Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption

Ghous Amjad* Sarvar Patel† Giuseppe Persiano‡ Kevin Yeo§ Moti Yung¶

June 10, 2021

Abstract

We study encrypted storage schemes where a client outsources data to an untrusted third-party server (such as a cloud storage provider) while maintaining the ability to privately query and dynamically update the stored data. We focus on *encrypted multi-maps*, a structured encryption (STE) scheme that stores pairs of label and value tuples that have several important applications (most notably, to searchable encryption and encrypted SQL databases). Encrypted multi-maps support queries for specific labels that return the associated value tuple. As responses are variable-length, encrypted multi-maps are subject to volume leakage attacks introduced by Kellaris *et al.* [CCS'16] with several follow-up works (including Grubbs *et al.* [CCS'18] and Lacharité *et al.* [S&P'18]). To prevent these attacks, *volume-hiding* encrypted multi-maps were introduced by Kamara and Moataz [Eurocrypt'19] that hide the volume of labels (i.e., the size of the associated value tuple).

As our main contribution, we present the first *fully dynamic* constructions of *volume-hiding* encrypted multi-maps that are both asymptotically and concretely efficient. Furthermore, our constructions simultaneously provide *forward and backward privacy* that are the de-facto standard security notions for dynamic STE schemes (Stefanov *et al.* [NDSS'14] and Bost [CCS'16]). Additionally, we implement our schemes to showcase their concrete efficiency. Our experimental evaluations show that our constructions are able to add dynamicity with minimal to no additional cost compared to the prior best static volume-hiding schemes of Patel *et al.* [CCS'20].

1 Introduction

Structured encryption (STE) schemes, introduced by Chase and Kamara [9], enable a client to outsource the storage of an encrypted version of their structured data to an untrusted third-party server (such as a cloud storage provider). The encrypted data is stored in a structured manner so that the client may still perform operations on the structured data without the server ever viewing the plaintext data. For privacy, the ideal goal is to ensure that the adversarial server does not learn any information about the outsourced data or operations performed by the client. Currently, this ideal privacy is only known to be achievable by using expensive cryptographic primitives such as fully homomorphic encryption or oblivious RAMs. Instead, STE schemes aim to strike a delicate balance between efficiency and privacy by enabling some leakage that is upper bounded by a well-defined and "sensible" leakage function to obtain efficiency that is necessary for real-world applications.

In our work, we focus on the *encrypted multi-map* (EMM) primitive that is an important example of a STE scheme that manage collections of pairs of *labels* and *value tuples* consisting of one or more values. Encrypted multi-maps form the basis of many important applications including searchable encryption [36]

*Brown University. ghous_amjad@brown.edu.
†Google. sarvar@google.com.
‡Università di Salerno. giuper@gmail.com.
§Google. kwlyeo@google.com.
¶Google. moti@google.com.

and encrypted databases [19]. Therefore, the construction of efficient and private EMMs is an important line of research to enable these applications. Throughout our work, we will focus on the dynamic variant of encrypted multi-maps that enable clients to update the encrypted data outsourced to the server.

While efficiency is clear to evaluate, assessing the level of privacy guaranteed by the leakage profile of an EMM (and STE schemes in general) is a challenging problem. So far, our only measure of privacy is a "sensible" or "reasonable" leakage function, which is both a vague and subjective qualifier. This has motivated the study of *leakage-abuse attacks* that aim at leveraging specific leakage profiles to compromise privacy. The first leakage-abuse attack was presented by Islam *et al.* [18]. Many follow up works [7, 26, 15, 21, 33, 38, 16] consider either different leakage profiles and/or weaker assumptions. These attacks significantly further our understanding of the dangers of various types of leakage profiles.

Volume-Hiding EMMs. One recent line of works has developed leakage-abuse attacks using volume leakage [21, 14, 3, 23, 17]. The seminal work of Kamara and Moataz [20] introduced the notion of *volume-hiding* encrypted multi-maps. These schemes ensure that the number of values (the volume) associated with any single label is never leaked to the adversary to protect against volume leakage attacks. This was subsequently improved by Patel *et al.* [31] that presented optimal constructions for *static* volume-hiding EMMs. Additionally, they considered the weaker notion of differentially private volume-hiding EMMs.

Most prior works focused on volume-hiding EMMs in the static setting where users are only able to query the outsourced, encrypted data. In many applications, dynamic EMMs are necessary where users are able to manipulate the outsourced encrypted multi-map by either adding, modifying or deleting pairs of label and value tuples. Kamara and Moataz [20] briefly studied dynamic volume-hiding EMMs but only present a construction offering a subset of natural update operations. Furthermore, these constructions are much less efficient than the static volume-hiding EMMs presented by Patel *et al.* [31]. In our work, we will help fill this gap by presenting dynamic EMMs offering all natural update operations while being more efficient.

Dynamic EMMs. Before presenting our dynamic volume-hiding EMM constructions, we first elaborate on the importance of enabling dynamicity for real-world usage. Consider any natural application of EMMs where the storage of highly sensitive data set is outsourced to a potentially untrusted cloud server (outsourcing is typically done for many reasons including fault tolerance and/or availability). Classic examples of highly sensitive data are personal health or financial information. In almost all of these applications, updating the outsourced encrypted data is almost immediately necessary. With medical data, any new information from patient examinations or lab tests must be stored in the EMM that require modifying the outsourced data. Similar updates are necessary for financial settings where the information from every new transaction must be propagated into the EMM. Even more importantly, it is straightforward to see that volume-hiding is also necessary in these scenarios. For medical data, the number of examinations for a patient is, typically, correlated to the current health status of the patient. Similarly, the number of financial transactions may correspond to that party's interest (or lack thereof) in the current market. By enabling dynamicity, our work will help open up volume-hiding EMMs to more practical applications.

From a technical perspective, there are significant difficulties when dealing with operations that enable updating the encrypted data even when ignoring volume-hiding requirements. At a high level, EMMs (and, generally, STE schemes) are attempting to find a delicate balance between functionality, efficiency and privacy. As dynamicity is increasing functionality, EMMs must ensure that only minimal loss of efficiency and/or privacy are incurred compared to the static setting. Due to this difficulty, there has been prior works that explored and defined necessary privacy requirements in dynamic settings to avoid privacy degradation incurred by update operations (such as being vulnerable to injection attacks [38, 3]). Formally, these standard notions for dynamic STE schemes are *forward* and *backward* privacy [37, 4, 5]. Forward privacy guarantees that insertion operations do not leak information on previous queries. Backward privacy addresses a similar concern with respect to deletion. Specifically, a backward private EMM guarantees that it is not possible to apply a query to data that has been deleted. Enabling update operations only becomes more difficult when studying volume-hiding EMMs. For static volume-hiding EMMs, constructions must ensure volume is not leaked only on query operations. In the dynamic setting, volume must not be leaked by either query

or update operations. Furthermore, designers must ensure that an adversary does not combine knowledge between query and update operations to learn volumes as well. In our work, we will design dynamic volume-hiding EMMs that provide forward and backward privacy while simultaneously being efficient.

1.1 Our Contributions

As our main contribution, we will present *dynamic volume-hiding* EMMs that are *forward and backward private* with better efficiency and to prior works. The state-of-the-art, dynamic, volume hiding scheme was presented in the original work by Kamara and Moataz [20] and is denoted as the *Dense Subgraph Transform*. For a multi-map with n total values and maximum volume ℓ, the Dense Subgraph Transform requires $O(\ell \log n)$ overhead for both query and update operations. Furthermore, this construction supports only a subset of update operations needed for many applications and does not support forward privacy, a standard security notion of the dynamic setting[1]. With this in mind, there are three main challenges that we address in our work.

1. **Dynamicity and Hiding Volume.** The volume-hiding construction in [20] only enables adding, deleting or overwriting the entire tuple associated with a label. In particular, users may not append a single value or remove a value (if it exists) from an existing value tuple. This is an important functionality in most applications. For example, searchable encryption may utilize this function for newly added documents whose identifiers must be added to each tuple associated with words appearing in the document. While one can achieve this functionality using a query before an update, it turns out that this degrades privacy significantly (we outline this further in Appendix A). This motivates the following question:

   Is it possible to construct fully-dynamic volume-hiding constructions with the ability to add/remove a set of values from an already existing tuple that is both efficient and private?

2. **Forward and Backward Privacy.** Introduced by [37, 4] for the special case of dynamic searchable encryptions, forward and backward privacy are the *de-facto* standard security notions for dynamic STEs to protect against various injection attacks [38]. At a high level, forward and backward privacy guarantee that modified data is not leaked until a query for the data is performed. Prior volume-hiding schemes [20, 31] are not simultaneously forward and backward private, which motivates the following problem:

   Is it possible to construct dynamic EMMs that simultaneously hide volumes while providing both forward and backward privacy?

3. **Efficiency.** The Dense Subgraph Transform [20] requires $O(\ell \cdot \log n)$ overhead for both queries and updates, which is larger than the $O(\ell)$ overhead needed by the best static volume-hiding construction [31] and raises the following question:

   Is it possible to construct a dynamic volume-hiding scheme with better efficiency while simultaneously providing forward and backward privacy?

We present two constructions $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ that address all three problems simultaneously and present different trade-offs between client storage and update overhead. We remind the reader that in the following statements ℓ is the maximum length of tuple associated to a label and n is the maximum total number of values.

Theorem 1 (Informal)**.** *There exists a fully-dynamic, volume-hiding encrypted multi-map* $\mathbf{2ch_{FB}}$ *with query overhead of* $O(\ell \log\log n)$*, amortized update overhead of* $O(\ell)$*, server storage of* $O(n)$ *and client storage of size* $O(m)$ *where* m *is the number of unique labels stored in the multi-map. The scheme is both forward and type-II backward private.*

[1]While [20] does not mention that the Dense Subgraph Transform is backward private, we found its leakage profile to be type-II backward private.

	Query Communication	Query Computation	Query Roundtrips	Update Communication	Update Computation	Client Storage	VH	FP	BP
Sofos [4]	$O(\ell_{\texttt{label}})$	$O(\ell_{\texttt{label}})$	1	$O(u_{\texttt{label}})$	$O(u_{\texttt{label}})$	$O(m)$	×	✓	×
Fides [5]	$O(\ell_{\texttt{label}})$	$O(\ell_{\texttt{label}})$	2	$O(u_{\texttt{label}})$	$O(u_{\texttt{label}})$	$O(m)$	×	✓	II
SD_a [11]	$O(\ell_{\texttt{label}})$	$O(\ell_{\texttt{label}})$	1	$O(u_{\texttt{label}} \log n)$	$O(u_{\texttt{label}} \log n)$	$O(1)$	×	✓	II
Dense Subgraph Transform [20]	$O(\ell \log n)$	$O(\ell \log n)$	1	$O(\ell \log n)$	$O(\ell \log n)$	$O(1)$	✓	×	II
Ours ($\mathbf{2ch_{FB}}$)	$O(\ell \log \log n)$	$O(\ell \log \log n)$	1	$O(\ell)$	$O(\ell)$	$O(m)$	✓	✓	II
Ours ($\mathbf{2ch^s_{FB}}$)	$O(\ell \log \log n)$	$O(\ell \log \log n)$	2	$O(\ell \log n)$	$O(\ell \log n)$	$\omega(\log n)$	✓	✓	II

Table 1: A comparison of amortized query and update overhead of dynamic schemes that provide either volume-hiding or forward and backward privacy with our constructions. We use the following abbreviations for each notion: volume-hiding (VH), forward privacy (FP) and backward privacy (BP). For notation, n denotes the maximum number of values in the multi-map and m denotes the number of unique labels in the multi-map. For volume-hiding schemes, ℓ represents the maximum volume. We denote $\ell_{\texttt{label}}$ to be the number of values associated with the queried label and $u_{\texttt{label}}$ to be the number of updated values.

$\mathbf{2ch_{FB}}$ achieves all our goals of dynamicity, volume-hiding, efficiency and forward/backward privacy. However, $\mathbf{2ch_{FB}}$ requires $O(m)$ client storage where m is the number of unique labels, which is common to the majority of forward private schemes, such as constructions in [4, 5]. We present $\mathbf{2ch^s_{FB}}$ with smaller permanent client storage at the cost of slightly more query and update overhead.

Theorem 2 (Informal)**.** *There exists a fully-dynamic, volume-hiding encrypted multi-map* $\mathbf{2ch^s_{FB}}$ *with query overhead of* $O(\ell \log \log n)$*, amortized update overhead of* $O(\ell \log n)$*, server storage of* $O(n)$ *and permanent client storage of size at most* $f(n)$*, for every function* $f(n) = \omega(\log n)$*. The scheme is both forward and type-II backward private.*

To our knowledge, $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ are the first *dynamic* EMM constructions that *simultaneously* provide *volume-hiding*, *forward* and *backward privacy* while being *concretely efficient* with a small number of roundtrips that exclusively use symmetric cryptographic primitives. A comparison of the asymptotic performance of our constructions and prior dynamic schemes obtaining at least one of volume-hiding, forward or backward privacy are presented in Table 1.

The experimental evaluation of our constructions in Section 4 shows that our constructions also improve on the concrete performance of previous constructions. First of all, our evaluations show that we enable dynamicity without incurring any additional cost when compared with previous static constructions [31]. This is very surprising as static constructions are optimized for query communication whereas experiments show that our dynamic constructions, despite having to support a very rich set of dynamic operations, have query cost comparable with the static construction of [31]. In addition, our constructions exhibit a 2-3x improvement in query communication cost also over the Dense Subgraph Transform [20], the best non-lossy volume hiding dynamic construction from the literature. This improvement is obtained while supporting a wider range of dynamic operations and providing stronger security guarantees.

Discussion about Backward Privacy. Both of $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ provide type-II backward privacy as defined by Bost [4]. We note that there are several constructions that provide stronger type-I backward privacy. However, current type-I backward private constructions are expensive and resort to usage of oblivious RAMs. As a result, we do not consider type-I backward privacy and leave it as an open problem for future work.

Discussion about Oblivious RAMs. From a theoretical perspective, oblivious RAMs [12, 28, 1] address the problems of dynamicity and forward/backward privacy (as outlined in [20]). However, all oblivious RAMs are expensive as they require logarithmic number of client-server roundtrips or fully homomorphic encryption (FHE) schemes. As evidenced by prior works such as [34], the high number of roundtrips of ORAMs significantly hinder their efficiency. In our work, we ensure all our constructions use either 1 or 2 roundtrips and only use cheap, symmetric primitives instead of expensive cryptographic tools such as ORAMs and FHE.

2 Definitions

In this section, we recall all necessary definitions for our constructions including structured encryption, security against adaptive adversaries, the specific operations supported by our encrypted multi-maps as well as forward and backward privacy. Additionally, we formally define the notion of volume-hiding in the dynamic setting.

2.1 Structured Encryption

In a structured encryption (STE) scheme, a client may encrypt and outsource storage of the data structure to a server. The encryption is structured in such a way that the underlying data structure may be operated on by the client in a private manner. The notion of STE was first presented by Chase and Kamara [9]. While we consider generic definitions for encrypting any data structure, our work focuses on multi-maps as they are a simple data structure with several applications.

STE schemes may be differentiated using several criteria. Static STE schemes only enable clients to query the underlying data structure while dynamic STE schemes additionally enable clients to update the underlying data structure. We will focus on dynamic STE schemes that consist of three two-party protocols to be executed by the client and the server: the Setup protocol to compute the initial encryption of the data structure, the Query protocol to query the data structure, and the Update protocol to update the data structure.

The number of communication rounds between the client and server is an important measure. We say that an operation of a STE scheme is r-interactive if it can be completed in at most r rounds of communication between the client and the server. A STE scheme is r-interactive if all operations use at most r rounds of interaction. In our work, we will exclusively focus on STE schemes with a low number of rounds of interaction as they are more practical and efficient thanks to the small number of communication roundtrips between the client and server.

Finally, STE schemes may be distinguished between response-hiding and response-revealing schemes where the former reveals the response to queries to the server in the plaintext whereas the latter typically do not leak anything except an upper bound on the response size. All our constructions will be response-hiding.

Definition 1 (r-Interactive, Dynamic Structured Encryption). *A r-interactive dynamic structured encryption scheme $\Sigma = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ consists of the following two-party protocols between the client $\mathbb{C}$ and the server $\mathbb{S}$:*

1. $(\mathsf{st}; \mathsf{EDS}) \leftarrow \mathsf{Setup}((1^\lambda, \mathsf{params}, \mathsf{DS}); 1^\lambda)$. *The setup protocol is executed jointly by $\mathbb{C}$, running on input the security parameter 1^λ, the parameters of the data structure* params*, and the input data structure* DS*, and $\mathbb{S}$, running on input the security parameter 1^λ. At termination, $\mathbb{C}$ receives and stores its* state st *and $\mathbb{S}$ receives and stores the* encrypted data structure EDS.

2. $((\mathsf{Response}, \mathsf{st}^{\mathsf{new}}); \mathsf{EDS}^{\mathsf{new}}) \leftarrow \mathsf{Query}((\mathsf{st}, \mathsf{qop}); \mathsf{EDS})$. *The query protocol is executed jointly by $\mathbb{C}$, running on the current state* st *and the query* qop *to be performed, and $\mathbb{S}$, running on input the encrypted data structure. The protocol runs for at most r rounds of interaction. For each $i \in \{0, \ldots, r-1\}$, $\mathbb{C}$ generates the i-th message using the state* st*, the query operation* qop *and the previous $i-1$ server messages. $\mathbb{S}$ generates the i-th message using the i-th client message and the encrypted data structure* EDS*. At termination, $\mathbb{C}$ receives the result of the query,* $\mathsf{Response}$ *and an updated state* $\mathsf{st}^{\mathsf{new}}$*, and $\mathbb{S}$ receives an updated encrypted data structure* $\mathsf{EDS}^{\mathsf{new}}$.

3. $(\mathsf{st}^{\mathsf{new}}; \mathsf{EDS}^{\mathsf{new}}) \leftarrow \mathsf{Update}((\mathsf{st}, \mathsf{up}); \mathsf{EDS})$. *The update protocol is executed jointly by $\mathbb{C}$ and $\mathbb{S}$ using at most r rounds of interaction. For each $i \in \{0, \ldots, r-1\}$, $\mathbb{C}$ generates the i-th message using the state* st*, the update operation* up *and the previous $i-1$ server messages. $\mathbb{S}$ generates the i-th message using the i-th client message and the encrypted data structure* EDS*. At termination, $\mathbb{C}$ receives an updated state* $\mathsf{st}^{\mathsf{new}}$ *and the $\mathbb{S}$ receives an updated the encrypted data structure* $\mathsf{EDS}^{\mathsf{new}}$.

2.2 Adaptive Security

We consider the notion of security for STE schemes against an honest-but-curious PPT adversary $\mathcal{A}$ with respect to a *leakage function* $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$. The leakage function is an upper bound on the amount of information leaked to the adversary in the sense that (1) the initial encrypted structure reveals no information beyond $\mathcal{L}_{\mathsf{Setup}}$; (2) performing a query operation reveals no information beyond $\mathcal{L}_{\mathsf{Query}}$; and (3) performing an update operation reveals no information beyond $\mathcal{L}_{\mathsf{Update}}$. Note that the leakage on an operation may depend on all the previous operations issued and on the initial data structure which are passed as arguments to the leakage function.

We consider *adaptive* security that considers adversaries that view the execution of one operation before choosing the next operation. Adaptive security was first formalized by Curtmola et al. [10] in the context of searchable encryption and later generalized to structured encryption in [9]. The definition is in the *real-ideal paradigm* with a stateful, honest-but-curious, PPT adversary $\mathcal{A}$ and a stateful, PPT simulator $\mathcal{S}$.

More formally, let $\Sigma = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ be a dynamic STE and consider the following *real game* $\mathbf{Real}_{\Sigma,\mathcal{A}}$ and *ideal game* $\mathbf{Ideal}^{\mathcal{L},\mathcal{S}}_{\Sigma,\mathcal{A}}$ between a stateful PPT adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. In the ideal game, $\mathcal{S}$ is a stateful PPT *simulator* and $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ is a leakage function.

$\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda, z)$: Adversary $\mathcal{A}(1^\lambda, z)$, on input the *security parameter* λ and the *auxiliary information* z, outputs an input data structure DS. The challenger $\mathcal{C}$ executes Setup on DS obtaining client state st and encrypted data structure EDS. $\mathcal{C}$ sends EDS to $\mathcal{A}$.

For $i = 1, \ldots, \mathtt{poly}(\lambda)$

- $\mathcal{A}$ adaptively picks operation o_i.
- If o_i is a query operation, $\mathcal{A}$ and $\mathcal{C}$ execute $((\mathsf{Response}, \mathsf{st}^{\mathsf{new}}); \mathsf{EDS}^{\mathsf{new}}) \leftarrow \mathsf{Query}((\mathsf{st}, o_i); \mathsf{EDS})$.
- If o_i is an update operation, $\mathcal{A}$ and $\mathcal{C}$ execute $(\mathsf{st}^{\mathsf{new}}; \mathsf{EDS}^{\mathsf{new}}) \leftarrow \mathsf{Update}((\mathsf{st}, o_i); \mathsf{EDS})$.
- In both cases $\mathcal{A}$ plays the role of the server $\mathbb{S}$ and challenger $\mathcal{C}$ plays the role of the client $\mathbb{C}$.

  Therefore, $\mathcal{A}$ receives a transcript of the protocol and updates EDS by setting $\mathsf{EDS} \leftarrow \mathsf{EDS}^{\mathsf{new}}$ and $\mathcal{C}$ updates the client state by setting $\mathsf{st} \leftarrow \mathsf{st}^{\mathsf{new}}$.

Finally, $\mathcal{A}$ outputs $b \in \{0, 1\}$.

$\mathbf{Ideal}^{\mathcal{L},\mathcal{S}}_{\Sigma,\mathcal{A}}(1^\lambda, z)$: Adversary $\mathcal{A}(1^\lambda, z)$, on input the *security parameter* λ and the *auxiliary information* z, outputs an input data structure DS. The challenger $\mathcal{C}$ runs the simulator $\mathcal{S}$ on input leakage $\mathcal{L}_{\mathsf{Setup}}(\mathsf{DS})$ and the auxiliary information z to obtain the encrypted data structure EDS that is sent to the adversary $\mathcal{A}$.

For $i = 1, \ldots, \mathtt{poly}(\lambda)$,

- $\mathcal{A}$ adaptively picks operation o_i
- if o_i is a query operation, then $\mathcal{A}$ and $\mathcal{S}$, on $\mathcal{L}_{\mathsf{Query}}(\mathsf{DS}, o_1, \ldots, o_i)$, execute protocol Query.
- if o_i is an update operation, then $\mathcal{A}$ and $\mathcal{S}$, on $\mathcal{L}_{\mathsf{Update}}(\mathsf{DS}, o_1, \ldots, o_i)$, execute protocol Update.
- In both cases $\mathcal{A}$ plays the role of the server $\mathbb{S}$ and $\mathcal{S}$ the role of the client $\mathbb{C}$. Therefore, $\mathcal{A}$ receives a transcript of the protocol and an updated version of EDS. Note, $\mathcal{S}$ is allowed to deviate from the protocols.

Finally, $\mathcal{A}$ outputs $b \in \{0, 1\}$.

We are now ready to present our notion of security.

Definition 2 (Adaptive Security). *STE scheme* Σ *is* adaptively $\mathcal{L}$-secure *if there exists a stateful, PPT simulator* $\mathcal{S}$ *such that for all stateful, PPT adversaries* $\mathcal{A}$ *and all auxiliary information* $z \in \{0,1\}^*$*:*

$$\left| \Pr\left[\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda, z) = 1\right] - \Pr\left[\mathbf{Ideal}^{\mathcal{L},\mathcal{S}}_{\Sigma,\mathcal{A}}(1^\lambda, z) = 1\right] \right| \leq \mathtt{negl}(\lambda).$$

2.3 Multi-Maps

A *multi-map* MM stores a collection of label-to-value-tuple pairs $(\mathtt{label}, \vec{\mathtt{v}})$ where $\mathtt{label}$ is from the *label universe* $\mathbb{L}$ and $\vec{\mathtt{v}}$ is a tuple of values from the *value universe* $\mathbb{V}$. For a multi-map MM we denote by $\mathtt{LABEL}(\mathsf{MM})$ the set of labels that appear in MM and, for each $\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})$, we denote by $\mathsf{MM}[\mathtt{label}]$ the tuple $\vec{\mathtt{v}}$ such that $(\mathtt{label}, \vec{\mathtt{v}}) \in \mathsf{MM}$. If $\mathtt{label} \notin \mathtt{LABEL}(\mathsf{MM})$ then $\mathsf{MM}[\mathtt{label}] := \perp$. We will use $m := |\mathtt{LABEL}(\mathsf{MM})|$ to denote the number of unique labels in the multi-map and $n := \sum_{\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})} |\mathsf{MM}[\mathtt{label}]|$ for its *size*; that is, the total number of values in the multi-map. We denote the *volume* of $\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})$ by $\ell_{\mathsf{MM}}(\mathtt{label}) := |\mathsf{MM}[\mathtt{label})]|$; if, instead, $\mathtt{label} \notin \mathtt{LABEL}(\mathsf{MM})$ then we set $\ell_{\mathsf{MM}}(\mathtt{label}) := 0$. The *volume* of a multi-map MM, denoted ℓ, is the maximum volume of a label; that is, $\ell := \max_{\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})} |\mathsf{MM}[\mathtt{label}]|$.

Dynamic multi-maps enable the following operations for an input multi-map MM.

1. $\mathsf{Response} \leftarrow \mathsf{Query}(\mathtt{label}, \mathsf{MM})$. The *query operation* retrieves the tuple $\mathsf{MM}[\mathtt{label}]$.
2. $\mathsf{MM}^{\mathsf{new}} \leftarrow \mathsf{Update}(\mathsf{op} \in \{\mathsf{edit}, \mathsf{app}, \mathsf{rm}, \mathsf{del}\}, \mathtt{label}, \vec{\mathtt{v}}, \mathsf{MM})$. A multi-map supports the following types of update operations.
   (a) If $\mathsf{op} = \mathsf{edit}$, this is a *label edit* operation. The label edit operations modify the tuple associated with $\mathtt{label}$. Specifically, if $\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})$, then the label edit operation overwrites $\mathsf{MM}[\mathtt{label}]$ by setting $\mathsf{MM}[\mathtt{label}] \leftarrow \vec{\mathtt{v}}$. Otherwise, the edit operations adds $(\mathtt{label}, \vec{\mathtt{v}})$ to MM.
   (b) If $\mathsf{op} = \mathsf{rm}$, this is a *label removal* operation. If $\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})$ then the label removal operation removes the pair $(\mathtt{label}, \mathsf{MM}[\mathtt{label}])$ from MM. The input $\vec{\mathtt{v}}$ is ignored. If $\mathtt{label} \notin \mathtt{LABEL}(\mathsf{MM})$, the label removal operation has no effect.
   (c) If $\mathsf{op} = \mathsf{app}$, this is an *label-value append* operation. If $\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})$, the label-value append operation sets $\mathsf{MM}[\mathtt{label}] \leftarrow \mathsf{MM}[\mathtt{label}] \mid\mid \vec{\mathtt{v}}$; that is, it appends the tuple $\vec{\mathtt{v}}$ to the current tuple associated with $\mathtt{label}$. If $\mathtt{label} \notin \mathtt{LABEL}(\mathsf{MM})$ then the label-value append has the effect of adding $(\mathtt{label}, \vec{\mathtt{v}})$ to MM, thus setting $\mathsf{MM}[\mathtt{label}] \leftarrow \vec{\mathtt{v}}$.
   (d) If $\mathsf{op} = \mathsf{del}$, this is a *label-value deletion* operation. If $\mathtt{label} \in \mathtt{LABEL}(\mathsf{MM})$, the deletion operation sets $\mathsf{MM}[\mathtt{label}] \leftarrow \mathsf{MM}[\mathtt{label}] \setminus \vec{\mathtt{v}}$. That is, all values in $\vec{\mathtt{v}}$ are removed from $\mathsf{MM}[\mathtt{label}]$. If $\mathtt{label} \notin \mathtt{LABEL}(\mathsf{MM})$ then the operation has not effect.

For convenience, we represent multi-map operations as the tuple $o = (\mathsf{op}, \mathtt{label}, \vec{\mathtt{v}})$ where op is the operation type, $\mathtt{label}$ is the input label and $\vec{\mathtt{v}}$ is the input value tuple. If o is a query, then $\mathsf{op} = \mathsf{qop}$, $\mathtt{label}$ is the queried label and $\vec{\mathtt{v}} = \perp$. If o is an update, then op is the update operation type, $\mathsf{op} \in \{\mathsf{edit}, \mathsf{rm}, \mathsf{app}, \mathsf{del}\}$, $\mathtt{label}$ is the updated label and $\vec{\mathtt{v}}$ is tuple of values to update. If $\mathsf{op} = \mathsf{rm}$, then $\vec{\mathtt{v}} = \perp$ as it is not needed. Additionally we use the function $\mathsf{op}(o) \in \{\mathsf{qop}, \mathsf{edit}, \mathsf{rm}, \mathsf{app}, \mathsf{del}\}$ to denote the operation type of an operation o. Similarly, we denote $\mathtt{label}(o)$ to be the label of focus for operation o and $\vec{\mathtt{v}}(o)$ to be the value tuple for the operation o.

In the above definition, we separate query and update operations for convenience when we consider forward and backward privacy that imposes one set of of requirements for queries and a different requirement for all types of update operations.

2.4 Volume Hiding Leakage Functions

Volume-hiding leakage functions were introduced in [20] and formally defined in [31] for static schemes. Volume-hiding ensures that no information about volumes are revealed at any point in time. We present a definition of a *volume-hiding leakage function* for dynamic encrypted multi-maps and use a game-based definition for volume-hiding to enable a clear comparison with the static definition in the previous work [31].

For a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ and an adversary $\mathcal{A}$, we consider two games $\mathbf{VHGame}_{\eta}^{\mathcal{A},\mathcal{L}}(n, \ell)$, with $\eta = 0, 1$, for maximum size n and maximum volume ℓ. The adversary $\mathcal{A}$ selects two multi-maps of his choice MM_0 and MM_1 with size at most n and maximum volume at most ℓ. The adversary then issues a sequence of operations and, in game $\mathbf{VHGame}_{\eta}$ it receives the leakage with respect

to MM^η. To make the game non-trivial we request that no operation that makes either of the two input multi-maps exceed the maximum size n or the maximum volume ℓ is issued by $\mathcal{A}$.

$\mathbf{VHGame}_\eta^{\mathcal{A},\mathcal{L}}(n,\ell)$:

1. $\mathcal{A}$ outputs two multi-maps MM_0 and MM_1 of respective sizes $n_0, n_1 \leq n$ and volume $\ell_0, \ell_1 \leq \ell$. Both MM^0 and MM^1 are sent to $\mathcal{C}$.

2. $\mathcal{C}$ computes $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}^\eta)$ which is sent to $\mathcal{A}$.

3. for $t = 1, \ldots,$

   (a) $\mathcal{A}$ adaptively picks operations $o_t^0 = (\mathtt{op}_t^0, \mathtt{label}_t^0, \vec{\mathtt{v}}_t^0)$ and $o_t^1 = (\mathtt{op}_t^1, \mathtt{label}_t^1, \vec{\mathtt{v}}_t^1)$ such that

      i. the operations are either both qop or both up.

      ii. the labels are the same: $\mathtt{label}_t^0 = \mathtt{label}_t^1$.

   (b) $\mathcal{A}$ updates MM_0 by executing o_t^0 and MM_1 by executing o_t^1.

   (c) If neither MM_0 and MM_1 exceeds the maximum size n or the maximum volume ℓ, then $\mathcal{C}$ returns $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}^\eta, o_0^\eta, \ldots, o_t^\eta)$, if $\mathtt{op}_t^\eta = \mathsf{qop}$, and $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}^\eta, o_0^\eta, \ldots, o_t^\eta)$, if $\mathtt{op}_t^\eta = \mathsf{up}$.

4. Finally, $\mathcal{A}$ outputs a bit $b \in \{0,1\}$.

We denote by $p_\eta^{\mathcal{A},\mathcal{L}}(n,\ell)$ as the probability that $\mathcal{A}$ outputs 1 when playing game $\mathbf{VHGame}_\eta^{\mathcal{A},\mathcal{L}}(n,\ell)$.

Definition 3 (Volume-Hiding Leakage Functions). *A leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ is volume-hiding if and only if for all adversaries $\mathcal{A}$ and for all values $n \geq \ell \geq 1$,*

$$p_0^{\mathcal{A},\mathcal{L}}(n,\ell) = p_1^{\mathcal{A},\mathcal{L}}(n,\ell).$$

Using the above definition, we define volume-hiding encrypted multi-maps.

Definition 4 (Volume-Hiding Encrypted Multi-Maps). *An encrypted multi-map scheme Σ is volume-hiding if there exists a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ such that:*

1. *Σ is adaptively $\mathcal{L}$-secure according to Definition 2.*

2. *$\mathcal{L}$ is a volume-hiding leakage function according to Definition 3.*

The above volume-hiding definition ensure that the adversary $\mathcal{A}$ cannot learn the volumes of the labels in the setup multi-map and how they change as update operations are performed. Even given this power, the volume-hiding leakage function ensures that the two operational sequences remain indistinguishable. We remind the reader that in this paper we only consider response-hiding leakage functions and this is reflected by the definition above. Indeed, the adversary is allowed to choose the initial values of the multi-maps and the values with which the multi-maps are updated and still it is unable to distinguish from which of the two operational sequences the leakage is originating.

2.5 Forward and Backward Privacy

Forward and backward privacy (see [37] and [4, 5]) provide guarantees on the amount of information leaked to an adversary as the client performs update operations. Forward privacy guarantees that the leakage of update operations is independent of all previous operations. Backward privacy controls the leakage viewed by the adversary during query operations about previous deletion operations.

We follow the original definitions by Bost [4] that are specified for encrypted multi-maps.

Definition 5 (Forward Privacy). *A leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ is* forward private *if there exists a leakage function $\mathcal{L}'_{\mathsf{Update}}$ such that for any multi-map MM, any sequence of operations O and any update operation o,*

$$\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = \mathcal{L}'_{\mathsf{Update}}(\mathtt{op}(o), \vec{\mathtt{v}}(o)).$$

We remind the reader that $\vec{v}(o)$ denotes the value tuple in the update operation o. For any forward private leakage function, the leakage incurred by an update o does not give any information on the sequence of operations O except the update operation itself as we can see from Definition 5.

Bost *et al.* [5] formally defined three types of backward privacy where type-I provides the strongest privacy to type-III, providing the weakest privacy. Backward privacy guarantees that deleted values are not revealed until they are queried. However, the amount of leakage revealed at query time differs in each of the types. With every operation o_i of a sequence $O = (o_1, \ldots, o_t)$, we associate the timestamp $\mathsf{time}(o_i)$ denoting the time at which operation o_i was performed. To define backward privacy, we need the following three additional leakage functions. Each function takes as argument a sequence O of operations that is omitted not to overburden notation.

$$\mathsf{TimeDB}(\texttt{label}) = \{(\mathsf{time}(o_i), v) \mid v \in \mathsf{MM}[\texttt{label}] \text{ and } o_i \text{ is the last operation to add } v \text{ to } \mathsf{MM}[\texttt{label}]\}.$$

That is, $\mathsf{TimeDB}(\texttt{label})$ contains pairs consisting of values appearing in $\mathsf{MM}[\texttt{label}]$ and of the timestamp of the operation that inserted those values into $\mathsf{MM}[\texttt{label}]$. Next, we define

$$\mathsf{TimeUpdate}(\texttt{label}) = \{\mathsf{time}(o_i) \mid \mathsf{op}(o_i) \in \{\mathsf{edit}, \mathsf{rm}, \mathsf{app}, \mathsf{del}\}, \texttt{label}(\mathsf{o_i}) = \texttt{label}\}$$

that consists of the timestamps of all update operations that modify `label`. Finally,

$$\mathsf{DelHist}(\texttt{label}) = \{(\mathsf{time}(o_i), \mathsf{time}(o_j)) \mid o_i \text{ inserted } v \text{ for } \texttt{label}, \text{ removed by } o_j\}$$

is a list of pairs of timestamps for operations o_i and o_j where o_i inserted a value into $\mathsf{MM}[\texttt{label}]$ that was later deleted by o_j. Finally, let $a_{\texttt{label}}$ denote the the total number of values inserted into $\mathsf{MM}[\texttt{label}]$ in total (including those values that were later deleted). We now define the three levels of backward privacy:

Definition 6 (Backward Privacy)**.** *A leakage function* $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ *is* Type-I, Type-II, Type-III backward private *if there exist leakage functions* $\mathcal{L}'$ *and* $\mathcal{L}''$ *such that the following conditions are satisfied.*

Type-I backward private*:*

- $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = \mathcal{L}'(\mathsf{op}(o))$*;*
- $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = \mathcal{L}''(\mathsf{TimeDB}(\texttt{label}(o)), a_{\texttt{label}})$*;*

Type-II backward private*:*

- $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = \mathcal{L}'(\mathsf{op}(o), \texttt{label}(o))$*;*
- $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = \mathcal{L}''(\mathsf{TimeDB}(\texttt{label}(o)), \mathsf{TimeUpdate}(\texttt{label}(o)))$*;*

Type-III backward private*:*

- $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = \mathcal{L}'(\mathsf{op}(o), \texttt{label}(o))$*;*
- $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = \mathcal{L}''(\mathsf{TimeDB}(\texttt{label}(o)), \mathsf{DelHist}(\texttt{label}(o)))$*;*

As seen from the definition, at query time for `label`, type-I backward privacy reveals the total number of updates performed on `label`. Type-II backward privacy also reveals the timestamps of each update operation for `label`. Finally, type-III backward privacy additionally reveals pairings of update operations that deleted values inserted by a prior update operation. All our constructions will be type-II backward private.

Discussion about Large Upper Bounds. We note that all types of backward privacy allow the adversary to learn the plaintext tuple value associated with a query `label` that appears in $\mathsf{TimeDB}(\texttt{label})$. This is done so that backward privacy may also be compatible with response-revealing schemes that are important in some applications. We stress that none of our schemes will ever leak information associated with $\mathsf{TimeDB}(\texttt{label})$ nor $\mathsf{DelHist}(\texttt{label})$. Indeed, neither $\mathsf{TimeDB}(\texttt{label})$ or $\mathsf{DelHist}(\texttt{label})$ will appear in our proofs. Instead, we note that backward privacy defines upper bounds on leakage. Our type-II backward private schemes will leak significantly less than these upper bounds.

2.6 Typical Leakage Functions

Finally, we describe common leakage functions that will be utilized in our paper.

Maximum Volume: This leakage reports (an upper bound of) the largest value tuple size denoted by ℓ.

Multi-Map Size: This leakage reports (an upper bound of) the total number of values in the multi-map denoted by n.

Label Equality: This leakage reports the label equality pattern leaking whether two operations are performed on the same label or not. For a sequence of operations $o_1, \ldots, o_t$, $\mathsf{leq}(o_1, \ldots, o_t) = M$ consists of a $t \times t$ matrix such that $M[i][j] = 1$ iff $\mathtt{label}(o_i) = \mathtt{label}(o_j)$.

3 Dynamic Volume Hiding EMMs

We move to presenting our new constructions. First, we present a warm-up construction **2ch** that achieves full-dynamicity, efficiency and backward privacy but not forward privacy. Next, we present $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ that build upon **2ch** to obtain forward and backward privacy simultaneously with different efficiency trade-offs.

3.1 Simple Warm-Up Scheme from Two-Choice Hashing

We start by discussing the optimal static volume-hiding scheme by Patel *et al.* [31]. At a high level, their idea is to utilize hashing to embed values into a small hash table that is stored by the server. Their construction utilizes cuckoo hashing [27, 22] to embed data into server storage. At a high level, cuckoo hashing guarantees that each value is stored in one of two hash table locations (or a small client stash that stores overflow values). To perform queries, one can simply access 2ℓ hash table locations. Unfortunately, inserting values with cuckoo hashing is much more complex. Cuckoo hashing insertion works in an iterative fashion where a value is placed into two locations and, if both locations are occupied, displaces one of the values that must be inserted again. This algorithm is not volume-hiding as the adversary learns whether certain entries are occupied or not by viewing how long the insertion algorithm ran.

Looking closer, the query algorithm with cuckoo hashing [31] is volume-hiding because 2ℓ entries are retrieved regardless of the hash table's contents. On the other hand, the insertion algorithm heavily depends on the hash table's contents. It turns out that the simple balls-into-bins hashing scheme obtains the property that both query and update operations are independent of the table contents. The balls-and-bins hashing scheme considers n bins. To insert a value, it is placed into one of the n bins uniformly at random. If we have n values and n bins, the maximum number of values assigned to a bin will be $\Theta(\log n)$ (see [25]). To obtain volume-hiding, all bins must be padded to the maximum load of $\Theta(\log n)$. Both query and update operations will access ℓ bins possibly with dummies to attain volume-hiding resulting in $O(\ell \log n)$ overhead. This technique is employed by the Dense Subgraph Transform [20].

Our goal is to find a hashing scheme with efficiency better than balls-into-bins hashing while ensuring both queries and updates are volume-hiding. To achieve this goal, we utilize *two-choice hashing* by Azar *et al.* [2]. Once again, there are n bins. To insert an value, two bins are chosen uniformly at random and the item is placed into the bin that is least loaded (i.e. currently contains less items). Using this technique, the maximum bin size becomes $O(\log \log n)$. Unfortunately, the server storage grows to $O(n \log \log n)$ since each of the n bins must be padded to $O(\log \log n)$ size to hide the true number of values in each bin.

To avoid this extra storage, we can utilize a modified version of two-choice hashing introduced in [29] with $O(n)$ storage that reduces the amount of unused space by arranging bins to share physical memory. The hash table consists of $O(n/\log n)$ binary trees each of height $O(\log \log n)$ such that there are at least n leaves. As a result, the hash table now consists of $O(n)$ nodes. Tree levels increase as nodes are further away from the root (roots are level 0 and leaves are at the highest level). All binary tree nodes will store at most one value. Each bin is uniquely assigned to a binary tree leaf and the bin's storage corresponds to the nodes that appear on the unique path from the bin's leaf to the root of its respective binary tree. For insertion, the least loaded bin is the one with the an empty node that is at the highest level (i.e. furthest

away from its corresponding root). Additionally, there is a stash to store overflows. Whenever a value is inserted into two bins that are completely filled (all nodes appearing on the unique leaf-to-root paths are occupied), the item is instead placed into the stash. The analysis in [29] shows that the stash does not exceed $f(n) = \omega(\log n)$ items except with negligible probability. To obtain volume-hiding, all empty nodes will be filled with encrypted dummies.

Using these new hashing techniques, we may obtain a dynamic volume-hiding scheme with $O(\ell \log\log n)$ overhead that we denote as **2ch** (standing for **2-c**hoice **h**ashing) following the same techniques as [20, 31] that maps values to bins using pseudorandom functions. We note that **2ch** already results in a more efficient construction that the state-of-the-art dynamic construction, Dense Subgraph Transform [20]. However, using the standard techniques from prior static, volume-hiding schemes would not achieve forward privacy. We note that **2ch** already offers type-II backward privacy. Therefore, we leave the main ideas to the readers and omit a formal description and analysis in the main body. We present the pseudocode for **2ch** in Appendix B along with a formal proof of security and efficiency.

3.2 Construction $\mathbf{2ch_{FB}}$

We formally present our dynamic volume-hiding STE scheme for multi-maps $\mathbf{2ch_{FB}}$ (standing for **2-c**hoice **h**ashing with **F**orward and **B**ackward privacy). $\mathbf{2ch_{FB}}$ builds upon our hashing techniques from the prior section. The major difference between $\mathbf{2ch_{FB}}$ and prior volume-hiding works lies in the update algorithms. For forward privacy, we need to make sure that an update on a label does not leak anything about previous queries on the same label. In prior works, all update operations will immediately applied to the encrypted data structure. For example, consider a query for `label` followed by an update for `label`. Constructions in [20, 31] will retrieve the identical bins for both operations. As a result, an adversary can link that both operations were performed on the same label. This is the main reason why prior constructions (as well as **2ch**) is not forward private.

We take a different approach to update operations for $\mathbf{2ch_{FB}}$ where update operations are not immediately applied to the underlying multi-map inspired by prior works of [4, 5]. The update operation is only applied when a query for the same label is performed. In more detail, $\mathbf{2ch_{FB}}$ outsources two encrypted data stores to the server. The first multi-map Table stores all values of update operations that have already been queried (i.e. the update operations were applied). The other multi-map EMM_u stores only update operations for labels that have yet to be queried. Once a query for `label` is performed, all update operations pertaining to `label` will be be retrieved from EMM_u and applied to Table before returning the final result. With this method, the adversary cannot learn any information about update operations until a query for the same label is performed. As a result, $\mathbf{2ch_{FB}}$ achieves forward privacy. Additionally, we show that $\mathbf{2ch_{FB}}$ also provides type-II backward privacy.

We now formally present $\mathbf{2ch_{FB}}$. The pseudocode can be found in Figure 1.

Setup. The setup algorithm is executed by the client $\mathbb{C}$ to construct an encrypted map EMM. It takes as input a security parameter 1^λ, $\mathsf{params} = (n, \ell)$, where n is an upper bound on the total number of values that will be stored and ℓ is an upper bound on the maximum volume, and a multi-map MM. Setup creates a two-choice hash table Table by constructing $s = \lceil n/(c \log n) \rceil$ full binary trees each of height $\lceil \log(c \log n) \rceil = O(\log\log n)$ for a sufficiently large constant $c \geq 1$. In the above, all logarithms are base 2. Each bin is assigned uniquely to a binary tree leaf arbitrarily. The bin's storage consists of the storage of all nodes appearing on the root-to-leaf path to the assigned leaf. A stash is also initialized for overflows that will be stored by the client locally. Next, setup inserts all labels and values of MM into the two-choice hash table. The algorithm selects a random seed K for PRF F and an encryption key K_{Enc}. For each $\mathtt{label} \in \mathsf{MM}$, setup first computes the seed $F_K(\mathtt{label})$; then, for each $\vec{v}[j] \in \mathsf{MM}[\mathtt{label}]$, the two bins to store $(\mathtt{label}, \vec{v}[j])$ are computed as $G_{F_K(\mathtt{label})}(j \mid\mid 0)$ and $G_{F_K(\mathtt{label})}(j \mid\mid 1)$, where G is a PRF. An encryption of $(\mathtt{label}, j, \vec{v}[j])$ is stored in the empty node in either bin with the highest level. If both bins are fully occupied, the tuple $(\mathtt{label}, j, \vec{v}[j])$ is stored in the stash. After inserting all values, all empty tree nodes are filled with encrypted dummies.

The forest of binary trees Table will be sent to the server $\mathbb{S}$, along with an empty multi-map EMM_u

Let F, G be PRFs, H be a keyed hash function and $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA encryption scheme.

$(\mathtt{st}; \mathsf{EMM}) \leftarrow \mathbf{2ch_{FB}}.\mathsf{Setup}\big(1^\lambda, \mathsf{params} = (n, \ell), \mathsf{MM} = \{(\mathtt{label}_i, \vec{\mathtt{v}}_i)\}_{i \in [|\mathtt{label} \in \mathsf{MM}|]}\big)$:

1. $\mathbb{C}$ randomly selects a PRF keys $K, K_u \leftarrow \{0,1\}^\lambda$ and generates $K_{\mathsf{Enc}} \leftarrow \mathsf{Gen}(1^\lambda)$.
2. $\mathbb{C}$ creates $s := \lceil n/(c \log n) \rceil$ full binary trees, $\mathtt{Table} \leftarrow (B_1, \ldots, B_s)$ each of height $h := \lceil \log(c \log n) \rceil$. Roots are at level 0 and leaf nodes are at height h. Each node has the capacity to hold a single encryption. Each of the n bins are uniquely assigned to n different leaf nodes.
3. $\mathbb{C}$ initializes $\mathtt{Stash} \leftarrow \emptyset$ and two empty multi-maps EMM_u, $\mathsf{MM}_{\mathsf{st}}$.
4. For each $\mathtt{label}_i \in \mathsf{MM}$:
   (a) Compute $\kappa \leftarrow F_K(\mathtt{label}_i)$ and for each $j \in [|\vec{\mathtt{v}}_i|]$:
      i. $\mathbb{C}$ computes $b_0 \leftarrow G_\kappa(j \mid\mid 0)$ and $b_1 \leftarrow G_\kappa(j \mid\mid 1)$ and locates the two leaf-to-root paths associated with bins b_0 and b_1.
      ii. $\mathbb{C}$ computes $\mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{label}_i, j, \vec{\mathtt{v}}[j]))$ and places it into the empty node at the highest level in either bin b_0 or b_1.
      iii. If both bin b_0 and bin b_1 contain no empty nodes, add $(\mathtt{label}_i, j, \vec{\mathtt{v}}[j])$ to $\mathtt{Stash}$.
5. For all empty nodes in the binary trees, $\mathbb{C}$ adds a fresh encryption of $\mathsf{Enc}(K_{\mathsf{Enc}}, (\perp, \perp, \perp))$.
6. $\mathbb{C}$ sets its state $\mathsf{st} \leftarrow (K, K_u, K_{\mathsf{Enc}}, \mathtt{Stash}, \mathsf{MM}_{\mathsf{st}})$ and $\mathbb{S}$ stores $\mathsf{EMM} \leftarrow (B_1, \ldots, B_s, \mathsf{EMM}_u)$.

$(\mathsf{st}'; \mathsf{EMM}') \leftarrow \mathbf{2ch_{FB}}.\mathsf{Update}\left(((\mathtt{op}, \mathtt{label}, \vec{\mathtt{v}}), \mathsf{st}), \mathsf{EMM}\right)$:

1. If $\mathtt{label} \notin \mathsf{MM}_{\mathsf{st}}$, $\mathbb{C}$ sets $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}] \leftarrow (0, 0)$.
2. $\mathbb{C}$ computes $x \leftarrow H(K_u, \mathtt{label} \,||\, \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][0])$ and $y \leftarrow H(x, \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1])$.
3. $\mathbb{C}$ pads $\vec{\mathtt{v}}$ up to ℓ values with $\perp$ and computes $z \leftarrow \mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{op}, \vec{\mathtt{v}}))$.
4. $\mathbb{C}$ sends (y, z) to $\mathbb{S}$ who updates EMM_u by setting $\mathsf{EMM}_u[y] \leftarrow z$.
5. $\mathbb{C}$ updates $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] \leftarrow \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] + 1$.

$((\mathsf{st}', \vec{\mathtt{v}}); \mathsf{EMM}') \leftarrow \mathbf{2ch_{FB}}.\mathsf{Query}\left(((\mathsf{qop}, \mathtt{label}), \mathsf{st}), \mathsf{EMM}\right)$:

1. $\mathbb{C}$ sends $\kappa := F_K(\mathtt{label})$ to $\mathbb{S}$ and if $\mathtt{label} \in \mathsf{MM}_{\mathsf{st}}$ and $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}] > 0$, $\mathbb{C}$ computes $x := H(K_u, \mathtt{label} \,||\, \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][0])$ and sends $(x, \mathsf{cnt} := \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1])$ to $\mathbb{S}$.
2. $\mathbb{S}$ computes $\{G_\kappa(j \,||\, 0), G_\kappa(j \,||\, 1)\}_{j \in [\ell]}$ and retrieves the 2ℓ associated bins that are sent to $\mathbb{C}$.
3. $\mathbb{S}$ also sends entries $\mathsf{EMM}_u[H(x, 0)], \ldots, \mathsf{EMM}_u[H(x, \mathsf{cnt} - 1)]$ to $\mathbb{C}$.
4. $\mathbb{C}$ decrypts the 2ℓ bins and all cached update operations for $\mathtt{label}$.
5. $\mathbb{C}$ compiles $\vec{\mathtt{v}}$ containing all values tagged with $\mathtt{label}$ in the 2ℓ bins and $\mathtt{Stash}$ and deletes them from the bins and $\mathtt{Stash}$.
6. For each $i = 0, \ldots, \mathsf{cnt} - 1$:
   (a) $\mathbb{C}$ computes $(\mathtt{op}_i, \vec{\mathtt{v}}_i) \leftarrow \mathsf{Dec}(K_{\mathsf{Enc}}, \mathsf{EMM}_u[H(x, i)])$.
   (b) If $\mathtt{op}_i = \mathsf{app}$, append $\vec{\mathtt{v}}_i$ to $\vec{\mathtt{v}}$. If $\mathtt{op}_i = \mathsf{edit}$, set $\vec{\mathtt{v}} \leftarrow \vec{\mathtt{v}}_i$. If $\mathtt{op}_i = \mathsf{rm}$, set $\vec{\mathtt{v}} \leftarrow \perp$. If $\mathtt{op}_i = \mathsf{del}$, remove any values in $\vec{\mathtt{v}}_i$ from $\vec{\mathtt{v}}$.
7. $\mathbb{C}$ adds back $\vec{\mathtt{v}}$ to the 2ℓ bins and $\mathtt{Stash}$, encrypts the bins and uploads them back to $\mathbb{S}$.
8. $\mathbb{C}$ increments the version number $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][0]$ by 1, resets the count $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1]$ to 0 and outputs $\vec{\mathtt{v}}$.

Figure 1: Pseudocode for Construction $\mathbf{2ch_{FB}}$

(which is meant to store future updates temporarily). The client maintains storage of the stash, an initially empty multi-map $\mathsf{MM_{st}}$ and secret keys K and K_u used to access EMM_u.

Update. For an update operation $(\mathtt{op}, \mathtt{label}, \vec{\mathtt{v}})$, the client checks locally if $\mathsf{MM_{st}}[\mathtt{label}]$ is defined. If not, the client sets $\mathsf{MM_{st}}[\mathtt{label}]$ to be $(0, 0)$. $\mathsf{MM_{st}}[\mathtt{label}][0]$ will denote the version of the PRF secret key currently being used for $\mathtt{label}$ and $\mathsf{MM_{st}}[\mathtt{label}][1]$ will denote the number of tuples for $\mathtt{label}$ in the update encrypted structure EMM_u. To cache the update operation, first $\vec{\mathtt{v}}$ is padded with dummies until its length is exactly ℓ. Next, the client computes $x := H(K_u, \mathtt{label} \mid\mid \mathsf{MM_{st}}[\mathtt{label}][0])$, $y := H(x, \mathsf{MM_{st}}[\mathtt{label}][1])$ where H is a keyed hash function and the encryption $z := \mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{op}, \vec{\mathtt{v}}))$ and sends the pair (y, z) to the server that sets $\mathsf{EMM}_u[y] := z$. The client also increments the count at $\mathsf{MM_{st}}[\mathtt{label}][1]$ by one as the number of update tuples for $\mathtt{label}$ in EMM_u has incremented by one. The label version $\mathsf{MM_{st}}[\mathtt{label}][0]$ remains unchanged at the time of Update.

Query. To query for $\mathtt{label}$, the client computes the seed $F_K(\mathtt{label})$. Additionally, the client checks if $\mathsf{MM_{st}}[\mathtt{label}]$ is defined and if $\mathsf{MM_{st}}[\mathtt{label}][1] > 0$. If so, the client computes another seed $x := H(K_u, \mathtt{label} \mid\mid \mathsf{MM_{st}}[0])$ and sends $F_K(\mathtt{label})$ and x to the server. Otherwise, if $\mathsf{MM_{st}}[\mathtt{label}]$ is not defined or $\mathsf{MM_{st}}[\mathtt{label}][1] = 0$, the client only sends the first seed. The server, first, expands the seed $F_K(\mathtt{label})$ to find the 2ℓ bins $\{G_{F_K(\mathtt{label})}(i \mid\mid 0), G_{F_K(\mathtt{label})}(i \mid\mid 1)\}_{i \in [\ell]}$ in the binary trees. Then the server returns the encryptions stored at $\mathsf{EMM}_u[H(x, i)]$ for all $i \in [\mathsf{MM_{st}}[\mathtt{label}][1]]$ along with the encrypted contents of all 2ℓ bins to the client. The server also deletes the encryptions retrieved from EMM_u.

The client decrypts all contents to find all values that are associated with $\mathtt{label}$ in the 2ℓ bins along with values that may be stored in the overflow stash. The client then decrypts the updates returned from EMM_u, applies them locally to the downloaded 2ℓ bins or the overflow stash. These updated 2ℓ bins are re-encrypted with fresh randomness and sent back to the server for storage. All values associated with $\mathtt{label}$ in these 2ℓ bins and the overflow stash are finally returned as the query's answer. The client increments the label version $\mathsf{MM_{st}}[\mathtt{label}][0] := \mathsf{MM_{st}}[\mathtt{label}][0] + 1$ in order to ensure forward privacy for future updates as the server now knows the current seed for $\mathtt{label}$ and EMM_u. The client also resets the count $\mathsf{MM_{st}}[\mathtt{label}][1] := 0$ as there are no unapplied updates for $\mathtt{label}$ in EMM_u.

3.2.1 Security

We present the leakage of $\mathbf{2ch_{FB}}$ against a persistent adversary. At setup, the adversary learns nothing except for the public parameter n. Therefore, $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}) = n$. Let O be any sequence of operations and o be the current operation. Then, the update leakage is $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\ell, \mathtt{uop})$ where $\mathtt{uop}$ leaks that the operation is an update but not anything specific about the type of update operation (as exactly ℓ encrypted values are inserted into a random entry of EMM_u). We observe that our update operations are forward private as the update leakage is independent of all previous operations. Finally, the query leakage $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{leq}(O, o), \mathsf{qop})$. label equality is revealed by retrieving all cached unapplied updates for $\mathtt{label}(o)$ from EMM_u and from the fact that all query operations on the same label, access the same 2ℓ bins from the two-choice hash table Table. As a result, all update and query operations may be linked to operating on the same label. The proof that $\mathbf{2ch_{FB}}$ is $\mathcal{L}$-secure for the leakage function $\mathcal{L}$ described above is in Appendix C.

Theorem 3. *If* SKE *is an IND-CPA-secure encryption and* F, G *are pseudorandom functions,* H *is a keyed hash function modeled as a random oracle*[2]*, then for every* $n \geq \ell \geq 1$*,* $\mathbf{2ch_{FB}}$ *is an adaptive* $\mathcal{L}$*-secure dynamic STE scheme for multi-maps.*

We prove that $\mathcal{L}$ is a dynamic volume-hiding, forward and type-II backward private in Appendix C to get the following:

Theorem 4. *If* SKE *is an IND-CPA-secure encryption and* F, G *are pseudorandom functions,* H *is a keyed hash function modeled as a random oracle,* $\mathbf{2ch_{FB}}$ *is a volume-hiding, forward private and type-II backward private* $\mathcal{L}$*-secure dynamic, encrypted multi-map scheme.*

[2]Note that the random oracle assumption can be removed by replacing H with a pseudo-random function having the client compute and send all the PRF evaluations, instead of sending just a seed to the server.

3.2.2 Efficiency

We split our analysis into amortized and worst case overhead starting with amortized. The amortized communication and computational cost of an update operation is $O(\ell)$. During updates, ℓ encrypted values are inserted into EMM_u. During a query operation, the same ℓ encrypted values are downloaded and decrypted locally. Amortized communication and computational complexity of a query is $O(\ell \log\log n)$ as exactly 2ℓ bins are retrieved where each bin contains $O(\log\log n)$ values.

Next, we consider worst case overhead. For update operations, ℓ encrypted values are always uploaded to EMM_u. The worst case query overhead heavily depends on the number of unapplied update operations. For label `label`, we denote the number of unapplied update operations since the last query for `label` by $u(\texttt{label})$. Then, the worst case overhead of a query operation is $O(\ell \log\log n + \ell u(\texttt{label}))$ from retrieving 2ℓ bins along with applying all prior update operations for `label`.

The server storage consists of $O(n)$ value along with the number of unapplied update operations. While this may be unbounded, we present a variant in Section 3.3.3 where the update operations in EMM_u may be applied every $O(n/\ell)$ update operations to ensure that server storage never exceeds $O(n)$. The client storage consists of the multi-map $\mathsf{MM_{st}}$ requiring at most $O(m)$ storage where m is the number of unique labels. The other portion of client storage is the two-choice hashing overflow stash that requires at most $f(n)$ storage except with probability negligible in n for any function $f(n) = \omega(\log n)$.

3.3 Construction $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$

Next, we present our final scheme $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ (standing for **2-c**hoice **h**ashing with **F**orward and **B**ackward privacy and **s**mall client storage) that is also volume-hiding, forward and type-II backward private like $\mathbf{2ch_{FB}}$. $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ improves upon $\mathbf{2ch_{FB}}$ by using smaller permanent client storage. Recall that $\mathbf{2ch_{FB}}$ uses client storage potentially linear in the number of unique labels $O(m)$. $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ will only require permanent client storage of size $\omega(\log n)$. Recall that $\mathbf{2ch_{FB}}$ required the client to locally store $\mathsf{MM_{st}}$. For any $\texttt{label} \in \mathbb{L}$, $\mathsf{MM_{st}}[\texttt{label}]$ stores two integers; a version number required by the keyed hash H and the number of unapplied update operations that are in EMM_u. Instead, we will outsource the storage of $\mathsf{MM_{st}}$ to the server inspired by ideas from hierarchical ORAMs [32, 13, 24, 28] and recent work by Demertzis *et al.* [11].

In order to get rid of $\mathsf{MM_{st}}$ at the client, $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ will explicitly store the location of cached operations in EMM_u, in a series of static, encrypted multi-maps $\mathsf{EMM}^{\mathsf{loc}}_0, \ldots, \mathsf{EMM}^{\mathsf{loc}}_{t-1}$ of geometrically increasing sizes stored on the server. The number of encrypted multi-maps will be $t = O(\log u)$ where u is the number of previous update operations. For any i, $\mathsf{EMM}^{\mathsf{loc}}_i$ stores at most 2^i cached update operations. We instantiate these t structures using $\mathbf{PiBas}^*$ (a modified version of **PiBas** [8]) that is a static, response-hiding, volume-revealing, encrypted multi-map scheme as described in [11]. We note however that any static encrypted search scheme with setup leakage being the size of the input multi-map and query leakage being at most query equality and volume of the tuple, will suffice as a replacement to $\mathbf{PiBas}^*$.[3] Specifically, $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ maintains the invariant that the encrypted multi-map $\mathsf{EMM}^{\mathsf{loc}}_i$ will store the locations of cached operations per label over the latest update operations that are not stored in smaller encrypted multi-maps, $\mathsf{EMM}^{\mathsf{loc}}_0, \ldots, \mathsf{EMM}^{\mathsf{loc}}_{i-1}$. As smaller encrypted multi-maps are filled, their contents are percolated to larger encrypted multi-maps in an efficient, but amortized, manner. As an example, suppose that all encrypted multi-maps $\mathsf{EMM}^{\mathsf{loc}}_0, \ldots, \mathsf{EMM}^{\mathsf{loc}}_{i-1}$ are fully occupied. For the next update operation, the contents will be combined and placed into the larger encrypted multi-map $\mathsf{EMM}^{\mathsf{loc}}_i$.

By querying all of $\mathsf{EMM}^{\mathsf{loc}}_0[\texttt{label}], \ldots, \mathsf{EMM}^{\mathsf{loc}}_{t-1}[\texttt{label}]$, the client learns the entries of EMM_u that contain all cached update operations. The result is that the client forgoes the storage of $\mathsf{MM_{st}}$ locally at the cost of an additional roundtrip and t additional encrypted multi-map queries.

We now formally present $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ whose the pseudocode may be found in Figure 2.

Setup. The client executes the same setup algorithm as $\mathbf{2ch_{FB}}$ except that the client does not store $\mathsf{MM_{st}}$ locally.

[3] Demertzis *et al.* [11] claim that any static, response-hiding encrypted multi-map would suffice, which is false. It is crucial for forward privacy that setup leakage does not reveal volumes.

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA encryption scheme, $\mathbf{2ch_{FB}}$ be as described in Figure 1 and $\mathbf{PiBas}^*$ (a modified version of **PiBas** [8]) be a static, response-hiding, encrypted multi-map scheme as described in [11].

$(\mathtt{st}; \mathsf{EMM}) \leftarrow \mathsf{Setup}\big(1^\lambda, \mathsf{params} = (n, \ell), \mathsf{MM} = \{(\mathtt{label}_i, \vec{\mathtt{v}}_i)\}_{i \in [|\mathtt{label} \in \mathsf{MM}|]}\big)$:

1. $\mathbb{C}$ executes $\big(\mathsf{st2}, \mathsf{EMM2}\big) \leftarrow \mathbf{2ch_{FB}}.\mathsf{Setup}(1^\lambda, \mathsf{params}, \mathsf{MM})$.
2. $\mathbb{C}$ sets $t = 0$ and stores $\mathsf{st} \leftarrow (t, \mathsf{st2})$, and $\mathbb{S}$ stores $\mathsf{EMM} = (\mathsf{EMM2})$.

$(\mathsf{st}'; \mathsf{EMM}') \leftarrow \mathsf{Update}\left(((\mathtt{op}, \mathtt{label}, \vec{\mathtt{v}}), \mathsf{st}), \mathsf{EMM}\right)$:

1. $\mathbb{C}$ generates random $x \leftarrow \{0,1\}^\lambda$, pads $\vec{\mathtt{v}}$ with dummies until $\vec{\mathtt{v}}$ contains ℓ values and computes $y \leftarrow \mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{op}, \vec{\mathtt{v}}))$.
2. $\mathbb{S}$ stores $\mathsf{EMM}_u[x] \leftarrow y$, finds the smallest i such that $\mathsf{EMM}_i^{\mathsf{loc}}$ is empty or un-initialized and sends $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$ to $\mathbb{C}$.
3. For each $j \in \{0, \ldots, i-1\}$, $\mathbb{C}$ decrypts each $\mathsf{EMM}_j^{\mathsf{loc}}$ using $\mathtt{st}_j$ to obtain MM_j.
4. For each $\mathtt{label}'$ appearing in at least one of the MM_j, $\mathbb{C}$ computes MM_i such that $\mathsf{MM}_i[\mathtt{label}'] = (\mathsf{EMM}_0^{\mathsf{loc}}[\mathtt{label}'], \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}[\mathtt{label}'])$.
5. $\mathbb{C}$ incorporates the current update operation by appending x to $\mathsf{MM}_i[\mathtt{label}]$.
6. $\mathbb{C}$ executes $(\mathtt{st}_i; \mathsf{EMM}_i^{\mathsf{loc}}) \leftarrow \mathbf{PiBas}^*.\mathsf{Setup}(1^\lambda, \mathsf{MM}_i)$.
7. $\mathbb{C}$ updates $\mathtt{st}$ by removing $\mathtt{st}_0, \ldots, \mathtt{st}_{i-1}$ and adding $\mathtt{st}_i$. If $i \geq t$, $\mathbb{C}$ updates $\mathtt{st}$ by setting $t \leftarrow i+1$.
8. $\mathbb{C}$ sends $\mathsf{EMM}_i^{\mathsf{loc}}$ to $\mathbb{S}$. $\mathbb{S}$ empties all of $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$ and adds $\mathsf{EMM}_i^{\mathsf{loc}}$ to EMM.

$((\mathsf{st}', \vec{\mathtt{v}}); \mathsf{EMM}') \leftarrow \mathsf{Query}\left(((\mathsf{qop}, \mathtt{label}), \mathsf{st}), \mathsf{EMM}\right)$:

1. $\mathbb{C}$ executes $\mathbf{PiBas}^*.\mathsf{Query}$ for $\mathtt{label}$ to all non-empty $\mathsf{EMM}_i^{\mathsf{loc}}$ to obtain $\mathsf{EMM}_i^{\mathsf{loc}}[\mathtt{label}]$ and sets $L \leftarrow (\mathsf{EMM}_0^{\mathsf{loc}}[\mathtt{label}], \ldots, \mathsf{EMM}_{t-1}^{\mathsf{loc}}[\mathtt{label}])$. $\mathbb{S}$ removes all entries in L from their corresponding $\mathsf{EMM}_i^{\mathsf{loc}}$.
2. $\mathbb{C}$ and $\mathbb{S}$ execute $((\mathsf{st2}', \vec{\mathtt{v}}); \mathsf{EMM2}') \leftarrow \mathbf{2ch_{FB}}.\mathsf{Query}(((\mathsf{qop}, \mathtt{label}), \mathsf{st2}); \mathsf{EMM2})$. In the execution, $\mathbb{C}$ uses L as the locations of cached update operations for $\mathtt{label}$ in EMM_u instead of sending a seed to $\mathbb{S}$ to compute these locations.
3. $\mathbb{C}$ computes $\mathtt{st}'$ by updating $\mathsf{st2}$ to $\mathsf{st2}'$ and $\mathbb{S}$ computes EMM' by updating $\mathsf{EMM2}$ to $\mathsf{EMM2}'$.

Figure 2: Pseudocode for construction $\mathbf{2ch_{FB}^{s}}$

Update. For an update operation $(\texttt{op}, \texttt{label}, \vec{v})$, the client chooses a random location x and stores an encryption of the current update operation at $\mathsf{EMM}_u[x]$ after padding $\vec{v}$ to be length ℓ. To store x, the client identifies the smallest, empty multi-map. Say, this is $\mathsf{EMM}_i^{\mathsf{loc}}$. Next, the client downloads all $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$, decrypts them locally and combines all counts into a single multi-map MM_i. For each $\texttt{label}'$ that appears in one of the i downloaded encrypted multi-maps, the client sets $\mathsf{MM}_i[\texttt{label}'] = (\mathsf{EMM}_0^{\mathsf{loc}}[\texttt{label}'], \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}[\texttt{label}'])$. The random location of the current update x is also appended to $\mathsf{MM}_i[\texttt{label}]$. MM_i is then encrypted using the setup algorithm of **PiBas*** or a valid replacement and sent to $\mathbb{S}$ for storage as $\mathsf{EMM}_i^{\mathsf{loc}}$ while all $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$ are emptied.

Query. For a query to $\texttt{label} \in \mathbb{L}$, $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ performs t queries with the server $\mathbb{S}$ to retrieve the locations of all cached update operations for $\texttt{label}$ in EMM_u. Afterwards, $\mathbb{C}$ uses the same algorithm as $\mathbf{2ch}_{\mathbf{FB}}.\mathsf{Query}$ to retrieve the final result. The only difference being that instead of sending a seed to $\mathbb{S}$ to compute the locations in EMM_u, $\mathbb{C}$ sends the locations directly.

3.3.1 Security

We present the leakage profile for $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ when each $\mathsf{EMM}_i^{\mathsf{loc}}$ is initialized by **PiBas***. While one could present generic leakage, we choose to present leakage of a specific instantiation for ease of readability. Recall this construction has setup leakage of simply the total number of values and query leakage of label-equality and the queried label volume. Let MM be the input multi-map, O be a operation sequence and o be the current operation. The setup leakage of $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ is identical to $\mathbf{2ch}_{\mathbf{FB}}$ as the adversary's view is the same. Therefore, $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}) = n$. In terms of update leakage, the server learns information about which encrypted multi-maps are downloaded and uploaded by the client. Note, this is a pre-determined schedule depending only on the number of previous updates. So, the update leakage is $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\ell, \texttt{uop})$ which is also same as $\mathbf{2ch}_{\mathbf{FB}}$. Finally, the query leakage of $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ is similar to $\mathbf{2ch}_{\mathbf{FB}}$ but it also has an extra leakage of queries on $\mathsf{EMM}_i^{\mathsf{loc}}$. Lets call this extra leakage $\mathcal{L}_{\mathsf{loc}}$. Therefore, $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{leq}(O, o), \mathsf{qop}, \mathcal{L}_{\mathsf{loc}})$. The proof that $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ is $\mathcal{L}$-secure for the leakage function $\mathcal{L}$ described above is in Appendix D.

Theorem 5. *If* SKE *is an IND-CPA-secure encryption, then for every* $n \geq \ell \geq 1$, $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ *is an adaptive* $\mathcal{L}$*-secure dynamic STE scheme for multi-maps.*

We prove that $\mathcal{L}$ is a dynamic volume-hiding, forward and type-II backward private in Appendix D to get the following:

Theorem 6. *If* SKE *is an IND-CPA-secure encryption,* $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ *is a volume-hiding, forward private and type-II backward private* $\mathcal{L}$*-secure dynamic, encrypted multi-map scheme.*

3.3.2 Efficiency

We start with the main improvement of $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ over $\mathbf{2ch}_{\mathbf{FB}}$ that is client storage. The client storage of $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ becomes only the overflow stash of size at most $f(n)$ for any function $f(n) = \omega(\log n)$ except with negligible probability. In Section 4, we show the overflow stash never exceeded more than a couple of items at a time through experimental evaluation. We note that client storage may be temporarily higher during operation time if and when rebuilding (discussed in Section 3.3.3) is required. The additional server storage consists of $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{t-1}^{\mathsf{loc}}$ that stores at most $|\mathsf{Update}(O)|$ values. So, $\mathbf{2ch}^{\mathsf{s}}_{\mathbf{FB}}$ has identical worst case client storage cost as $\mathbf{2ch}_{\mathbf{FB}}$.

Note, the only additional query and update overhead costs consist of the downloading, uploading, constructing and querying the encrypted multi-maps used to store counts. Consider the encrypted multi-map $\mathsf{EMM}_i^{\mathsf{loc}}$ that stores at most 2^i counts. We note that $\mathsf{EMM}_i^{\mathsf{loc}}$ is downloaded and re-uploaded when $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$ are full. This occurs every 2^i update operations. For u update operations, the total cost of $\mathsf{EMM}_i^{\mathsf{loc}}$ is $O(2^i \cdot u/2^i) = O(u)$ or $O(1)$ amortized cost across all 2^i update operations. Over all t encrypted multi-maps, the amortized cost of update operations becomes $O(t)$. As we set $t = O(\log u)$ where u is the number of prior update operations, the additional additive cost is $O(\log n)$ assuming at most a polynomial number of operations $u = \texttt{poly}(n)$. The cost of querying each $\mathsf{EMM}_i^{\mathsf{loc}}$ is equivalent to $O(\log n + u(\texttt{label}))$

as $O(\log n)$ queries occur and a total of $u(\texttt{label})$ encrypted values are retrieved in the worst case. Therefore, the worst case communication and computational cost of queries are $O(\ell \log\log n + \ell u(\texttt{label}) + \log n)$. In terms of amortized cost, we note that each update operation incurs $O(\ell)$ overhead at update time by writing ℓ encrypted values into the smallest multi-map. As each of these encrypted values may move up through the $O(\log n)$ levels, the amortized update cost may be viewed as $O(\ell \log n)$. Through this lens, the amortized query overhead remains $O(\ell \log\log n)$.

3.3.3 Variants

Note that both $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ require server storage linear in the number of update operations in the worst case (i.e. the updated labels are never queried). Moreover, the static structures $\mathsf{EMM}_i^{\mathsf{loc}}$ in $\mathbf{2ch^s_{FB}}$ do not take into account the space wasted due to resolved updates. We show that one can ensure the server storage stays at $O(n)$ using a scheduled clean-up algorithm. Every $O(n/\ell)$ update operations, the client and server agree to perform a scheduled clean-up. The client downloads the entire encrypted storage, decrypts locally, applies all cached update operations and re-uploads a freshly encrypted version of the two-choice hash table. As a result, the server storage never exceeds $O(n)$. Furthermore, the additional amortized cost of each update operation increases by $O(\ell)$ that does not increase the total cost. Also particularly for $\mathbf{2ch^s_{FB}}$, one can modify the update algorithm in such a way that $\mathsf{EMM}_i^{\mathsf{loc}}$ that is selected to be locally reconstructed is one that has some space newly freed due to a deletion of an entry from $\mathsf{EMM}_i^{\mathsf{loc}}$ during update resolution in a past query operation.

4 Experimental Evaluation

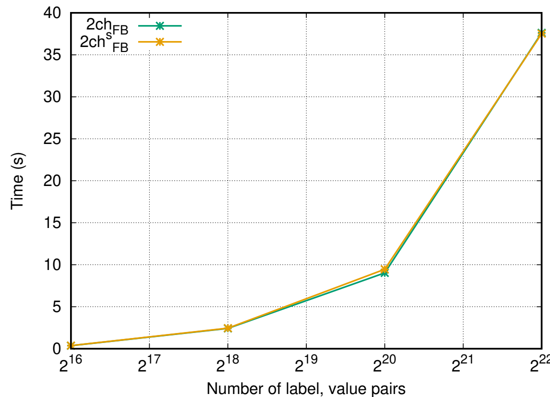


Figure 3: Time taken by the Setup protocol for $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$.

In this section, we evaluate the practicality of our volume-hiding schemes: $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$. We start by describing the setup of our experiments as well as the choice of parameters of our constructions. By performing these experiments, we attempt to answer two important questions. Firstly, are our constructions, with better privacy guarantees than previous schemes, concretely efficient as well? Secondly, what is the total cost of our constructions?

4.1 Experimental Setup

Our experiments are performed using the same machine for both the client and the server; a Ubuntu PC with an Intel(R) Core(TM) i5-9400 CPU with 6 cores, and 64 GB of RAM. Our schemes are implemented in Rust in about 500-800 lines of code each. Both schemes are instantiated in-memory. All the results of

	Dense Subgraph Transform [20]				$\mathbf{2ch_{FB}}$				$\mathbf{2ch^{s}_{FB}}$			
Input Multi-Map Number of Values (n)	2^{16}	2^{18}	2^{20}	2^{22}	2^{16}	2^{18}	2^{20}	2^{22}	2^{16}	2^{18}	2^{20}	2^{22}
EMM Storage												
Server (MB)	5.53	22.74	88.25	384.40	7.99	44.78	167.90	634.98	13.52	66.59	255.14	983.95
Client (KB)	< 1	< 1	< 1	< 1	4.09	11.58	32.76	92.68	< 1	< 1	< 1	< 1

Table 2: Observed sizes of structures. We denote n as the total number of label, value pairs in the input multi-map.

our experiments have standard deviations less than 2% of their average and have been repeated at least 10 times. We will make our code publicly available.

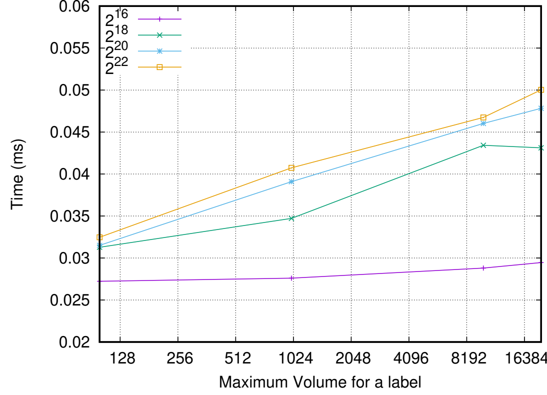
Primitives. We use and build on top of cryptographic primitives provided by ring [6] and OPENSSL [35] rust crates. For symmetric encryption, we use AES in CTR mode with key of size 32 bytes. In all our experiments, we consider PRFs with 32 byte outputs. In particular, we implement our PRFs using HMAC with SHA256.

Input Multi-Maps. In our experiments, we will consider general multi-maps containing $n \in \{2^{16}, 2^{18}, 2^{20}, 2^{22}\}$ maximum values which are considered standard in the literature [11, 31]. As our schemes are dynamic, we initialize our input multi-maps with 90% of their maximum capacity. The final 5% is set aside to support updates. Since we are trying to guage efficiency of volume-hiding schemes, we set the number of unique labels to be $n/100$ so that volume of labels are large and also comparable to experiments in other works such as [11]. Maximum size of label and value strings will be 20 bytes. Other measurements are discussed in the following subsections.

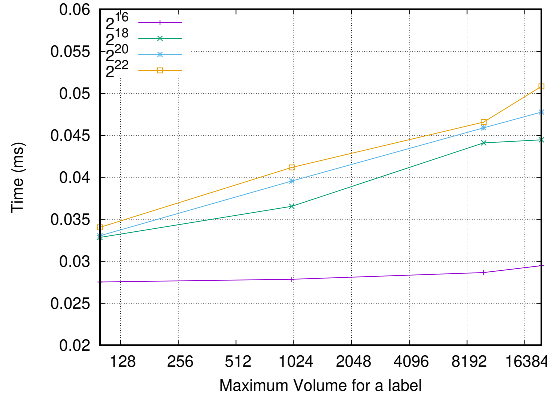
Setup Protocol. The time taken by the setup algorithm of both $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ ranges from 0.35s to 36.7s as the size of the input multi-map increases. For our experiments, we set the value for the parameters $c = 1$ as throughout our experiments on different multi-map sizes, our client stash never exceeded more than a couple of items at a time. We refer the readers to Figure 3 for a detailed plot of setup times.

Query Protocol. Next, we computed the time a query takes on average for both schemes. The query time refers to the total time taken by the client and the server collectively to produce a final result. In our experiments, for each data point, we would do multiple rounds of three queries on the same label but each time we would increase the number of updates done on that label prior to a round of queries. At a certain point the volume of the label would approach the maximum volume set for that particular instantiation of the scheme and we would stop updating further. We would then take the average of query times for this label across these rounds. This experiment is done in this way to factor in the effect of updates on query times. Figures 4a and 4b are side by side plots of query times for different input multi-map sizes against different maximum volumes. Note that the query time in these graphs are per result where the number of results for a query is the maximum volume. This is done so that a direct comparison to the static volume hiding schemes in [31] can be made. Here we note that the query times are comparable to the query times in [31] even though our schemes support dynamic operations. The query times of $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ are also very comparable with each other, surprisingly, even though updates are stored differently in both schemes. For both schemes the query times ranged from 0.027ms to 0.051ms per result as the maximum volume and the size on the input multi-map increased.

We now refer the readers to Figures 5 and 6 for a detailed look at query times. The purpose of these graphs is to show the effects updates have on a query. In each of the graphs in these figures, there are 9 queries issued and the x-axis represents the ith query of the 9 queries. The y-axis represents the total query time for each query. Right before the first, fourth and seventh query on a label, there were 10 , 50 and 100 updates made on that label, respectively. The size of each update was randomly sampled. The figures, hence show small spikes in the first, fourth and seventh query times because of unresolved updates at those times and sudden speed up of the following two queries. We note that $\mathbf{2ch^{s}_{FB}}$ which saves a lot on client storage,



(a) $\mathbf{2ch_{FB}}$



(b) $\mathbf{2ch^{s}_{FB}}$

Figure 4: Query time for different values of ℓ and different database sizes.

tends to be comparable but slower than $\mathbf{2ch_{FB}}$. This is because its query protocol takes two rounds and has to do considerably more rebuilding that $\mathbf{2ch_{FB}}$. We, however, note that our query times are in order of microseconds (25μs to 50μs) per single label, value pairs for both $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$.

Update Protocol. We measured the time taken during our updates. We note that for $\mathbf{2ch_{FB}}$, our updates stay under 25ms and for $\mathbf{2ch^{s}_{FB}}$ under 76ms even for maximum volume of $20,000$. This is primarily becuase of forward and backward privacy, updates are not directly applied to the two-choice hashing structure and some of the work is postponed, until queries. The updates for $\mathbf{2ch^{s}_{FB}}$ are costly compared to $\mathbf{2ch_{FB}}$ as expected because unlike $\mathbf{2ch_{FB}}$ where a tuple is directly inserted into an encrypted multi-map, in $\mathbf{2ch^{s}_{FB}}$ mutiple encrypted structures are downloaded and rebuilt which takes extra time. However, $\mathbf{2ch^{s}_{FB}}$ is still very efficient and a great candidate for a scenario where permanent client state is not desirable.

Comparison with the Dense Subgraph Transform [20]. We compare our schemes with the Dense Subgraph Transform construction described in [20] as it is the only existing dynamic, volume-hiding scheme. The other construction in the same work is based on Pseudo-Random Transform [20], but is lossy in nature (i.e., tuples in the multi-map are subject to random truncations and hence query results might not be accurate). Hence, we do not believe a fair comparison is possible with that scheme.

We note that the Dense Subgraph Transform construction lacks several features offered by our constructions such as no forward privacy and only semi-dynamic operations[4]. For a comprehensive treatment, we

[4]For more details about the semi-dynamicity of this scheme, see Appendix A.

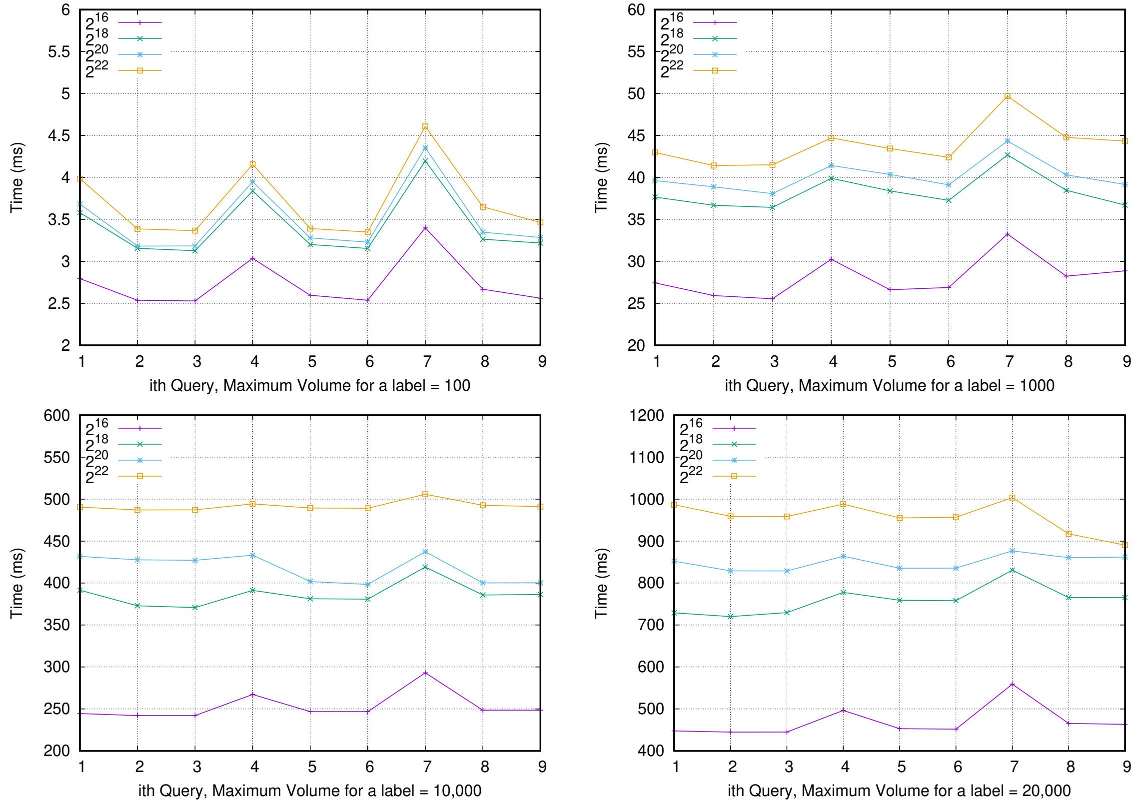


Figure 5: This is the time cost when executing queries in $\mathbf{2ch_{FB}}$. We computed the times for $\ell \in \{100, 1000, 10000, 20000\}$ where ℓ is the maximum volume any label can have. For each value of ℓ, we executed queries over varying database sizes as shown in the graphs.

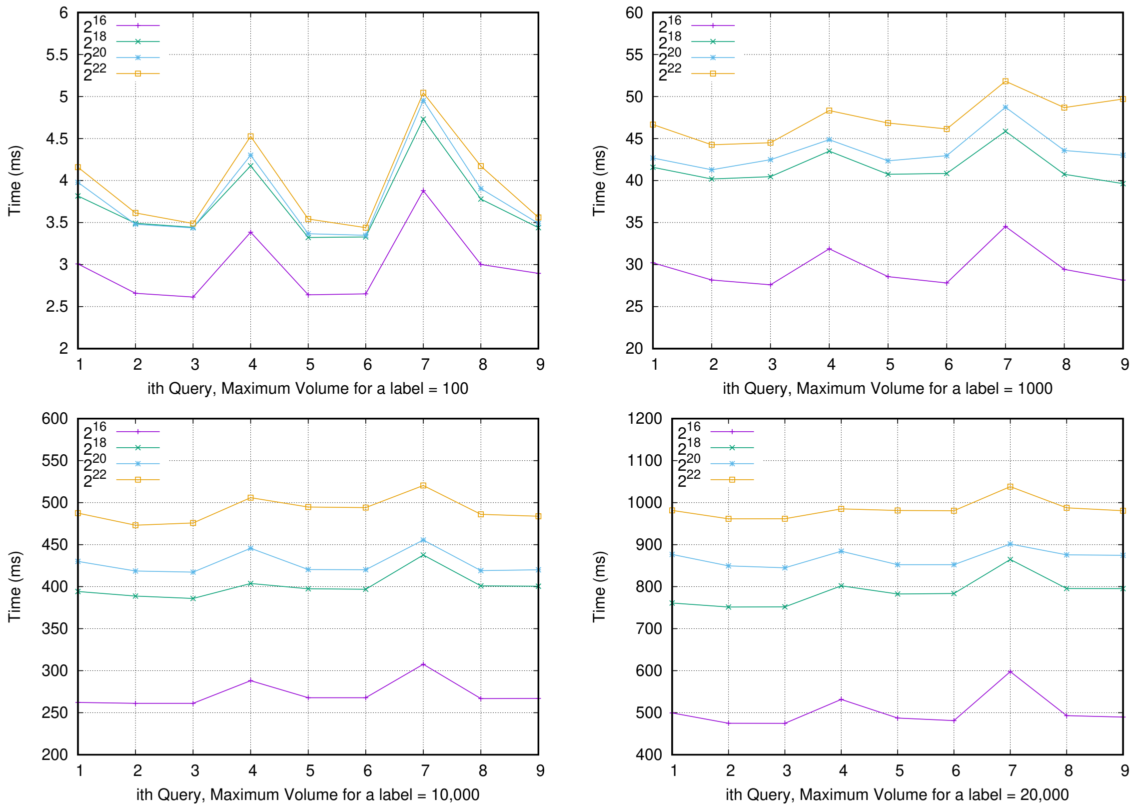
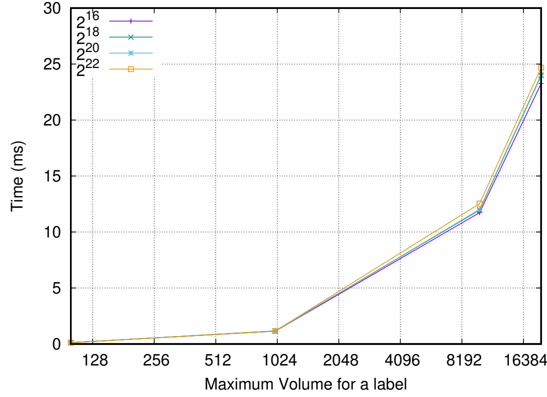
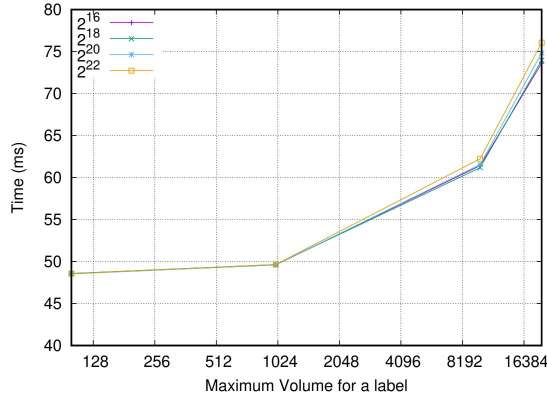


Figure 6: Similar to Fig. 5, this figure represent total time taken when executing queries in $\mathbf{2ch^{s}_{FB}}$.



(a) $\mathbf{2ch_{FB}}$



(b) $\mathbf{2ch^s_{FB}}$

Figure 7: Update time for different values of ℓ and different database sizes.

still present a comparison between our constructions and the Dense Subgraph Transform. We will show that our constructions $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ offer these additional functionalities with minimal (or no) increase in costs compared to the Dense Subgraph Transform.

Starting with the simple case when ignoring updates, our schemes $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ improve the communication during queries by 2-3x. When incorporating updates, the query communication cost increases with the number of updates and, thus, may be worse than the Dense Subgraph Transform scheme when considering sequences with a lot of updates. This is not surprising as our queries are responsible for resolving updates (as done in most forward/backward private schemes). However, the computational cost of queries remains similar even when resolving updates. Furthermore, each update is only resolved by a single query, so future queries do not incur the same cost. For use cases with a lot of queries, our schemes end up being more efficient than the Dense Subgraph Transform. In settings where queries occur more frequently than updates, our constructions will end up outperforming the Dense Subgraph Transform.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020*, pages 403–432, 2020.

[2] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[3] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS 2020*, 01 2020.

[4] Raphael Bost. Sophos: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1143–1154. ACM, 2016.

[5] Raphael Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1465–1482. ACM, 2017.

[6] briansmith. RING. `https://docs.rs/ring/0.17.0-alpha.1/ring/index.html`.

[7] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS '15*, 2015.

[8] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26, 2014.

[9] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 577–594, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS '06*, pages 79–88, 2006.

[11] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. Cryptology ePrint Archive, Report 2019/1227, 2019.

[12] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3), 1996.

[13] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 157–167. SIAM, 2012.

[14] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 315–331, 2018.

[15] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1353–1364. ACM, 2016.

[16] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 655–672. IEEE, 2017.

[17] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 361–378, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[19] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 149–180. Springer, 2018.

[20] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019*, pages 183–213, 2019.

[21] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1329–1340, 2016.

[22] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

[23] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *2021 IEEE Symposium on Security and Privacy, SP 2021, May 24-27, 2021*. IEEE, 2021.

[24] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[25] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[26] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 644–655, 2015.

[27] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[28] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS '18*, 2018.

[29] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '19, pages 182–199, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Leakage cell probe model: Lower bounds for key-equality mitigation in encrypted multi-maps. In CRYPTO 2020, 2020. https://eprint.iacr.org/2019/1132.

[31] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 79–93, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Annual Cryptology Conference*, pages 502–519. Springer, 2010.

[33] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS '16*, 2016.

[34] Daniel S Roche, Adam Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 178–197. IEEE, 2016.

[35] Rust. OPENSSL. `https://docs.rs/openssl/0.10.29/openssl/`.

[36] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55, 2000.

[37] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, volume 71, pages 72–75, 2014.

[38] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, pages 707–720, 2016.

A Semi-Dynamicity of Dense Subgraph Transform [20]

Throughout our work, we refer to the Dense Subgraph Transform construction of Kamara and Moataz [20] as semi-dynamic. In particular, the construction only provides adding, deleting or overwriting an entire value tuple associated with a label. The missing functionality is appending or removing values from an existing value tuple. The acute reader might note that one could implement this using two semi-dynamic EMM operations: querying the value tuple, modifying the value tuple locally and updating the entire value tuple. While this achieves the desired functionality, it degrades privacy significantly. A recent work [30] shows that unless one is willing to utilize ORAM-like overheads, leakage of label equality patterns must be revealed by queries. Label equality patterns reveal whether two different operations are performed on the same label or not. Furthermore, this label equality leakage ends up being a significant privacy degradation. For example, label equality leakage violates the privacy requirements of being forward private. Therefore, the above transformation requires either the EMM to use ORAM-like overhead or not provide forward privacy. In our work, we avoid this problem by directly building update operations that avoid performing query operations.

B Pseudocode and Analysis of 2ch

Pseudocode of **2ch** is presented in Figure 8.

B.1 Security

We present the leakage profile for our scheme against a persistent adversaries. During Setup, no information is leaked about the plaintext MM other than an upper bound n on the total number of values stored and hence $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}) = n$. As far as query and update operations are concerned, we observe that operations on the same label access the same 2ℓ bins. So, the adversary may link different operations as operating on the same label or not. Moreover, update operations write back the bins accessed whereas query operations do not and so the type of operation, update or query, is also leaked. For a sequence of operations O and the last operation o, we have query leakage $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = (\mathsf{op}(o), \mathsf{leq}(O, o), \ell)$ and update leakage $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\mathsf{op}(o), \mathsf{leq}(O, o), \ell)$. We now prove that following theorem for **2ch**.

Let F and G be PRFs and $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA encryption scheme.

$(\mathsf{st}; \mathsf{EMM}) \leftarrow \mathbf{2ch}.\mathsf{Setup}\big(1^\lambda, \mathsf{params} = (n, \ell), \mathsf{MM} = \{(\mathtt{label}_i, \vec{\mathsf{v}}_i)\}_{i \in [m]}\big)$:

1. $\mathbb{C}$ randomly selects a PRF key $K \leftarrow \{0,1\}^\lambda$ and generates $K_{\mathsf{Enc}} \leftarrow \mathsf{Gen}(1^\lambda)$.
2. $\mathbb{C}$ creates $s := \lceil n/(c \log n) \rceil$ full binary trees, $\mathtt{Table} \leftarrow (B_1, \ldots, B_s)$ each of height $h := \lceil \log(c \log n) \rceil$. Roots are at level 0 and leaf nodes are at height h. Each node has the capacity to hold a single encryption. Each of the n bins are uniquely assigned to n different leaf nodes.
3. $\mathbb{C}$ initializes $\mathtt{Stash} \leftarrow \emptyset$.
4. For each $\mathtt{label}_i \in \mathsf{MM}$:
   (a) Compute $x \leftarrow F_K(\mathtt{label}_i)$ and for each $j \in [|\vec{\mathsf{v}}_i|]$:
      i. $\mathbb{C}$ computes $b_0 \leftarrow G_x(j \mid\mid 0)$ and $b_1 \leftarrow G_x(j \mid\mid 1)$ and locates the two leaf-to-root paths associated with bins b_0 and b_1.
      ii. $\mathbb{C}$ computes $\mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{label}_i, j, \vec{\mathsf{v}}[j]))$ and places it into the empty node at the highest level in either bin b_0 or b_1.
      iii. If both bin b_0 and bin b_1 contain no empty nodes, add $(\mathtt{label}_i, j, \vec{\mathsf{v}}[j])$ to $\mathtt{Stash}$.
5. For all empty nodes in the binary trees, $\mathbb{C}$ adds a fresh encryption of $\mathsf{Enc}(K_{\mathsf{Enc}}, (\perp, \perp, \perp))$.
6. $\mathbb{C}$ sets its state $\mathsf{st} \leftarrow (K, K_{\mathsf{Enc}}, \mathtt{Stash})$ and sets $\mathsf{EMM} \leftarrow (B_1, \ldots, B_s)$.

$((\mathsf{st}', \vec{\mathsf{v}}); \mathsf{EMM}') \leftarrow \mathbf{2ch}.\mathsf{Query}\left((\mathsf{st}, (\mathsf{qop}, \mathtt{label}))\,, \mathsf{EMM}\right)$.

1. $\mathbb{C}$ parses st as $(K, K_{\mathsf{Enc}}, \mathtt{Stash})$, and $\mathbb{S}$ parses EMM as $(B_1, \ldots, B_s)$.
2. $\mathbb{C}$ computes $x \leftarrow F_K(\mathtt{label})$ that is sent to the $\mathbb{S}$.
3. $\mathbb{S}$ computes $\{G_x(i \mid\mid 0), G_x(i \mid\mid 1)\}_{i \in [\ell]}$ and retrieves the 2ℓ associated bins that are sent to $\mathbb{C}$.
4. $\mathbb{C}$ decrypts all 2ℓ bins and returns $\vec{\mathsf{v}}$ consisting of all values that are tagged with $\mathtt{label}$ in the bins as well as in $\mathtt{Stash}$.

$(\mathsf{st}'; \mathsf{EMM}') \leftarrow \mathbf{2ch}.\mathsf{Update}\left((\mathsf{st}, (\mathsf{op}, \mathtt{label}, \vec{\mathsf{v}}'))\,, \mathsf{EMM}\right)$:

1. $\mathbb{C}$ computes $x \leftarrow F_K(\mathtt{label})$ that is sent to the $\mathbb{S}$.
2. $\mathbb{S}$ computes $\{G_x(i \mid\mid 0), G_x(i \mid\mid 1)\}_{i \in [\ell]}$ and retrieves the 2ℓ associated bins that are sent to $\mathbb{C}$.
3. $\mathbb{C}$ decrypts all 2ℓ bins and compiles $\vec{\mathsf{v}}$ consisting of all values that are tagged with $\mathtt{label}$ that are removed the downloaded bins.
4. $\mathbb{C}$ checks $\mathtt{Stash}$ for any values also tagged with $\mathtt{label}$ that should be added to $\vec{\mathsf{v}}$. All entries corresponding to $\mathtt{label}$ are removed from $\mathtt{Stash}$.
5. If $\mathsf{op} = \mathsf{app}$, $\mathbb{C}$ appends $\vec{\mathsf{v}}'$ to $\vec{\mathsf{v}}$. If $\mathsf{op} = \mathsf{edit}$, $\mathbb{C}$ sets $\vec{\mathsf{v}} \leftarrow \vec{\mathsf{v}}'$. If $\mathsf{op} = \mathsf{del}$, $\mathbb{C}$ removes the values in $\vec{\mathsf{v}}'$ from $\vec{\mathsf{v}}$. If $\mathsf{op} = \mathsf{rm}$, $\mathbb{C}$ sets $\vec{\mathsf{v}} \leftarrow \perp$.
6. For $i \in [|\vec{\mathsf{v}}|]$:
   (a) $\mathbb{C}$ computes $b_0 \leftarrow G_x(i \mid\mid 0)$ and $b_1 \leftarrow G_x(i \mid\mid 1)$.
   (b) $\mathbb{C}$ locally checks bin b_0 and bin b_1, finds highest level node with a dummy encryption and replaces the dummy with $\mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{label}, i, \vec{\mathsf{v}}[i]))$.
   (c) If both bin b_0 and bin b_1 contain no nodes with dummy encryptions, add $(\mathtt{label}, i, \vec{\mathsf{v}}[i])$ to $\mathtt{Stash}$.
7. $\mathbb{C}$ re-encrypts all 2ℓ bins and sends back to $\mathbb{S}$ for storage.

Figure 8: Pseudocode for Construction **2ch**

Theorem 7. *If* SKE *is an IND-CPA-secure encryption and* F, G *are pseudorandom functions, then for every* $n \geq \ell \geq 1$, **2ch** *is a volume-hiding and type-II backward private,* $\mathcal{L}$*-secure dynamic STE scheme for multi-maps.*

In order to prove this Theorem 7, we first prove Theorem 8, Lemma 1 and Lemma 2.

Theorem 8. *If* SKE *is an IND-CPA-secure encryption and* F, G *are pseudorandom functions, then for every* $n \geq \ell \geq 1$, **2ch** *is an adaptive* $\mathcal{L}$*-secure dynamic STE scheme for multi-maps.*

Proof of Theorem 8. We consider a stateful simulator $\mathcal{S}$ with state $\mathtt{st}$ that works as follows:
$\mathsf{EMM} \leftarrow \mathcal{S}.\mathsf{SimSetup}(1^\lambda, n)$**:**

1. Construct $s = \lceil n/c\log(n) \rceil$ full binary trees $B_1, \ldots, B_s$ each with height $h = \lceil \log(c \log n) \rceil$.
2. Fill each node of every tree with an encryption of $\perp$.
3. Return $B_1, \ldots, B_s$.

$\mathsf{Response} \leftarrow \mathcal{S}.\mathsf{SimQuery}(1^\lambda, \mathsf{leq}(O, o), \ell)$**:**

1. Using $\mathsf{leq}(O, o)$, find smallest i such that $\mathtt{label}(o) = \mathtt{label}(O[i])$.
2. If no such i exists, set $\mathtt{st}[|O| + 1]$ to be a uniformly random string from $\{0, 1\}^\lambda$ and set $i \leftarrow |O| + 1$.
3. Return $\mathtt{st}[i]$.

$\mathsf{Response} \leftarrow \mathcal{S}.\mathsf{SimUpdate}(1^\lambda, \mathsf{leq}(O, o), \ell)$**:**

1. Using $\mathsf{leq}(O, o)$, find smallest i such that $\mathtt{label}(o) = \mathtt{label}(O[i])$.
2. If no such i exists, set $\mathtt{st}[|O| + 1]$ to be a uniformly random string from $\{0, 1\}^\lambda$ and set $i \leftarrow |O| + 1$.
3. Return $\mathtt{st}[i]$.
4. Return $2 \cdot \ell$ arrays $(A_{0,i}, A_{1,i})_{i \in [1,\ell]}$ of size h each. Each entry of the array is an encryption of $\perp$.

We now show that for all PPT adversaries $\mathcal{A}$, the probability that $\mathbf{Real}_{\mathbf{2ch},\mathcal{A}}(1^\lambda)$ outputs 1 is negligibly different from the probability that $\mathbf{Ideal}_{\mathbf{2ch},\mathcal{A},\mathcal{S}}(1^\lambda)$ outputs 1. To do this, we use the following sequence of games:

- $\mathbf{Game}_0$ is identical to $\mathbf{Real}_{\mathbf{2ch},\mathcal{A}}(1^\lambda)$.
- $\mathbf{Game}_1$ replaces the PRF F with a random function. This is indistinguishable from $\mathbf{Game}_0$ because of the pseudo-randomness of F.
- $\mathbf{Game}_2$ replaces the IND-CPA encryption $\mathsf{SKE.Enc}$ steps with encryptions of $\perp$ that are indistinguishable due to IND-CPA guarantees.
- $\mathbf{Game}_3$ replaces the outputs of random functions with uniformly random chosen values. This is indistinguishable from $\mathbf{Game}_2$ as the output of random function and a random string are indistinguishable.

$\mathbf{Game}_3$ is the same as the ideal experiment completing the proof. □

Next we will prove that **2ch** is volume-hiding.

Lemma 1. *Leakage function* $\mathcal{L}$ *is volume-hiding.*

Proof. To prove that $\mathcal{L}$ is volume-hiding, we consider any two multi-maps with the number of values $\leq n$ and with maximum volume of a label $\leq \ell$. Note that the only other leakage is the label-equality pattern which is independent of the input maps as well as the response lengths of the query operations even after updates. As a result, the input to the adversary in both games with different multi-maps is identical, completing the proof. □

Next we will prove that **2ch** is type-II backward private.

Lemma 2. *Leakage function $\mathcal{L}$ is type-II backward private.*

Proof. Note that $\mathcal{L}_{\mathsf{Update}}$ is dependent on the public parameter ℓ and the label on which the update is being performed. The leakage during queries on previous updates is the timestamps of all previous updates via leq. This leakage profile falls under the definition of type-II backward privacy. □

Proof of Theorem 7. Follows directly from Theorem 8, Lemma 1 and Lemma 2. □

B.2 Efficiency

Communicational and computational query and update operations are $O(\ell \log\log(n))$ as the client uploads a single PRF evaluations and uploads and/or downloads 2ℓ bins of size $O(\log\log n)$. By the analysis of [29], the overflow stash in client storage contains at most $f(n)$ values, for any function $f(n) = \omega(\log n)$, except with probability negligible in n. Server storage for our scheme is $\lceil n/(c\log n)\rceil \cdot \lceil \log(c\log n)\rceil = O(n)$ encrypted values. If we had used standard two-choice hashing, server storage would be $O(n\log\log n)$ without a client stash.

B.3 Variants

In our pseudocode, query and update algorithms are distinguishable since query algorithms are non-interactive while update algorithms are interactive. If we wish to hide operational types from the adversary, we can modify the query algorithm in the following way. After receiving the 2ℓ bins, the query algorithm re-encrypts all values in 2ℓ bins and re-uploads them back to the server. The resulting variant of **2ch** will ensure that adversaries cannot distinguish between query and update algorithms.

C Security Proof of $\mathbf{2ch_{FB}}$

Proof of Theorem 3. We consider a stateful simulator $\mathcal{S}$ with state st that works as follows:
$\mathsf{EMM} \leftarrow \mathcal{S}.\mathsf{SimSetup}(1^\lambda, n)$**:**

1. Construct $s = \lceil n/c\log(n)\rceil$ full binary trees $B_1, \dots, B_s$ each with height $h = \lceil \log(c\log n)\rceil$.
2. Fill each node of every tree with an encryption of $\perp$ and initialize an empty multi-map EMM_u.
3. Set $\mathtt{st} \leftarrow (\mathsf{M}, U)$ where M is an empty map and U is an empty array.
4. Return $(B_1, \dots, B_s, \mathsf{EMM}_u)$.

$\mathsf{Response} \leftarrow \mathcal{S}.\mathsf{SimQuery}(1^\lambda, \mathsf{qop}, \ell, \mathsf{leq}(O, o))$**:**

1. Using $\mathsf{leq}(O, o)$, find smallest i such that $\mathtt{label}(o) = \mathtt{label}(O[i])$ and $O[i]$ is a query.
2. If no such i exists, set $i \leftarrow |O| + 1$ and $\mathsf{M}[i]$ to be a uniformly random string from $\{0,1\}^\lambda$.
3. Let j be the largest integer such that $O[j]$ is a query and $\mathtt{label}(O[j]) = \mathtt{label}(o)$. If no such j exists, set j to -1. For all $m > j$, is $\mathtt{label}(o) = \mathtt{label}(O[m])$ and if $O[m]$ is an update, append the corresponding string from U to a list U'.

4. If $| U' | > 0$, set x to be a uniformly random string from $\{0,1\}^\lambda$ and program the random oracle H as follows: for all $s \in [| U' |]$, $H(x,s) := U'[s]$.
5. If $| U' | > 0$, return $((x, | U' |), \mathsf{M}[i])$. Else, return $\mathsf{M}[i]$.
6. Initialize $2 \cdot \ell$ arrays $(A_{0,i}, A_{1,i})_{i \in [1,\ell]}$ of size h each. Each entry of the array is an encryption of $\perp$.
7. Return $(A_{0,i}, A_{1,i})_{i \in [1,\ell]}$.

$(\mathtt{st}, \mathsf{Response}) \leftarrow \mathcal{S}.\mathsf{SimUpdate}(1^\lambda, \mathtt{uop}, \ell)$**:**

1. Compute an y that is an encryption of tuple $(\perp, \vec{\mathsf{v}}^\perp)$ where $\vec{\mathsf{v}}^\perp$ consists of ℓ values of $\perp$.
2. Choose x uniformly at random from $\{0,1\}^\lambda$ and append x to U.
3. $\mathcal{S}$ returns (x, y).

We now show that for all PPT adversaries $\mathcal{A}$, the probability that $\mathbf{Real}_{\mathbf{2ch_{FB}},\mathcal{A}}(1^\lambda)$ outputs 1 is negligibly different from the probability that $\mathbf{Ideal}_{\mathbf{2ch_{FB}},\mathcal{A},\mathcal{S}}(1^\lambda)$ outputs 1. To do this, we use the following sequence of games:

- $\mathbf{Game}_0$ is identical to $\mathbf{Real}_{\mathbf{2ch_{FB}},\mathcal{A}}(1^\lambda)$.
- $\mathbf{Game}_1$ replaces the PRFs F, G with a random function. This is indistinguishable from $\mathbf{Game}_0$ because of the pseudo-randomness of F, G.
- $\mathbf{Game}_2$ replaces output of H with random strings during update protocol and during search the random oracle H is programmed so that H spits out the random strings picked during update when queried.
- $\mathbf{Game}_3$ replaces the IND-CPA encryption SKE.Enc steps with simply producing a random string. RCPA security of SKE guarantees indistinguishability between a ciphertext and a randomly generated string.
- $\mathbf{Game}_4$ replaces the outputs of random functions with uniformly random chosen values. This is indistinguishable from $\mathbf{Game}_3$ as the output of random function and a random string are indistinguishable.

$\mathbf{Game}_4$ is the same as the ideal experiment completing the proof. □

Lemma 3. *Leakage function $\mathcal{L}$ is volume-hiding.*

Proof. To prove that $\mathcal{L}$ is volume-hiding, we consider any two multi-maps with the number of values $\leq n$ and with maximum volume of a label $\leq \ell$. Note that the only other leakages are the global label-equality pattern and the number of updates performed for queried labels since the last searches on them. In particular, no leakage about the value tuples associated with update operations is leaked. As a result, the input to the adversary in both volume-hiding games with different multi-maps is identical, completing the proof. □

Lemma 4. *Leakage function $\mathcal{L}$ is forward private and type-II backward private.*

Proof. Note that $\mathcal{L}_{\mathsf{Update}}$ is only dependent on the public parameter ℓ and independent of all previous operations. Therefore, $\mathcal{L}$ is forward private. For type-II backward privacy, we note that the leakage during queries on previous updates is the number of previous updates on the queried label that may be computed using $\mathsf{TimeUpdate}(O)$ where O is all previous operations. Therefore, $\mathcal{L}$ is also type-II backward private. □

Proof of Theorem 4. Follows directly from above theorem and lemmas. □

D Security Proof of $\mathbf{2ch^{s}_{FB}}$

Proof of Theorem 5. We will utilize a simulator $\mathcal{S}^{pi}$ for our initialization of each $\mathsf{EMM}^{\mathsf{loc}}_i$ using the **PiBas*** construction. We assume $\mathcal{S}'$ to be the simulator for $\mathbf{2ch_{FB}}$. We consider a stateful simulator $\mathcal{S}$ with state st that works as follows:

$\mathsf{EMM} \leftarrow \mathcal{S}.\mathsf{SimSetup}(1^\lambda, n)$**:**

1. execute $\mathcal{S}'.\mathsf{SimSetup}(1^\lambda, n)$

$\mathsf{Response} \leftarrow \mathcal{S}.\mathsf{SimQuery}(1^\lambda, \mathtt{uop}, \ell, \mathsf{leq}(O, o), \mathcal{L}_{\mathsf{loc}})$**:**

1. Using the total number of updates so far, determine the encrypted multi-maps $\mathsf{EMM}^{\mathsf{loc}}_i$ that will be non-empty.
2. For each $\mathsf{EMM}^{\mathsf{loc}}_i$ that is non-empty, execute and return $\mathcal{S}^{pi}.\mathsf{SimQuery}(1^\lambda, \mathcal{L}_{\mathsf{loc}})$.
3. Using $\mathsf{leq}(O, o)$, find smallest i such that $\mathtt{label}(o) = \mathtt{label}(O[i])$ and $O[i]$ is a query.
4. If no such i exists, set $i \leftarrow |O| + 1$ and $\mathsf{M}[i]$ to be a uniformly random string from $\{0, 1\}^\lambda$.
5. Let j be the largest integer such that $O[j]$ is a query and $\mathtt{label}(O[j]) = \mathtt{label}(o)$. If no such j exists, set j to -1. For all $m > j$, if $\mathtt{label}(o) = \mathtt{label}(O[m])$ and if $O[m]$ is an update, append the corresponding string from U to a list U'.
6. Return $(U', \mathsf{M}[i])$.
7. Initialize $2 \cdot \ell$ arrays $(A_{0,i}, A_{1,i})_{i \in [1,\ell]}$ of size h each. Each entry of the array is an encryption of $\perp$.
8. Return $(A_{0,i}, A_{1,i})_{i \in [1,\ell]}$.

$(\mathtt{st}, \mathsf{Response}) \leftarrow \mathcal{S}.\mathsf{SimUpdate}(1^\lambda, \ell, \mathtt{uop})$**:**

1. Using the total number of updates so far, determine the encrypted multi-map $\mathsf{EMM}^{\mathsf{loc}}_i$ that will be constructed and uploaded. Execute and return $\mathcal{S}^{pi}.\mathsf{SimSetup}(1^\lambda, 2^i)$.
2. Compute an y that is an encryption of tuple $(\perp, \vec{\mathtt{v}}^{\perp})$ where $\vec{\mathtt{v}}^{\perp}$ consists of ℓ values of $\perp$.
3. Choose x uniformly random from $\{0, 1\}^\lambda$ and append x to U.
4. $\mathcal{S}$ returns (x, y).

We now show that for all PPT adversaries $\mathcal{A}$, the probability that $\mathbf{Real}_{\mathbf{2ch^{s}_{FB}},\mathcal{A}}(1^\lambda)$ outputs 1 is negligibly different from the probability that $\mathbf{Ideal}_{\mathbf{2ch^{s}_{FB}},\mathcal{A},\mathcal{S}}(1^\lambda)$ outputs 1. To do this, we use the following sequence of games:

- $\mathbf{Game}_0$ is identical to $\mathbf{Real}_{\mathbf{2ch^{s}_{FB}},\mathcal{A}}(1^\lambda)$.
- $\mathbf{Game}_1$ replaces the PRFs F, G with a random function. This is indistinguishable from $\mathbf{Game}_0$ because of the pseudo-randomness of F, G.
- $\mathbf{Game}_2$ replaces the IND-CPA encryption $\mathsf{SKE.Enc}$ steps with simply producing a random string. RCPA security of SKE guarantees indistinguishability between a ciphertext and a randomly generated string.
- $\mathbf{Game}_3$ replaces the outputs of random functions with uniformly random chosen values. This is indistinguishable from $\mathbf{Game}_2$ as the output of random function and a random string are indistinguishable.

- $\mathbf{Game}_4$ replaces the search and setup algorithms of $\mathbf{PiBas}^*$ with corresponding algorithms of $\mathcal{S}^{pi}$. This is indistinguishable from $\mathbf{Game}_3$ as otherwise this would break the security of $\mathbf{PiBas}^*$.

$\mathbf{Game}_4$ is the same as the ideal experiment completing the proof. □

Lemma 5. *Leakage function $\mathcal{L}$ is volume-hiding.*

Proof. As discussed above, the only additional leakage of $\mathbf{2ch^s_{FB}}$ compared to $\mathbf{2ch_{FB}}$ is $\mathcal{L}_{\mathsf{loc}}$ when the query protocol is executed and that adds no additional information about the volume of the searched-for `label`. □

Lemma 6. *Leakage function $\mathcal{L}$ is forward private and type-II backward private.*

Proof. Note, the update leakage remains independent of all previous operations. Hence, $\mathbf{2ch^s_{FB}}$ is forward private. The query leakage is identical for both $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ except for $\mathcal{L}_{\mathsf{loc}}$. Looking closer, $\mathcal{L}_{\mathsf{loc}}$ only reveals the number of updates that occured for ℓ, which is something that is already revealed by $\mathbf{2ch_{FB}}$. Therefore $\mathcal{L}$ is type-II backward private. □

Proof of Theorem 6. Follows directly from above theorems and lemmas. □