain all the real values associated with the queried key, and there will be one and only one ciphertext (either associated with the queried key or another key) for each of the ℓ slots of any key, thus maintaining the same response volume (*i.e.*, ℓ) for any key.

Our communication overhead is ℓ, which is as good as all prior schemes (except dpMM [27] with possible data loss). Successor discovery takes $\Theta(\log n)$ with the binary search. It can be accelerated to $\Theta(\log\log n)$ with advanced structures [32] at the cost of extra $\Theta(n)$ storage, providing a trade-off.

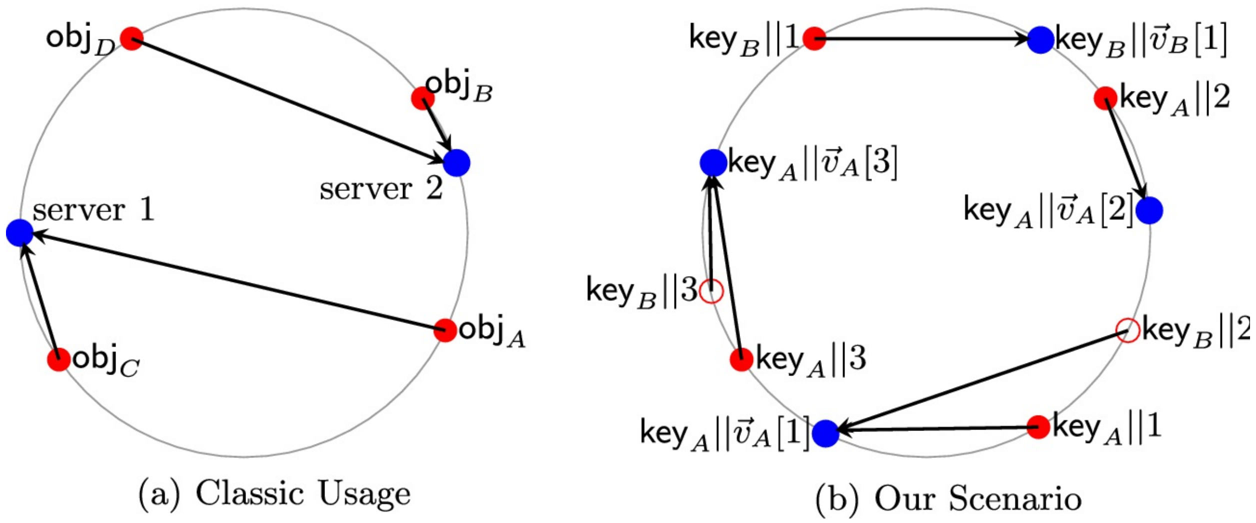

Fig. 1. Consistent hashing and its application in load balancing and our EMM (Color figure online)

Figure 1b depicts a sample EMM of $\{(\mathsf{key}_A, \vec{v}_A), (\mathsf{key}_B, \vec{v}_B)\}$ with $|\vec{v}_A| = 3$, $|\vec{v}_B| = 1$, and the maximum volume $\ell = |\vec{v}_A| = 3$. Each key and its value index, *i.e.*, $(\mathsf{key}_A, 1)$, $(\mathsf{key}_A, 2)$, $(\mathsf{key}_A, 3)$, and $(\mathsf{key}_B, 1)$ is mapped to a ring slot (red node). As the volume of key_B is less than ℓ, two extra slots (red ring) are derived by hashing $(\mathsf{key}_B, 2)$ and $(\mathsf{key}_B, 3)$ in a way that the slots of key_B are ensured to have distinct successors ($\mathsf{ct}_2 = \mathsf{Enc}(\mathsf{key}_A \| \vec{v}_A[1])$ and $\mathsf{ct}_3 = \mathsf{Enc}(\mathsf{key}_A \| \vec{v}_A[3])$ in our example). To query for key_B, the token consists of $\ell = 3$ slots related to key_B in a randomly permuted order. The server fetches the successor of each slot and returns the corresponding ciphertext, *i.e.*, $\{\mathsf{Enc}(\mathsf{key}_B \| \vec{v}_B[1]), \mathsf{ct}_2, \mathsf{ct}_3\}$ (an unordered set). After client decryption, $\vec{v}_B[1]$ will be identified as the result.

```
– $(K, \mathsf{st}, \mathsf{EMM}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{MM} = \{\mathsf{key}_i, \vec{v}_i\}_{i\in[m]})$
   1. Sample PRF seed $K_{\mathsf{PRF}} \leftarrow\$ \{0,1\}^\lambda$ and encryption key $K_{\mathsf{SKE}} \leftarrow\$ \{0,1\}^\lambda$.
   2. Initialize array $\mathsf{HR}$ of size $2^s$ and arrays $\mathsf{U}, \mathsf{V}$ of size $n$.
   3. Initialize two dictionaries $\mathsf{DX}_C, \mathsf{DX}_S$.
   4. For each $i \in [m]$:
      (a) Sample $\mathsf{r}_i \leftarrow\$ \{0,1\}^\lambda$.
      (b) For each $j \in [|\vec{v}_i|]$,
            i. Insert $F_{K_{\mathsf{PRF}}}(\mathsf{r}_i||j)$ into $\mathsf{U}$.
           ii. Set $\mathsf{HR}[F_{K_{\mathsf{PRF}}}(\mathsf{r}_i||j)] \leftarrow (\mathsf{key}_i, \vec{v}_i[j])$.
      (c) Set $\mathsf{DX}_C[\mathsf{key}_i] \leftarrow (|\vec{v}_i|, \mathsf{r}_i)$.
   5. Sort elements in $\mathsf{U}$ in ascending order.
   6. For each $i \in [n]$:
      (a) Sample $t$ from $[\mathsf{U}[i-1]+1\,..\,\mathsf{U}[i \bmod n]]$.
      (b) Set $\mathsf{V}[i-1] \leftarrow t$.
      (c) Set $\mathsf{DX}_S[\mathsf{V}[i-1]] \leftarrow \mathsf{Enc}(K_{\mathsf{SKE}}, \mathsf{HR}[\mathsf{U}[i-1]])$.
   7. Sort elements in $\mathsf{V}$ in ascending order.
   8. For each $i \in [m]$:
      (a) Sample $\mathsf{r}'_i \leftarrow\$ \{0,1\}^\lambda$, and set $(t_1, \ldots, t_\ell) =$

          $\{F_{K_{\mathsf{PRF}}}(\mathsf{r}_i||1), \ldots, F_{K_{\mathsf{PRF}}}(\mathsf{r}_i||(|\vec{v}_i|)), F_{K_{\mathsf{PRF}}}(\mathsf{r}'_i||(1+|\vec{v}_i|)), \ldots, F_{K_{\mathsf{PRF}}}(\mathsf{r}'_i||\ell)\}$.

      (b) If there exist distinct indices $j, k \in [\ell]$ such that $\mathsf{FindSuccessor}(t_j, \mathsf{V}) =$
          $\mathsf{FindSuccessor}(t_k, \mathsf{V})$, redo Step 8a, which includes the sampling of $\mathsf{r}'_i$.
      (c) Set $\mathsf{DX}_C[\mathsf{key}_i] \leftarrow \mathsf{DX}_C[\mathsf{key}_i]||\mathsf{r}'_i$.
   9. Delete $\mathsf{HR}, \mathsf{U}$.
  10. Return $K \leftarrow (K_{\mathsf{PRF}}, K_{\mathsf{Enc}})$, $\mathsf{st} \leftarrow \mathsf{DX}_C$, and $\mathsf{EMM} \leftarrow (\mathsf{DX}_S, \mathsf{V})$.
– $\mathsf{tk} \leftarrow \mathsf{Query}(K, \mathsf{st}, \mathsf{key})$
   1. Parse $K$ as $(K_{\mathsf{PRF}}, K_{\mathsf{Enc}})$ and $\mathsf{st}$ as $\mathsf{DX}_C$.
   2. Parse $\mathsf{DX}_C[\mathsf{key}]$ as $(|\vec{v}|, \mathsf{r})||\mathsf{r}'$.
   3. Set $\mathsf{tk} \leftarrow \{F_{K_{\mathsf{PRF}}}(\mathsf{r}||j)\}_{j\in[|\vec{v}|]} \cup \{F_{K_{\mathsf{PRF}}}(\mathsf{r}'||(j+|\vec{v}|))\}_{j\in[\ell-|\vec{v}|]}$.
   4. Permute elements in $\mathsf{tk}$.
   5. Return $\mathsf{tk}$.
– $\mathsf{res} \leftarrow \mathsf{Reply}(\mathsf{tk}, \mathsf{EMM})$
   1. Parse $\mathsf{EMM}$ as $(\mathsf{DX}_S, \mathsf{V})$.
   2. Initialize an empty set $\mathsf{res}$.
   3. For each $t \in \mathsf{tk}$, set $\mathsf{res} \leftarrow \mathsf{res} \cup \mathsf{DX}_S[\mathsf{FindSuccessor}(t, \mathsf{V})]$.
   4. Return $\mathsf{res}$.
– $\vec{v} \leftarrow \mathsf{Result}(K, \mathsf{key}, \mathsf{res})$
   1. Parse $K$ as $(K_{\mathsf{PRF}}, K_{\mathsf{Enc}})$ and $\mathsf{res}$ as $\{\mathsf{ct}_i\}_{i\in[\ell]}$.
   2. Initialize an empty array $\vec{v}$.
   3. For $i \in [\ell]$:
      (a) Set $(\mathsf{key}', \mathsf{value}') \leftarrow \mathsf{Dec}(K_{\mathsf{Enc}}, \mathsf{ct}_i)$.
      (b) If $\mathsf{key}' = \mathsf{key}$, insert $\mathsf{value}'$ into $\vec{v}$.
   4. Return $\vec{v}$.
```

Fig. 2. Efficient volume-hiding encryption for multi-maps

3.2 Description

Figure 2 describes the details of our construction. Let $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^s$ be a pseudorandom function, and $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$ be an RCPA-secure encryption scheme. $\mathsf{FindSuccessor}$ is a simple function for successor discovery in the hash ring. It takes as input an integer and a sorted array, and outputs the smallest value in the array that is larger than the input integer. If the integer is larger than any value in the array, it outputs the minimum of the array.

Setup. *Cryptographic Keys.* The client randomly picks a PRF seed K_{PRF} and an SKE key K_{SKE} as the secret (cryptographic) key *K*.

Data Structure. Let *n* be the number of values over all (non-cryptographic) keys of the MM. The client initializes arrays $\mathsf{U}[0..n-1], \mathsf{V}[0..n-1], \mathsf{HR}[0..2^s-1]$ for the hash ring, and dictionaries $\mathsf{DX}_C, \mathsf{DX}_S$, which are all initially empty.

Client State DX_C. For each key_i with $i \in [m]$, the client samples a λ-bit randomness r_i. $\mathsf{DX}_C[\mathsf{key}_i]$ stores the volume $|\vec{v}_i|$ and the randomness r_i of key_i.

Hash Ring HR. For each value $\vec{v}_i[j]$ associated with key_i, the client determines a hash ring slot by $F_{K_{\mathsf{PRF}}}(\mathsf{r}_i \| j)$, puts it in some unoccupied slot of U, and puts the key-value tuple $(\mathsf{key}_i, \vec{v}_i[j])$ in $\mathsf{HR}[F_{K_{\mathsf{PRF}}}(\mathsf{r}_i \| j)]$.

Red Nodes U *as Boundaries for Ciphertexts.* Eventually, *n*-out-of-2^s slots of HR are occupied. These locations stored in U are then sorted in ascending order.

Blue Nodes V *for Ciphertexts.* For each $i \in [n]$, the client randomly samples an integer t from $[\mathsf{U}[i-1]+1\,..\,\mathsf{U}[i \bmod n]]$, with the abused notation for $i=n$ where $[\mathsf{U}[n-1]+1\,..\,\mathsf{U}[0]] = [\mathsf{U}[n-1]+1\,..\,2^s-1] \cup [0\,..\,\mathsf{U}[0]]$. The client sets $\mathsf{V}[i-1]$ as t, and stores in $\mathsf{DX}_S[t]$ an encryption of the key-value tuple at $\mathsf{HR}[\mathsf{U}[i-1]]$.

Red Rings for Generating Token tk. Corresponding to each key_i for $i \in [m]$, the client samples another λ-bit randomness r'_i, and outputs a tuple $(t_1, \ldots, t_\ell)$:

$$\{F_{K_{\mathsf{PRF}}}(\mathsf{r}_i \| 1), \ldots, F_{K_{\mathsf{PRF}}}(\mathsf{r}_i \| (|\vec{v}_i|)), F_{K_{\mathsf{PRF}}}(\mathsf{r}'_i \| (1+|\vec{v}_i|)), \ldots, F_{K_{\mathsf{PRF}}}(\mathsf{r}'_i \| \ell)\}$$

where the first $|\vec{v}_i|$ elements are key_i-related slots in HR, and the rest are “fake.”

Note that r'_i needs to be chosen in the way that any distinct $j, k \in [\ell]$ satisfy $\mathsf{FindSuccessor}(t_j, \mathsf{V}) \neq \mathsf{FindSuccessor}(t_k, \mathsf{V})$. Otherwise, we re-sample a new r'_i to fulfill the condition[2]. We show that the client does not need to re-sample with the probability tending to 1 in Sect. 3.3. The client concatenates r'_i to $\mathsf{DX}_C[\mathsf{key}_i]$.

The client deletes $(\mathsf{HR}, \mathsf{U})$, sets DX_C as st, and outsources $(\mathsf{DX}_S, \mathsf{V})$ as EMM.

Query. To query key, the client retrieves $(|\vec{v}|, \mathsf{r}) \| \mathsf{r}'$ from $\mathsf{DX}_C[\mathsf{key}]$, and computes:

$$\mathsf{tk} = \{\{F_{K_{\mathsf{PRF}}}(\mathsf{r} \| j)\}_{j \in [|\vec{v}|]}, \{F_{K_{\mathsf{PRF}}}(\mathsf{r}' \| (j + |\vec{v}|))\}_{j \in [\ell - |\vec{v}|]}\}.$$

The client permutes elements in tk before sending it to the server.

For each element $t \in \mathsf{tk}$, the server figures out its successor of the hash ring by $t' = \mathsf{FindSuccessor}(t, \mathsf{V})$ and retrieves the ciphertext stored in $\mathsf{DX}_S[t']$. The collection of these ciphertexts is returned as the query response res.

The client decrypts the ciphertexts in res with K_{SKE}, and collects all decrypted values associated with key as the query result $\vec{v}$.

—

3.3 Analysis

Security. For Setup, the server receives the EMM, which contains the ciphertexts of all key-value tuples and randomly sampled hash-ring slots. Thus, the setup leakage $\mathcal{L}_{\mathsf{S}}$ only contains the data size dsize.

For Query, the server will know ℓ, the maximum volume of the input multi-map by the size of any query token. It could determine the repetition of the same query and the common slots accessed by different queries. Thus, the query leakage $\mathcal{L}_{\mathsf{Q}}$ contains the query-equality pattern qeq, the access intersection pattern aintx, and the *maximum* response length $\mathsf{mrlen} = \ell$. Formally, Theorem 1 (proven in Appendix A) asserts the adaptive security of our construction with the leakage function $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}) = (\mathsf{dsize}, (\mathsf{qeq}, \mathsf{aintx}, \mathsf{mrlen}))$.

Theorem 1 If $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$ is an RCPA-secure encryption scheme and *F* is a pseudorandom function, our construction is an $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}})$-adaptive-secure encryption scheme for multi-maps, where $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}) = (\mathsf{dsize}, (\mathsf{qeq}, \mathsf{aintx}, \mathsf{mrlen}))$.

Nevertheless, an active adversary with the knowledge of EMM could adaptively issue queries and try to match the corresponding access intersection aintx with any prior background information about the actual volume. This seems to be inherent when we “reuse” real slots for realizing volume-hiding EMM efficiently. Passive adversaries cannot exploit such kinds of strategies. Intuitively, the response length rlen for any key remains hidden since the slots for different keys are pseudorandom as our scheme decides them via pseudorandom functions.

No Re-sampling Probability. Our construction requires each of the last $\ell - |\vec{v}_i|$ elements of $(t_1, \ldots, t_\ell)$ for key_i, *i.e.*, $F_{K_{\mathsf{PRF}}}(\mathsf{r}'_i \| (1 + |\vec{v}_i|)), \ldots, F_{K_{\mathsf{PRF}}}(\mathsf{r}'_i \| \ell)$, has a unique successor. For ease of analysis, we model *F* as a random function. Obviously, the elements within the same interval will share the same successor, and the probability that an element is assigned to an interval by the random function depends on the interval length. Suppose that we divide the hash ring of length 2^s into *n* intervals with *n* values among $\{0,1\}^s$ in V. Let c_i be the length of the interval where t_i is placed for $i \in [\ell]$. Consider the case that the first $k-1$ elements of $(t_{|\vec{v}_i|+1}, \ldots, t_\ell)$ belong to distinct intervals. The probability that the *k*-th element will be assigned to an interval different from any previous elements is $(1 - \frac{c_1 + \cdots + c_{|\vec{v}_i|+k-1}}{2^s})$. Then, the probability that the remaining $\ell - |\vec{v}_i|$ elements are all placed in distinct intervals, *i.e.*, they have distinct outputs of $\mathsf{FindSuccessor}$ and no re-sampling is needed for key_i, is:

$$p = \prod_{k=1}^{\ell - |\vec{v}_i|} (1 - \frac{c_1 + \cdots + c_{|\vec{v}_i|+k-1}}{2^s}) \geq (1 - \frac{(\ell - |\vec{v}_i|) \times c_{\mathsf{avg}}}{2^s})^{\ell - |\vec{v}_i|}$$

where c_{avg} is the average length of $(c_1, \ldots, c_{\ell-1})$. We take the expected interval length of the hash ring for c_{avg}, *i.e.*, $\frac{2^s}{n}$. The probability becomes $(1 - \frac{\ell - |\vec{v}_i|}{n})^{\ell - |\vec{v}_i|}$, which is approximate to $e^{-\frac{(\ell - |\vec{v}_i|)^2}{n}}$. It tends to 1 assuming[3] $\ell = o(n)$.

Efficiency. Our construction avoids padding of dummies or storing the entire hash ring. Consider an input multi-map MM with maximum volume ℓ, *m* distinct keys, and *n* key-value tuples. The server stores a dictionary DX_S of size *n* for all key-value tuples of MM and an auxiliary array V of size *n* to quickly determine the successor of any slot, outperforming the state-of-the-art dprfMM and dpMM [27]. The client stores the state of size *m* rather than a stash for any parts of the input MM. It could be outsourced as *m* is independent of ℓ.

To query for any key, the client only needs to generate ℓ PRF values as slots in the token. The server determines exactly ℓ successors correspondingly for these ℓ slots. With binary search, the server can figure out a successor from V with $\mathsf{FindSuccessor}$ in $\Theta(\log n)$ time. The overall query complexity is thus $\Theta(\ell \cdot \log n)$, which is asymptotically the same as AVLH for general MMs. ℓ ciphertexts are returned to the client as a response. The communication overhead only falls behind the differentially private scheme [27] with possible data loss.

The prior arts we improved upon are practically efficient [27]. Moreover, unlike a particular existing scheme [14], the efficiency of our scheme is not affected by the data distribution. As stressed in Footnote 3, we only made one reasonable assumption of $\ell = o(n)$, which appears to be necessary for any practically-efficient volume-hiding EMM. (If the volume is in the order of magnitude of the database size, any volume-hiding scheme ought to be inefficient.) Our underlying cryptographic operations are mostly PRF as typically used in many practically-efficient STEs. It means that our asymptotic improvements over the prior arts can be directly translated to improvements in concrete efficiency, following the doctrine in computer science that asymptotic analysis of complexity remains meaningful, regardless of the everchanging computational power of the time.

Different Trade-Offs on Server. For $\mathsf{EMM} = (\mathsf{DX}_S, \mathsf{V})$ outsourced to the server, the dictionary keys in DX_S are essentially the elements stored in the auxiliary array V. Our construction can choose to further outsource V or directly process over DX_S, which improves the storage overhead to *n* at the cost of either communication overhead or computation overhead for the server.

With more advanced data structures, it is possible to reduce the computation overheads of the binary search for successors, which accelerates the query, while the server storage remains asymptotically the same, *i.e.*, $\Theta(n)$. For example, we could arrange V in a y-fast trie data structure [32], which efficiently supports successor queries for integers using extra storage. Concretely, this trade-off offers a computation overhead of $\Theta(\log \log n)$ for successor discovery with $\Theta(n)$ server-side overhead, while the performance of other criteria remains the same.

4 Volume-Hiding Dynamic EMM via Batching Updates

A dynamic EMM enables the client to insert or remove one or multiple value(s) associated with a key. Dynamic updates can be challenging to realize when we simultaneously consider security and efficiency (*e.g.*, minimizing leakage while maintaining a parallelly-traversable encrypted structure [23–25]). Similarly, we face a dilemma in designing a volume-hiding EMM. The root cause is that even a single update for a key-value pair may change the maximum volume of the multi-map. On one hand, the client should do dummy updates for *distinct* keys to hide whether the real one is associated with the maximum volume. On the other, it might leak more as those dummy updates for *distinct* keys it triggers probably cause extra leakage. For dynamic update, it sounds better to start from a static EMM that is free from dummies (or troubles), which guided our design.

Another way round is to let the client query the tuple regarding the key to be updated in advance, and re-upload the modified tuple, which is confined to the queried key but not any other keys. $\mathsf{VLH}^{\mathsf{d}}$ and $\mathsf{AVLH}^{\mathsf{d}}$ [14] follow this approach. However, they are still subject to the maximum-volume limit (unless those extra ones are truncated by the pseudorandom transform in $\mathsf{VLH}^{\mathsf{d}}$). They also fail to consider either forward or backward privacy emphasized in recent dynamic STE.

When saving space is one of our design goals, it is natural that the resulting EMM is "highly" optimized with respect to the maximum volume, which makes "volume-affecting" updates tricky. We thus consider dynamic EMM for batch updates [7]. Namely, we set up a new EMM instance for batches and periodically merge instances for efficiency. Setting up new instances for later merging may sound conceptually straightforward; however, it is not an approach practically applicable to all volume-hiding EMMs. As in Table 1, our static construction has decent communication overhead and server storage, which will be scaled up by the number of batches in such an approach (*e.g.*, for storing many EMM instances), making ours more applicable than others. This also aligns with the update feature of some commercial databases (*e.g.*, bulk-loading in Vertica [26]).

4.1 Definition

Syntax. We consider the batch of updates in a similar way as the multi-maps. The μ-th batch of updates MM_μ is defined as $\{\mu, \mathsf{key}_i, \vec{v}_i\}_{\mathsf{key}_i \in \mathcal{K}}$, where $\vec{v}_i$ now is a tuple of operation-value pairs regarding key_i. For each $j \in \ell_\mu(\mathsf{key})$, $\vec{v}[j] = (\vec{o}[j], \vec{u}[j])$, where $\vec{o}[j] \in \{\mathsf{add}, \mathsf{del}\}$ refers to insertion or deletion of value $\vec{u}[j]$ for key, and $\ell_\mu(\mathsf{key})$ is the number of updates associated with key in MM_μ. We treat the initial input multi-map for Setup as MM_0 with an insertion to be done.

We introduce the following update algorithm for non-interactive response-hiding structured encryption scheme for multi-maps Σ:

- $(K_\mu, \mathsf{st}', \mathsf{EMM}_\mu) \leftarrow \mathsf{Update}(\mathsf{MM}_\mu, \mathsf{st})$ is a probabilistic algorithm executed by the client that takes as input a batch of updates MM_μ and a state st. It outputs a private key K_μ, an updated state st', and the encrypted multi-map EMM_μ. The client stores (K_μ, st') while sending EMM_μ to the server.

We extend the leakage function to $\mathcal{L} = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U})$, where $\mathcal{L}_\mathsf{U}$ is the update leakage. STE security guarantees that Update reveals no information beyond $\mathcal{L}_\mathsf{U}$.

The definition of adaptive security follows Definition 2, except that the simulator answers the update request on MM_μ from $\mathcal{A}$ with EMM_μ generated using $\mathcal{L}_\mathsf{U}$. Volume hiding is still considered with respect to passive adversaries.

Dynamic STE schemes [1, 3, 4, 25] attach great importance to forward privacy, which requires updates to reveal nothing on the updated key, and backward privacy, which ensures the deleted value remains inaccessible to the server. While a recent work [28] pointed out that forward-private STE still suffers from subtle attacks, forward privacy could mitigate the severe adaptive file injection attacks [33], thus becoming the *de facto* standard of dynamic STE nowadays. We formalize forward and backward privacy for batch updates in Appendix B.

4.2 Description

For the μ-th update batch $\mathsf{MM}_\mu = \{\mu, \mathsf{key}_i, \vec{v}_i = (\vec{o}_i, \vec{u}_i)\}_{\mathsf{key}_i \in \mathcal{K}}$, the client generates $(K_\mu, \mathsf{st}', \mathsf{EMM}_\mu)$ similar to Setup in Fig. 2. The new state is concatenated to the old one according to keys. The only difference is that, for each $\mathsf{key} \in \mathcal{K}$ and each $j \in \ell_\mu(\mathsf{key})$, $(\mathsf{key}, \vec{o}[j], \vec{u}[j])$ is encrypted instead of a key-value pair.

Suppose the client has issued *b* batches of updates, and the server stores $b+1$ encrypted multi-maps, including the initial one in Setup. Upon a query, the client issues $b+1$ tokens with Query, and the server answers with Reply proposed in our static construction, respectively. The client removes those values marked with del and returns the remaining values associated with the queried key.

4.3 Analysis

Efficiency. For an update batch MM_μ, let n_μ be the total number of updates over all keys, m_μ be the number of distinct keys, and ℓ_μ be the maximum number of updates for any key. To update with MM_μ, the server stores EMM_μ of size $2n_\mu$. It takes the client $\Theta(m_\mu \cdot \ell_\mu)$ to generate this EMM_μ.

For a query over our EMM with *b* batches of updates, the communication overhead is $\sum_{\mu=0}^{b} \ell_\mu$, while the computation complexity is $\Theta(\sum_{\mu=0}^{b} \ell_\mu \cdot \log n_\mu)$.

Note that our dynamic extension inherits the nice properties of the underlying static EMM, with neither client stash for overflowed multi-maps nor data loss.

For amortized efficiency, we could exploit the periodic-merge approach of Vertica [26] as also adopted by a prior STE [7]. The client decides a consolidation step *s* and the EMM for each update batch is organized as leaves of an *s*-ary tree, created bottom-up. When *s* nodes are occupied in the same level, the client downloads these *s* EMMs, consolidates existing values associated with the same keys, and re-generates an encrypted multi-map, which will be assigned as the parent of these *s* nodes. We refer to [7, 26] for a more detailed description.

Despite the periodic cost for consolidation, this optimization can save the server-side storage by reclaiming the space for deleted values, and reduce the query overhead when the consolidation does not increase the maximum volume. The actual consolidation strategy can be different for different scenarios.

Security. We first introduce two new leakage functions for batch updates.

- *Update-batch pattern* updb indicates the batches with any updates on key happened. Formally, $\mathsf{updb}(\mathsf{MM}, \mathsf{key}) = \{\mu \mid \ell_\mu(\mathsf{key}) > 0\}$ with $\mathsf{MM} = \cup \mathsf{MM}_\mu$.
- *Update size pattern* usize reports the total number of updates over all keys in an update batch. Formally, for a batch of updates MM_μ with (non-cryptographic) key space $\mathcal{K}$, $\mathsf{usize}(\mathsf{MM}_\mu) = \sum_{\mathsf{key} \in \mathcal{K}} \ell_\mu(\mathsf{key})$.

Theorem 2 If $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$ is an RCPA-secure encryption scheme and *F* is a PRF, our dynamic construction is an $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U})$-adaptive-secure dynamic encryption scheme for multi-maps with forward privacy, level-*II* backward privacy, and volume hiding against passive adversaries where $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U}) = (\mathsf{dsize}, (\mathsf{qeq}, \mathsf{aintx}, \mathsf{mrlen}, \mathsf{updb}), \mathsf{usize})$.

The proof is omitted due to the page limit and since it can be done in a standard way. Roughly, given usize, $\mathsf{Update}()$ can be simulated similar to simulating $\mathsf{Setup}()$ as in Theorem 1. The update leakage contains usize of the update batch, not revealing the volume associated with any key against passive adversaries.

The construction is forward-private as the token generated for previous encrypted multi-maps could not be applied to any subsequent updates and no key information is revealed during updates. It also satisfies our level-II backward privacy for batch updates (Appendix B) as the construction is response-hiding and nothing except the batches is revealed when updates over the queried key.

5 Conclusion

Structured encryption is volume hiding when each query has the same volume as the largest one. It was originally proposed as a stringent requirement and might be perceived as a theoretical notion with no practical realization.

Motivated by passive attacks abusing volume leakage in plaintext reconstruction [21], this work considers a practically relevant setting in which the cloud is honest-but-curious and does not launch active attacks, which could alert the data owner. Achieving the volume-hiding property remains non-trivial given our primary efficiency goal is to save the actual volume of encrypted data. The novelty of our scheme is that the slots needed in hiding the volume across the queries are all real slots storing some meaningful ciphertexts, in contrast to the known paradigm of volume-hiding structured encryption. Thanks to its nice performance and compatibility with batch updates, we extend it to the first dynamic volume-hiding encrypted multi-maps with forward and backward privacy.

Conceptually, this work brings new ideas to the (volume-hiding) structured encryption realm and hopefully inspires more research. We leave as an open problem to provide volume-hiding against stronger adversaries and handle sporadic volume-affecting updates while maintaining a similar efficiency level.

A Security Proof for Theorem 1

Proof We construct the simulator $\mathcal{S}$ as follows.

To simulate EMM with $\mathcal{L}_\mathsf{S} = \mathsf{dsize}$, the simulator $\mathcal{S}$ initializes two arrays V, U of size $n = \mathsf{dsize}(\mathsf{MM})$, and a dictionary $\mathsf{DX}_\mathcal{S}$. $\mathcal{S}$ fills each slot of U with a uniformly random value sampling from $\{0,1\}^s$, and then sorts U. For $i \in [n]$, $\mathcal{S}$ randomly selects a uniformly random integer t as

$\mathsf{V}[i-1]$ from the interval $[\mathsf{U}[i-1]+1\,..\,\mathsf{U}[i \bmod n]]$ and sets $\mathsf{DX}_S[\mathsf{V}[i-1]]$ as a uniformly random value sampling from $\{0,1\}^\lambda$. $\mathcal{S}$ sorts and keeps V. $(\mathsf{DX}_S, \mathsf{V})$ is returned as EMM.

To simulate tk for the *i*-th query of key_i with $\mathcal{L}_\mathsf{Q} = (\mathsf{qeq}, \mathsf{aintx}, \mathsf{mrlen})$, the simulator $\mathcal{S}$ checks whether key_i has been queried before using $\mathsf{qeq}(\mathsf{key}_1, \ldots, \mathsf{key}_i)$. If so, $\mathcal{S}$ returns the same tk as that of the previous queries of key_i. Otherwise, $\mathcal{S}$ checks $\mathsf{aintx}(\mathsf{EMM}, \mathsf{key}_1, \ldots, \mathsf{key}_i)$ and get the common slots of key_i with previous queries. $\mathcal{S}$ picks $\ell = \mathsf{mrlen}(\mathsf{MM})$ intervals in V, with parts of them determined by the common slots from aintx and the rest picked randomly without repeating, and samples a uniformly random value from each selected interval. These ℓ values are returned as tk.

We show that for all PPT adversary $\mathcal{A}$, the outputs of the real-world game and ideal-world game are indistinguishable. We derive a standard game sequence from the real-world game $\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda)$ to the ideal-world game $\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda)$.

- $\mathbf{Game}_0$ is the same as $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$.
- $\mathbf{Game}_1$ replaces the pseudorandom function *F* in $\mathbf{Game}_0$ with a random function (and recalled when needed).
- $\mathbf{Game}_2$ replaces the RCPA-secure encryption scheme in $\mathbf{Game}_1$ with a random function.
- $\mathbf{Game}_3$ replaces the outputs of random functions in $\mathbf{Game}_2$ with values chosen uniformly at random.
- $\mathbf{Game}_4$, for any query, randomly picks $\ell = \mathsf{mrlen}(\mathsf{MM})$ intervals in V

and samples a uniformly random value from each selected interval. $\mathbf{Game}_4$ is the same as $\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda)$.

$\mathbf{Game}_0$ and $\mathbf{Game}_1$ are indistinguishable; otherwise, it violates the security of the pseudorandom function. $\mathbf{Game}_1$ and $\mathbf{Game}_2$ are indistinguishable; otherwise, it violates the RCPA security of the encryption scheme. By the definition of random functions, winning $\mathbf{Game}_2$ or $\mathbf{Game}_3$ shares an equal probability. The probabilities of winning $\mathbf{Game}_3$ and $\mathbf{Game}_4$ are also equal since DX_S, V, tk follow the same distributions in both games. By combining these (in)equalities, we have $|\Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$.

B Forward and Backward Privacy for Batch Updates

Forward privacy for batch updates requires that any batch of updates reveals nothing about the keys to be updated. Thus, the adversary cannot figure out the relation between newly updated multi-maps and any previous query. We extend the definition from that designed for a single key-value update [3].

Definition 3 (Forward Privacy). We say that an $\mathcal{L}$-adaptively-secure structured encryption scheme for multi-maps Σ over key space $\mathcal{K}$ is forward private, if the update leakage function $\mathcal{L}_\mathsf{U}$ can be written as $\mathcal{L}_\mathsf{U}(\mathsf{MM}_\mu) = \mathcal{L}'(\{\vec{v}_i\}_{\mathsf{key}_i \in \mathcal{K}})$, where $\mathsf{MM}_\mu = \{\mu, \mathsf{key}_i, \vec{v}_i\}_{\mathsf{key}_i \in \mathcal{K}}$, and $\mathcal{L}'$ is stateless.

Backward privacy hides deleted values during subsequent queries. Like the previous definition [4], we formalize it for *batch updates* by introducing leakage functions constructed from the union of historical update batches $\mathsf{MM} = \cup \mathsf{MM}_\mu$:

- *Value-batch pattern* valb reports the values currently associated with key and in which batches they are inserted. Formally,
  $\mathsf{valb}(\mathsf{MM}, \mathsf{key}) = \{(\mu, u) \mid (\mu, \mathsf{key}, (\mathsf{add}, u)) \in \mathsf{MM} \ \wedge \ \forall \mu', (\mu', \mathsf{key}, (\mathsf{del}, u)) \notin \mathsf{MM}\}$
  .
- *Delete-batch pattern* delb lists the batch-pairs of deletions and corresponding insertions on key. Formally,
  $\mathsf{delb}(\mathsf{MM}, \mathsf{key}) = \{(\mu, \mu') \mid \exists u \text{ s.t. } (\mu, \mathsf{key}, (\mathsf{del}, u)) \in \mathsf{MM} \ \wedge \ (\mu', \mathsf{key}, (\mathsf{add}, u)) \in \mathsf{MM}\}$
  .

Definition 4 (Backward Privacy). We say that an $\mathcal{L}$-adaptively-secure structured encryption scheme for multi-maps Σ over key space $\mathcal{K}$ is

- insertion-pattern revealing backward-private (Level-*I*) if
$$\mathcal{L}_{\mathsf{U}}(\mathsf{MM}_\mu) = \mathcal{L}'(\{\vec{o}_i\}_{\mathsf{key}_i \in \mathcal{K}}),$$
$$\mathcal{L}_{\mathsf{S}}(\mathsf{MM}, \mathsf{key}) = \mathcal{L}''(\mathsf{valb}(\mathsf{MM}, \mathsf{key}), \ell(\mathsf{key})),$$

- update-pattern revealing backward-private (Level-*II*) if
$$\mathcal{L}_{\mathsf{U}}(\mathsf{MM}_u) = \mathcal{L}'(\{\mathsf{key}_i, \vec{o}_i\}_{\mathsf{key}_i \in \mathcal{K}}),$$
$$\mathcal{L}_{\mathsf{S}}(\mathsf{MM}, \mathsf{key}) = \mathcal{L}''(\mathsf{valb}(\mathsf{MM}, \mathsf{key}), \mathsf{updb}(\mathsf{MM}, \mathsf{key})),$$

- weakly backward-private (Level-*III*) if
$$\mathcal{L}_{\mathsf{U}}(\mathsf{MM}_u) = \mathcal{L}'(\{\mathsf{key}_i, \vec{o}_i\}_{\mathsf{key}_i \in \mathcal{K}}),$$
$$\mathcal{L}_{\mathsf{S}}(\mathsf{MM}, \mathsf{key}) = \mathcal{L}''(\mathsf{valb}(\mathsf{MM}, \mathsf{key}), \mathsf{delb}(\mathsf{MM}, \mathsf{key})),$$

where $\mathsf{MM} = \cup \mathsf{MM}_\mu$ with $\mathsf{MM}_\mu = \{\mu, \mathsf{key}_i, \vec{v}_i = (\vec{o}_i, \vec{u}_i)\}_{\mathsf{key}_i \in \mathcal{K}}$. $\ell(\mathsf{key})$ is the volume of key in MM. $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

References

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

2. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: Network and Distributed System Security Symposium (NDSS) (2020)

3. Bost, R.: $\sum o \varphi o \varsigma$: forward secure searchable encryption. In: ACM Conference on Computer and Communications Security (CCS), pp. 1143–1154 (2016)

4. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: ACM Conference on Computer and Communications Security (CCS), pp. 1465–1482 (2017)

5. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: ACM Conference on Computer and Communications Security (CCS), pp. 668–679 (2015)

6. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_33
[Crossref]

7. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.N.: Practical private range search revisited. In: ACM International Conference on Management of Data (SIGMOD), pp. 185–198 (2016)

8. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: beyond exact matches. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015, Part II. LNCS, vol. 9327, pp. 123–145. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24177-7_7
[Crossref]

9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
[MathSciNet][Crossref]

10. Grubbs, P., Lacharité, M., Minaud, B., Paterson, K.G.: Pump up the volume: practical database reconstruction from volume leakage on range queries. In: ACM Conference on Computer and Communications Security (CCS), pp. 315–331 (2018)

11. Gui, Z., Paterson, K.G., Patranabis, S., Warinschi, B.: SWiSSSE: system-wide security for searchable symmetric encryption. IACR Cryptology ePrint Archive: 2020/1328 (2020)

12. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption:

ramification, attack and mitigation. In: Network and Distributed System Security Symposium (NDSS) (2012)

13. Kamara, S., Moataz, T.: SQL on structurally-encrypted databases. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part I. LNCS, vol. 11272, pp. 149–180. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03326-2_6
[Crossref]

14. Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 183–213. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17656-3_7
[Crossref]

15. Kamara, S., Moataz, T., Ohrimenko, O.: Structured encryption and leakage suppression. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 339–370. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_12
[Crossref]

16. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM Conference on Computer and Communications Security (CCS), pp. 965–976 (2012)

17. Karger, D.R., Lehman, E., Leighton, F.T., Panigrahy, R., Levine, M.S., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Annual ACM Symposium on the Theory of Computing (STOC), pp. 654–663 (1997)

18. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: ACM Conference on Computer and Communications Security (CCS), pp. 1329–1340 (2016)

19. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: ACM Conference on Computer and Communications Security (CCS), pp. 669–684 (2013)

20. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. SIAM J. Comput. **39**(4), 1543–1561 (2009)
[MathSciNet][Crossref]

21. Kornaropoulos, E.M., Papamanthou, C., Tamassia, R.: Response-hiding encrypted ranges: revisiting security via parametrized leakage-abuse attacks. In: IEEE Symposium on Security and Privacy (S&P), pp. 750–767 (2021)

22. Lacharité, M., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. In: IEEE Symposium on Security and Privacy (S&P), pp. 297–314 (2018)

23. Lai, R.W.F., Chow, S.S.M.: Structured encryption with non-interactive updates and parallel traversal. In: International Conference on Distributed Computing Systems (ICDCS), pp. 776–777 (2015)

24. Lai, R.W.F., Chow, S.S.M.: Parallel and dynamic structured encryption. In: Deng, R., Weng, J., Ren, K., Yegneswaran, V. (eds.) SecureComm 2016. LNICST, vol. 198, pp. 219–238. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59608-2_12
[Crossref]

25. Lai, R.W.F., Chow, S.S.M.: Forward-secure searchable encryption on labeled bipartite graphs. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 2017. LNCS, vol. 10355, pp. 478–497. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61204-1_24
[Crossref]

26. Lamb, A., et al.: The vertica analytic database: C-store 7 years later. Proc. VLDB Endow. **5**(12), 1790–1801 (2012)
[Crossref]

27. Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: volume-hiding for multi-maps via hashing. In: ACM Conference on Computer and Communications Security (CCS), pp. 79–93 (2019)

28. Salmani, K., Barker, K.: Don't fool yourself with forward privacy, your queries still belong to us! In: ACM Conference on Data and Application Security and Privacy (CODASPY), pp. 131–142 (2021)

29. Stoica, I., Morris, R.T., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp. 149–160 (2001)

30. Wang, J., Chow, S.S.M.: Forward and backward-secure range-searchable symmetric encryption. IACR Cryptology ePrint Archive: 2019/497 (2019)

31. Wang, J., Du, M., Chow, S.S.M.: Stargazing in the dark: secure skyline queries with SGX. In: Nah, Y., Cui, B., Lee, S.-W., Yu, J.X., Moon, Y.-S., Whang, S.E. (eds.) DASFAA 2020, Part III. LNCS, vol. 12114, pp. 322–338. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59419-0_20
[Crossref]

32. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space theta(n). Inf. Process. Lett. (IPL) **17**(2), 81–84 (1983)
[Crossref]

33. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: USENIX Security Symposium, pp. 707–720 (2016)

Footnotes

1 Even powerful cloud/server-side security enclaves would not hide the communication volume.

A highlight of our scheme is its simplicity (fits in a single page) and its usage of lightweight tools (*e.g.*, PRF). No significant constant is hidden in the asymptotic evaluations. There are no (non-colluding) servers. These emphases might be random; however, we found the need to respond to a very negative review (not from DBSec) of an imagined paper, if not related subfields (volume-hiding SSE/ORAM/*etc.*).

2 A similar re-sampling is required in AVLH [14], while dprfMM and dpMM [27] abort when the stash of cuckoo hashing overflows due to collisions and evictions.

3 This is our only assumption on the input multi-map. AVLH [14] made more constraints on the multi-map for security or efficiency concerns, including this one.