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

Ghous Amjad
Google
New York, New York, USA
gamjad@google.com

Sarvar Patel
Google
New York, New York, USA
sarvar@google.com

Giuseppe Persiano
Università di Salerno
Salerno, Italy
giuper@gmail.com

Kevin Yeo
Google
New York, New York, USA
kwlyeo@google.com

Moti Yung
Google
New York, New York, USA
moti@google.com

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 data. We focus on *encrypted multi-maps* (EMMs), a structured encryption (STE) scheme that stores pairs of label and value tuples. EMMs allow queries on labels and return the associated value tuple. As responses are variable-length, EMMs are subject to volume leakage attacks introduced by Kellaris *et al.* [CCS'16]. To prevent these attacks, *volume-hiding* EMMs were introduced by Kamara and Moataz [Eurocrypt'19] that hide the label volumes (i.e., the value tuple lengths).

As our main contribution, we present the first *fully dynamic volume-hiding* EMMs that are both asymptotically and concretely efficient. Furthermore, they are simultaneously *forward and backward private* which are the de-facto standard security notions for dynamic STE schemes. 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'19].

KEYWORDS

Structured Encryption, Dynamic Encrypted Search, Volume-Hiding

1 INTRODUCTION

Structured encryption (STE) schemes, introduced by Chase and Kamara [10], 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 it 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 (ORAMs). 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 an STE scheme that manage collections of pairs of *labels* and *value tuples* consisting of one or more values. EMMs form the basis of many important applications where clients outsourced encrypted data to a untrusted cloud server. By leveraging EMMs, one can build systems that enable searching over the encrypted data (also known as searchable encryption [9, 11, 39]), or performing SQL queries over the encrypted databases [19]. Therefore, the construction of efficient and private EMMs is an important line of research to enable these real-world applications. We will focus on dynamic EMMs that also enable clients to update the outsourced encrypted data.

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 vague and subjective. 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 (such as [8, 14–16, 22, 28, 36, 43]) improve the accuracy of the attack or 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 have leveraged volume leakage to compromise privacy in certain settings (see [5, 17, 22, 24] and references therein). Previous works (such as [5]) have shown that volume leakage may be used to compromise even the most powerful cryptographic primitives including ORAMs. Kamara and Moataz [20] introduced the notion of *volume-hiding* EMMs to protect against these attacks. 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.* [35] that presented asymptotically optimal schemes for the *static* setting. In a concurrent work, Wang and Chow [42] presented another

Proceedings on Privacy Enhancing Technologies 2023(1), 417–436
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2023-0025

static volume-hiding EMM that slightly reduced server storage by increasing query computation.

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 multi-map (MM) 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 scheme offering a subset of natural update operations (see Appendix A for more details). Furthermore, these schemes are less efficient than the static volume-hiding EMMs presented by Patel *et al.* [35]. In our work, we fill this gap by presenting dynamic EMMs offering natural update operations while being as efficient as their static counterparts.

Dynamic EMMs. Before presenting our dynamic volume-hiding EMMs schemes, 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 is outsourced to a potentially untrusted cloud server (outsourcing is typically done for many reasons including fault tolerance and/or availability). In many such applications, updating this data regularly is necessary.

Classic examples of highly sensitive data are financial or medical information. For example, a firm may wish to keep track of the transactions that occurred at each hour during a work day by uploading its financial transactions to an outsourced EMM hourly meaning the EMM is updated every hour. It can be immediately seen that volume-hiding is important in this setting as the volume directly reveals the number of transactions by the hour. A similar situation can occur for hospitals that record their medical inventory usage at the end of each day. The volume is correlated to the number of patients that were treated. Both cases are examples where volume-hiding is integral as volumes would reveal information that is not exposed elsewhere (for example, the patterns of updates do not reveal anything as they occur at fixed, regular time intervals). Moreover, even with volume-hiding during updates, (changes in) volume during queries can essentially reveal the same information.

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 standard privacy requirements in dynamic settings to avoid privacy degradation. Formally, these standard notions for dynamic STE schemes are *forward* and *backward* privacy [6, 7, 40]. Forward privacy guarantees that insertion operations do not leak information on previous queries. Backward privacy addresses a similar concern with respect to deletion ensuring 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, schemes 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 combining leakage between query and update operations does not reveal volumes as well.

In our work, we will design dynamic volume-hiding encrypted multi-maps that provide forward and backward privacy while simultaneously being efficient.

1.1 Our Contributions

As our main contribution, we present *dynamic volume-hiding* EMMs that are *forward and backward private* with better efficiency than 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* (DST). For a MM with n total values and maximum volume ℓ, DST requires $O(\ell \log n)$ overhead for both queries and updates. Furthermore, DST supports only a subset of update operations and is not forward private, a standard security notion of the dynamic setting. With this in mind, there are four main challenges that we address in our work.

(1) **Dynamicity and Hiding Volume.** The volume-hiding scheme 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 from an existing value tuple. While one can achieve this functionality using a query before an update, it turns out that this degrades privacy significantly (see Appendix A). This motivates the following question: *Is it possible to construct fully-dynamic volume-hiding schemes 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 [6, 7] 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 [43]. At a high level, these notions guarantee that modified data is not leaked until a query for the data is performed. Prior volume-hiding schemes [20, 35] are not forward and backward private, which motivates the following problem: *Is it possible to construct volume-hiding dynamic* EMM*s that are both forward and backward private?*

(3) **Efficiency.** DST [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 scheme [35] 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?*

(4) **Leakage.** Beyond forward and backward privacy, our schemes will aim to leak as little information as possible. We identify

three leakages that are necessary for functionality or efficiency: MM size n, maximum volume ℓ and label equality leakage (whether two operations are for the same label). The MM size n is necessary as the server stores the EMM. We show that hiding the maximum volume ℓ would require $\Omega(n)$ communication for any reasonable error probability in Appendix B. Patel *et al.* [34] showed that avoiding label equality leakage would require overhead equivalent to ORAMs [3, 13, 31] (see Appendix J for more details). It is not a coincidence that prior works [20, 35, 42] leaked all three of n, ℓ and label equality. This leads to the natural question: *Is it possible to construct dynamic volume-hiding schemes supporting the above properties with minimal leakage?*

We present two schemes $\mathbf{2ch_{FB}}$ 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 that ℓ is the maximum length of any value tuple and n is the maximum total number of values.

Theorem 1 (Informal). *There exists a fully-dynamic, volume-hiding, forward and type-II backward private* EMM, $\mathbf{2ch_{FB}}$ *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 in the* MM.

$\mathbf{2ch_{FB}}$ achieves all our goals of dynamicity, volume-hiding, efficiency, forward/backward privacy and minimal leakage of only n, ℓ and label equality. However, $\mathbf{2ch_{FB}}$ requires $O(m)$ client storage, which is common to the majority of forward private schemes, such as schemes in [6, 7]. We present $\mathbf{2ch^{s}_{FB}}$ with smaller permanent client storage at the cost of slightly larger overhead.

Theorem 2 (Informal). *There exists a fully-dynamic, volume-hiding, forward and type-II backward private* EMM $\mathbf{2ch^{s}_{FB}}$ *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* $f(n) = \omega(\log n)$.

To our knowledge, $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ are the first *dynamic* EMM schemes that *simultaneously* provide *volume-hiding*, *forward* and *backward privacy* while being *concretely efficient* with a small number of roundtrips with minimal leakage of n, ℓ and label equality. A comparison of the asymptotic performance of our schemes and prior dynamic schemes obtaining at least one of volume-hiding, forward or backward privacy are presented in Table 1. The experimental evaluation in Section 5.1 shows that our schemes also improve on the concrete performance of prior schemes. It also shows that we enable dynamicity without incurring any additional cost when compared with prior static schemes [35]. This is very surprising as static schemes are optimized for query communication whereas experimentally our schemes, despite having to support a very rich set of dynamic operations, have query cost comparable with the static scheme of [35]. In addition, our schemes exhibit a 2-3x improvement in query communication cost over DST [20], the best existing non-lossy volume hiding dynamic scheme, while supporting a wider range of dynamic operations and providing stronger security guarantees.

Discussion about Concurrent Works. Zhao et al. [44] give two schemes for volume-hiding dynamic EMMs. Even though their schemes offer stronger type-I backward privacy and avoid leaking ℓ (along with similar security guarantees elsewhere), they fail to return the correct value tuple for all instances. In particular, only an ℓ/n-fraction is guaranteed to be returned. However, their schemes obtain smaller query computation and communication of $O(\ell + \log n)$ compared to our schemes.

Wang and Chow [42] also construct dynamic volume-hiding EMMs with the same privacy guarantees as ours along with very small server storage overhead by using consistent hashing. To guarantee forward and backward privacy, they cache the update operations and then execute them as part of the first available query operation. A query then requires time proportional to the number of cached updates as it needs to handle them. To improve query performance, they allow updates to be processed in batches. A set of updates make a batch, if they arrive simultaneously. Each batch update is stored in a separate volume-hiding EMM. A query consists of querying all uploaded EMMs increasing query overhead for each set of batch updates. Thus, the ability of handling updates in batches does not improve the worst-case running time, unless the client is willing to accumulate the updates in local memory to form a batch. Our schemes also employ caching of the update operations (that is, the updates are not instantly implemented on the main data structure). However, this is done while guaranteeing that the worst case query overhead is independent of the number of updates. We note that when the number of batched updates is small such as $b = O(1)$, the scheme in [42] has smaller query communication $O(b\ell)$ but either larger query computation $O(b\ell \log n)$[1] or client storage $O(m)$ compared to $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ respectively.

Discussion about Backward Privacy. Both of $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ provide type-II backward privacy as defined in [7]. We note that there are several schemes that provide stronger type-I backward privacy. However, current type-I backward private schemes are expensive and resort to usage of ORAMs. 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, ORAMs [3, 13, 31] address the problems of dynamicity, forward and backward privacy (as outlined in [20]). However, ORAMs are expensive as they require logarithmic number of client-server roundtrips or fully homomorphic encryption schemes. As evidenced by prior works such as [37], the high number of roundtrips of ORAMs significantly hinder efficiency. In our work, we ensure our schemes use either 1 or 2 roundtrips and only use cheap, symmetric primitives instead of expensive cryptographic tools such as ORAMs and FHE. DST [20] has the same asymptotic overhead as an ORAM, but is faster in practice due to 1 roundtrip and no FHE usage.

Discussion about Parallelism. Prior works [21, 26, 40] investigated enabling the client to issue multiple operations in parallel. In our work, we will focus on constructing dynamic volume-hiding schemes in the sequential setting that were previously not known to exist with our efficiency and privacy guarantees. To our knowledge, we believe that DST [20] and DSSE [44] may enable issuing parallel queries (but could not verify this). We believe all other volume-hiding works (including [42] and ours) do not have this

[1]Theoretically, the authors in [42] mention the usage of more complex predecessor data structures can reduce this to $O(b\ell \log\log n)$.

	Query Comm.	Query Comp.	Query RT	Update Comm.	Update Comp.	Client Storage	VH	FP	BP	Correct %-age	Leakage
Sofos [6]	$O(\ell_{\texttt{label}})$	$O(\ell_{\texttt{label}})$	1	$O(u_{\texttt{label}})$	$O(u_{\texttt{label}})$	$O(m)$	×	✓	×	1	$(n, \ell_{\texttt{label}}, \mathsf{leq})$
Fides [7]	$O(\ell_{\texttt{label}})$	$O(\ell_{\texttt{label}})$	2	$O(u_{\texttt{label}})$	$O(u_{\texttt{label}})$	$O(m)$	×	✓	II	1	$(n, \ell_{\texttt{label}}, \mathsf{leq})$
SD$_a$ [12]	$O(\ell_{\texttt{label}})$	$O(\ell_{\texttt{label}})$	1	$O(u_{\texttt{label}} \log n)$	$O(u_{\texttt{label}} \log n)$	$O(1)^*$	×	✓	II	1	$(n, \ell_{\texttt{label}}, \mathsf{leq})$
DST [20]	$O(\ell \log n)$	$O(\ell \log n)$	1	$O(\ell \log n)$	$O(\ell \log n)$	$O(1)$	✓	×	II	1	(n, ℓ, leq)
$\mathcal{S}^4$ [42]	$O(b\ell)$	$O(b\ell \log n)$	1	$O(\ell)$	$O(\ell)$	$O(m)$	✓	✓	II	1	(n, ℓ, leq)
DSSE [44]	$O(\ell + \log n)$	$O(\ell + \log n)$	1	$O(\ell \log n)$	$O(\ell \log n)$	$O(1)$	✓	✓	I	ℓ/n	(n, leq)
DSSE$^{\mathsf{k}}$ [44]	$O_\lambda(\ell + \log n)$	$O_\lambda(\ell + \log n)$	1	$O_\lambda(\ell \log n)$	$O_\lambda(\ell \log n)$	$O(1)$	✓	✓	I	ℓ/n	(n, leq)
2ch	$O(\ell \log\log n)$	$O(\ell \log\log n)$	1	$O(\ell \log\log n)$	$O(\ell \log\log n)$	$\omega(\log n)$	✓	×	II	1	(n, ℓ, leq)
2ch$_{\mathbf{FB}}$	$O(\ell \log\log n)$	$O(\ell \log\log n)$	1	$O(\ell)$	$O(\ell)$	$O(m)$	✓	✓	II	1	(n, ℓ, leq)
2ch$^{\mathbf{s}}_{\mathbf{FB}}$	$O(\ell \log\log n)$	$O(\ell \log\log n)$	2	$O(\ell \log n)$	$O(\ell \log n)$	$\omega(\log n)^*$	✓	✓	II	1	(n, ℓ, leq)

Table 1: A comparison of amortized query and update overhead of dynamic schemes that provide either volume-hiding, forward or backward privacy with our schemes. We use the following abbreviations: roundtrips (RT), volume-hiding (VH), forward privacy (FP) and backward privacy (BP). For notation, n denotes the maximum number of values in the multi-map MM and m denotes the number of unique labels in the MM. For volume-hiding schemes, ℓ represents the maximum volume. We denote $\ell_{\texttt{label}}$ to be the volume associated with the queried label and $u_{\texttt{label}}$ to be the number of updated values. Correct %-age refers to the percentage of returned correct values. For client storage, an asterisk* means client storage may increase up to $O(n)$ temporarily. For [42], b refers to the number of batch updates. In [44], $O_\lambda(x)$ means there are hidden λ factors. Label equality leakage is referred to by leq.

property. We leave it as an open problem to enable parallel queries in our schemes.

2 DEFINITIONS

2.1 Structured Encryption

In a 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 [10]. While we consider generic definitions for encrypting any data structure, our work focuses on MMs 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 protocols to be executed between 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 an STE scheme is r-interactive if it can be completed in at most r rounds of communication between the client and the server. An 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.

Definition 1. *An r-interactive dynamic STE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ consists of the following protocols between client $\mathbb{C}$ and 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 $\mathbb{C}$ and $\mathbb{S}$ where $\mathbb{C}$ receives* $(1^\lambda, \mathsf{params}, \mathsf{DS})$ *and $\mathbb{S}$ receives 1^λ. At termination, $\mathbb{C}$ receives its* state st *and $\mathbb{S}$ receives 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 $\mathbb{C}$ and $\mathbb{S}$ where $\mathbb{C}$ receives* $(\mathsf{st}, \mathsf{qop})$ *and $\mathbb{S}$ receives* EDS. *For each $i \in \{0, \ldots, r-1\}$, $\mathbb{C}$ generates the i-th message using the state* st*, the query operation* qop *and all previous messages. $\mathbb{S}$ generates the i-th message using the encrypted data structure* EDS *and all previous messages. At termination, $\mathbb{C}$ receives the query result* 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 $\mathbb{S}$ where $\mathbb{C}$ receives* $(\mathsf{st}, \mathsf{up})$ *and $\mathbb{S}$ receives* EDS. *For each $i \in \{0, \ldots, r-1\}$, $\mathbb{C}$ generates the i-th message using the state* st*, the update operation* up *and all previous messages. $\mathbb{S}$ generates the i-th message using the encrypted data structure* EDS *and all previous messages. 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 setup reveals no information beyond $\mathcal{L}_{\mathsf{Setup}}$; (2) a query reveals no information beyond $\mathcal{L}_{\mathsf{Query}}$; and (3) an update operation reveals no information beyond $\mathcal{L}_{\mathsf{Update}}$. The leakage on an operation may depend on all the previous operations and the setup phase.

We consider *adaptive* security that considers adversaries that view the execution of one operation before choosing the next operation that was first formalized by Curtmola et al. [11]. The definition utilizes 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 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 $\mathcal{A}(1^\lambda, z)$, takes as input the *security parameter* 1^λ and the *auxiliary information* z, outputs an input data structure DS. The challenger C executes Setup on DS obtaining client state st and encrypted data structure EDS. 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 C jointly 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 C jointly 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 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 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 C runs the simulator $\mathcal{S}$ on input leakage $\mathcal{L}_{\mathsf{Setup}}(\mathsf{DS}, n, \ell)$ 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 $\mathcal{S}$ jointly execute protocol Query on input $\mathcal{L}_{\mathsf{Query}}(\mathsf{DS}, o_1, \ldots, o_i)$.
- If o_i is an update operation, then $\mathcal{A}$ and $\mathcal{S}$ jointly execute protocol Update on input $\mathcal{L}_{\mathsf{Update}}(\mathsf{DS}, o_1, \ldots, o_i)$.
- 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 protocol transcript and an updated version of EDS. Note, $\mathcal{S}$ may deviate from the protocol.

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

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

(1) $\mathsf{Response} \leftarrow \mathsf{Query}(\mathtt{label}, \mathsf{MM})$. The *query operation* retrieves the tuple $\mathsf{MM}[\mathtt{label}]$.
(2) $\mathsf{MM}^{\mathsf{new}} \leftarrow \mathsf{Update}(\mathsf{op} \in \{\mathsf{edit}, \mathsf{app}, \mathsf{rm}, \mathsf{del}\}, \mathtt{label}, \vec{\mathsf{v}}, \mathsf{MM}, n, \ell)$. The following types of update operations are supported:
   (a) If $\mathsf{op} = \mathsf{edit}$, then the *label edit* operation sets the entry $\mathsf{MM}[\mathtt{label}] \leftarrow \vec{\mathsf{v}}$.
   (b) If $\mathsf{op} = \mathsf{rm}$, then the *label removal* operation removes the pair $(\mathtt{label}, \mathsf{MM}[\mathtt{label}])$ from MM. If $\mathtt{label} \notin \mathsf{LABEL}(\mathsf{MM})$, the label removal operation has no effect. The input $\vec{\mathsf{v}}$ is ignored.
   (c) If $\mathsf{op} = \mathsf{app}$, then the *label value append* operation sets $\mathsf{MM}[\mathtt{label}] \leftarrow (\mathsf{MM}[\mathtt{label}], \vec{\mathsf{v}})$; to append tuple $\vec{\mathsf{v}}$ to the current tuple.
   (d) If $\mathsf{op} = \mathsf{del}$, then the *label value deletion* operation. sets $\mathsf{MM}[\mathtt{label}] \leftarrow \mathsf{MM}[\mathtt{label}] \setminus \vec{\mathsf{v}}$. That is, all values in $\vec{\mathsf{v}}$ are removed from $\mathsf{MM}[\mathtt{label}]$.

For convenience, we represent MM operations as the tuple $o = (\mathsf{op}, \mathtt{label}, \vec{\mathsf{v}})$ where op is the operation type, $\mathtt{label}$ is the input label and $\vec{\mathsf{v}}$ is the input value tuple. We use $\mathsf{op}(o) \in \{\mathsf{qop}, \mathsf{edit}, \mathsf{rm}, \mathsf{app}, \mathsf{del}\}$ to denote the type of an operation o, $\mathtt{label}(o)$ to be the label of o and $\vec{\mathsf{v}}(o)$ to be the value tuple of o.

If the size n or maximum volume ℓ may change, the new values must be submitted as parameters to Update.

2.4 Label Equality Leakage

The label equality pattern leaks 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 $\mathtt{label}(o_i) = \mathtt{label}(o_j)$.

We note that [34] proved a lower bound showing mitigating label equality leakage in any small way would require $\Omega(\ell \log n)$ computational overhead. Our schemes will all leak label equality to obtain better efficiency.

2.5 Volume Hiding Leakage Functions

Volume-hiding leakage functions were introduced in [20] and formally defined in [35] for static schemes. We present a definition of a *volume-hiding leakage function* for dynamic EMMs that extends the definition of Patel *et al.* [35] using game-based definitions.

For a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}})$ and adversary $\mathcal{A}$, we consider games $\mathbf{VHGame}^{\mathcal{A},\mathcal{L}}_{\eta}$ with $\eta \in \{0, 1\}$. The adversary $\mathcal{A}$ selects two MMs 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}. After each operation, $\mathcal{A}$ must report a valid upper bound of the size and maximum volume for both MM_0 and MM_1.

$\mathbf{VHGame}^{\mathcal{A},\mathcal{L}}_{\eta}$:

(1) $\mathcal{A}$ picks size and volume upper bounds n and ℓ and sends two multi-maps MM^0_0 and MM^0_1 to C satisfying n and ℓ.
(2) C computes $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}^0_{\eta}, n, \ell)$ which is sent to $\mathcal{A}$.
(3) For $t = 1, \ldots,$
   (a) $\mathcal{A}$ adaptively picks $o^t_0 = (\mathsf{op}^t_0, \mathtt{label}^t_0, \vec{\mathsf{v}}^t_0, n^t, \ell^t)$ and $o^t_1 = (\mathsf{op}^t_1, \mathtt{label}^t_1, \vec{\mathsf{v}}^t_1, n^t, \ell^t)$ such that
      (i) The operation types and label equality leakage are the same: $\mathsf{op}^t_0 = \mathsf{op}^t_1$ and $\mathsf{leq}(o^1_0, \ldots, o^t_0) = \mathsf{leq}(o^1_1, \ldots, o^t_1)$.
      (ii) Let MM^t_0 (MM^t_1) be the MM obtained by executing o^t_0 (o^t_1) on MM^{t-1}_0 (MM^{t-1}_1). n^t and ℓ^t must be valid size and volume upper bounds for MM^t_0 and MM^t_1.

(b) C returns $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}^t_\eta, o^1_\eta, \ldots, o^t_\eta)$ for queries and $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}^t_\eta, o^1_\eta, \ldots, o^t_\eta)$ for updates.

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

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

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 $n \geq \ell \geq 1$,*

$$p^{\mathcal{A},\mathcal{L}}_0(n,\ell) = p^{\mathcal{A},\mathcal{L}}_1(n,\ell).$$

Definition 4 (Volume-Hiding Encrypted Multi-Maps). *An* EMM *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 according to Definition 3.

We note this definitions reflects that both the MM size n and maximum volume ℓ will grow over time as more operations occur. An upper bound on n and ℓ will be inherently leaked after each operation. In a concurrent work [44], an alternative definition is provided where ℓ is not leaked. In Appendix B, we show such a definition inherently requires large query communication. If we want to even guarantee that ϵ-fraction of matching values are returned, we show a query communication lower bound of $\Omega(\epsilon n)$. In other words, if we want at least half the matching values, then the query communication is already linear. As a result, we choose to use a definition that leaks ℓ to ensure better efficiency and correctness. The construction in [44] adheres to our lower bound as they can only return (ℓ/n)-fraction of matching values when using $O(\ell)$ query communication.

Discussion about Label Equality Leakage. In our volume hiding definition, the adversary must choose two sequences with the same label equality leakage. Instead, we could have chosen a more general definition by parameterizing the game with some leakage function $\mathcal{L}_{\mathtt{label,op}}$ over the labels and operations and force the adversary to submit two sequences with the same leakage with respect to $\mathcal{L}_{\mathtt{label,op}}$. We chose label equality as prior works [34] showed mitigating label equality requires large overhead similar to ORAMs (see Appendix J for more details). On the other hand, leaking only label equality is sufficient for faster constructions. Therefore, label equality seems to be the minimal leakage required to obtain efficient constructions faster than ORAMs.

2.6 Forward and Backward Privacy

Forward and backward privacy provide guarantees on the amount of information leaked to an adversary as the client performs update operations. We present the standard definitions of forward and backward privacy (readers may also refer to [6, 7]).

Forward privacy guarantees that the leakage of update operations is independent of all previous operations. For any forward private leakage, an update o does not give any information on the sequence of operations O except the update operation itself.

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* MM*, sequence O and update operation o, $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = \mathcal{L}'_{\mathsf{Update}}(\mathsf{op}(o), \vec{\mathsf{v}}(o))$.*

Backward privacy controls the leakage viewed by the adversary during queries about previous deletion operations. In our work, we obtain type-II backward privacy where only the total number of updates and their timestamps are revealed for deleted items.

Bost *et al.* [7] formally defined three types of backward privacy where type-I provides the strongest privacy to type-III with the weakest privacy. To define backward privacy, we need the following three additional leakage functions that takes as argument a sequence O of operations that is omitted for convenience.

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

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

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

that consists of the timestamps of all update operations that modify $\mathtt{label}$. Finally,

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

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

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

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

We note that Type-II backward privacy reveals the total number of updates performed on label and the timestamps of each update operation for label. All our constructions will be type-II backward private. We point readers to Appendix C for definitions of other types of backward privacy.

2.7 Cryptographic Tools

We will utlize pseudorandom functions (PRFs) and IND-CPA encryption. PRF F guarantees its output is computationally indistinguishable from random functions for a secret seed. In our proofs, we may model them as random oracles. IND-CPA encryption scheme SKE = (Gen, Enc, Dec) ensures each ciphertext is computationally indistinguishable from random strings.

3 OUR CONSTRUCTIONS

In this section, we present our new constructions. We start with 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 privacy with different efficiency trade-offs. Throughout this section, we focus on the simpler setting where the upper bounds on MM size n and volume ℓ are fixed through all operations. In Section 4, we present generic transformations to handle changing n and ℓ.

3.1 Problems with Naive Padding

We start by discussing a naive solution of adding padding to prior dynamic constructions (such as [6, 7]) to obtain volume-hiding. At a high level, one could pad the storage and communication with sufficient dummy values to always return ℓ values. Unfortunately, this straightforward approach incurs $O(n\ell)$ blowup in server storage, which can be very large for many values of ℓ (such as $\ell = \sqrt{n}$). Instead, we will utilize hashing (like prior works [20, 35, 42]) to enable re-using encryptions of real values as padding for multiple queries and avoid significant storage increase. As a result, all our constructions will require the minimal $O(n)$ storage.

3.2 2ch: Warm-Up Scheme

We start from the optimal static volume-hiding scheme by Patel *et al.* [35]. Their construction utilizes cuckoo hashing [23, 30] to embed data into server storage. Cuckoo hashing guarantees that each value is stored in one of two hash table locations or in a small client stash. To perform a query, one can simply access 2ℓ hash table locations, two for each of the ℓ possible values associated with the label. 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, the algorithm displaces one of the values; the displaced value is inserted by using the same algorithm. This algorithm is not volume-hiding as the adversary learns whether certain entries are occupied or not by viewing how long the insertion algorithm runs.

Looking closer, the query algorithm with cuckoo hashing [35] 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 [27]). 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 as employed by DST [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.* [4]. 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 by Patel *et al.* [33] that we denote by H_{2ch}. This hashing scheme reduces the amount of unused space by arranging bins to share physical memory. At a high level, the hash table consists of $n/\log n$ binary trees each of height $\log\log n$ such that there are n leaf nodes. All node store at most one value. As a result, the total size becomes at most $2n$. 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 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. We formally present bounds on the stash size and point readers to the proof in [33].

Theorem 3 ([33]). *Let $f(n) = \omega(\log n)$. When mapping at most n items using H_{2ch}, the stash stores at most $f(n)$ items except with probability* $\mathtt{negl}(n)$.

Using the H_{2ch} hashing scheme and padding empty binary nodes with dummy values, 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, 35] that maps values to bins using pseudorandom functions. We note that **2ch** already results in a more efficient, volume-hiding construction than DST [20].

However, **2ch** does not achieve forward privacy as updating a label enables association with previous queries to the same label. Our next constructions will solve this problem to obtain forward privacy. In contrast, **2ch** is already type-II backward private as updates only reveal the timestamp of previous queries and updates for the same label (encapsulated by TimeDB and TimeUpdate respectively in Definition 6).

We present the pseudocode for **2ch** in Appendix D along with a formal proof of security and efficiency.

Comparison with [33]. Both [33] and our work aim to build privacy-preserving maps. However, [33] aims to hide access patterns using dummy queries for maps that store at most one value per label. In contrast, our work aims to mitigate volume leakage for MMs storing multiple values per label.

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

We formally present our dynamic volume-hiding STE scheme for MMs, $\mathbf{2ch_{FB}}$ (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 particular, we need to hide label equality leakage during updates. In **2ch** and [20], identical bins are retrieved for both queries and updates. Otherwise, an adversary can link that both operations were performed on the same label, which is why prior constructions (as well as **2ch**) are 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 MM inspired by prior works such as [6, 7, 26] to obtain forward privacy. 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 will accumulate update operations for labels that have not yet been queried. A table of PRF keys used to generate locations for storing updates in EMM_u is stored locally by the client. Once a query for `label` is performed, all update operations pertaining to `label` will be retrieved from EMM_u and applied to Table before returning the final result. As a result, all the accumulated updates for `label` in EMM_u cannot be linked until a query for `label` is performed ensuring $\mathbf{2ch_{FB}}$ achieves forward privacy. We note these ideas have been abstracted in [26]. $\mathbf{2ch_{FB}}$ is also type-II backward private inheriting the same properties as **2ch**.

We present the pseudocode of $\mathbf{2ch_{FB}}$ in Figure 1.

Setup. The setup algorithm is executed by the client $\mathbb{C}$ to construct an EMM. It takes as input a security parameter 1^λ, 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 of either bin with the highest level. If both bins are full, the tuple $(\mathtt{label}, j, \vec{v}[j])$ is stored in the stash. Remaining 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 (which is meant to store future updates temporarily). The client maintains storage of the stash, an initially empty multi-map $\mathsf{MM}_{\mathsf{st}}$ and seed K and key K_u for accessing EMM_u.

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

Query. To query for `label`, the client computes seed $F_K(\mathtt{label})$ and checks if $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}]$ exists and $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] > 0$. If so, the client computes another seed $x := H(K_u, \mathtt{label} \mid\mid \mathsf{MM}_{\mathsf{st}}[0])$ and sends seeds $F_K(\mathtt{label})$ and x to the server. Otherwise, when $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}]$ is not defined or $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] = 0$, the client only sends $F_K(\mathtt{label})$. The server expands the seed $F_K(\mathtt{label})$ to find the 2ℓ bins $\{G_{F_K(\mathtt{label})}(i \mid\mid b)\}_{i \in [\ell], b \in \{0,1\}}$ in the binary trees. Afterwards, the server returns the encryptions stored at $\{\mathsf{EMM}_u[H(x, i)]\}_{i \in [\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1]]}$ 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 `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 `label` in these 2ℓ bins and the overflow stash are finally returned as the query's answer. The client increments the label version $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][0] := \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][0] + 1$ in order to ensure forward privacy for future updates as the server now knows the current seed for `label` and EMM_u. The client also resets the count $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] := 0$ as there are no unapplied updates for `label` in EMM_u.

3.3.1 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 $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{uop})$ where 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. Type-II backward privacy is inherited as $\mathbf{2ch_{FB}}$ has essentially the same properties as **2ch** for deletions. 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. The proof that $\mathbf{2ch_{FB}}$ is $\mathcal{L}$-secure for the leakage function $\mathcal{L}$ described above is in Appendix E.

THEOREM 4. *If* SKE *is IND-CPA secure, F and G are PRFs, and 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,* EMM *scheme.*

3.3.2 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

```
Let $F$, $G$, $H$ be PRFs and SKE = (Gen, Enc, Dec) be an IND-CPA encryption scheme.
$(\mathsf{st}; \mathsf{EMM}) \leftarrow \mathbf{2ch_{FB}}.\mathsf{Setup}(1^\lambda, \mathsf{params} = (n, \ell, c), \mathsf{MM} = \{(\mathtt{label}_i, \vec{\mathsf{v}}_i)\}_{i \in [|\mathtt{label} \in \mathsf{MM}|]})$:
  (1) $\mathbb{C}$ randomly selects a PRF seeds $K, K_u \leftarrow \{0,1\}^\lambda$ 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 MMs: $\mathsf{EMM}_u$, $\mathsf{MM}_{\mathsf{st}}$.
  (4) For each $\mathtt{label}_i \in \mathsf{MM}$:
    (a) Compute $\kappa \leftarrow F_K(\mathtt{label}_i)$ and for each $j \in [|\vec{\mathsf{v}}_i|]$:
      (i) $\mathbb{C}$ computes $b_0 \leftarrow G_\kappa(j \,||\, 0)$ and $b_1 \leftarrow G_\kappa(j \,||\, 1)$ 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 $\mathsf{st} \leftarrow (K, K_u, K_{\mathsf{Enc}}, \mathtt{Stash}, \mathsf{MM}_{\mathsf{st}})$ and $\mathbb{S}$ stores $\mathsf{EMM} \leftarrow (B_1, \ldots, B_s, \mathsf{EMM}_u)$.
$(\mathsf{st}'; \mathsf{EMM}') \leftarrow \mathbf{2ch_{FB}}.\mathsf{Update}\left(((\mathsf{op}, \mathtt{label}, \vec{\mathsf{v}}), \mathsf{st}), \mathsf{EMM}\right)$:
  (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 $\vec{\mathsf{v}}$ up to $\ell$ values with $\perp$ and computes $z \leftarrow \mathsf{Enc}(K_{\mathsf{Enc}}, (\mathsf{op}, \vec{\mathsf{v}}))$.
  (4) $\mathbb{C}$ sends $(y, z)$ to $\mathbb{S}$ who updates $\mathsf{EMM}_u$ by setting $\mathsf{EMM}_u[y] \leftarrow z$.
  (5) $\mathbb{C}$ updates $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] \leftarrow \mathsf{MM}_{\mathsf{st}}[\mathtt{label}][1] + 1$.
$((\mathsf{st}', \vec{\mathsf{v}}); \mathsf{EMM}') \leftarrow \mathbf{2ch_{FB}}.\mathsf{Query}\left(((\mathsf{qop}, \mathtt{label}), \mathsf{st}), \mathsf{EMM}\right)$:
  (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 $2\ell$ 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 $2\ell$ bins and all cached update operations for $\mathtt{label}$.
  (5) $\mathbb{C}$ locally compiles $\vec{\mathsf{v}}$ containing all values tagged with $\mathtt{label}$ and deletes them from the downloaded bins and $\mathtt{Stash}$.
  (6) For each $i = 0, \ldots, \mathsf{cnt} - 1$:
    (a) $\mathbb{C}$ computes $(\mathsf{op}_i, \vec{\mathsf{v}}_i) \leftarrow \mathsf{Dec}(K_{\mathsf{Enc}}, \mathsf{EMM}_u[H(x, i)])$.
    (b) If $\mathsf{op}_i = \mathsf{app}$, append $\vec{\mathsf{v}}_i$ to $\vec{\mathsf{v}}$. If $\mathsf{op}_i = \mathsf{edit}$, set $\vec{\mathsf{v}} \leftarrow \vec{\mathsf{v}}_i$. If $\mathsf{op}_i = \mathsf{rm}$, set $\vec{\mathsf{v}} \leftarrow \perp$. If $\mathsf{op}_i = \mathsf{del}$, remove any values in $\vec{\mathsf{v}}_i$ from $\vec{\mathsf{v}}$. Afterwards, $\mathbb{C}$ compacts results so that all non-dummies appear before dummies.
  (7) $\mathbb{C}$ adds back $\vec{\mathsf{v}}$ to the $2\ell$ 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 $\vec{\mathsf{v}}$.
```

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

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`, we denote the number of unapplied update operations since the last query for `label` by $u(\mathtt{label})$. Then, the worst case overhead of a query operation is $O(\ell \log \log n + \ell u(\mathtt{label}))$ 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 Appendix G 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 $\mathsf{MM}_{\mathsf{st}}$ 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 using $f(n)$ storage except with probability negligible in n for any $f(n) = \omega(\log n)$.

Discussion about Forward Privacy. In $\mathbf{2ch_{FB}}$, the client storage increases as there are more update operations without intermediate query operations. This directly maps to the setting that forward privacy becomes more important as more information in the updates are protected from the adversarial server. In other words, the additional client storage is a direct result of providing stronger protection for updates without intermediate queries. In the next section, we present a construction providing the same forward privacy protection without the increasing client storage.

3.4 Construction $\mathbf{2ch^s_{FB}}$

Next, we present our final scheme $\mathbf{2ch^s_{FB}}$ (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 $\mathbf{2ch_{FB}}$. $\mathbf{2ch^s_{FB}}$ 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 $O(m)$. $\mathbf{2ch^s_{FB}}$ will only require permanent client storage of size $\omega(\log n)$. Recall that $\mathbf{2ch_{FB}}$ required the client to locally store $\mathsf{MM}_{\mathsf{st}}$. For any $\mathtt{label} \in \mathbb{L}$, $\mathsf{MM}_{\mathsf{st}}[\mathtt{label}]$ 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 $\mathsf{MM}_{\mathsf{st}}$ to the server inspired by ideas from recent work in ORAMs [25, 31] and encrypted search [12].

In order to get rid of $\mathsf{MM}_{\mathsf{st}}$ at the client, $\mathbf{2ch^s_{FB}}$ will explicitly store the location of cached operations in EMM_u, in a series of static, encrypted multi-maps $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{t-1}^{\mathsf{loc}}$ 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, $\mathsf{EMM}_i^{\mathsf{loc}}$ stores at most 2^i cached update operations. We instantiate these t structures using **PiBas*** (a modified version of **PiBas** [9]) that is a static, response-hiding, volume-revealing, encrypted multi-map scheme as described in [12]. 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 **PiBas***. Specifically, $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ maintains the invariant that the encrypted multi-map $\mathsf{EMM}_i^{\mathsf{loc}}$ will store the locations of cached operations per label over the latest update operations that are not stored in smaller encrypted multi-maps, $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$. 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 $\mathsf{EMM}_0^{\mathsf{loc}}, \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}$ are fully occupied. For the next update operation, the contents will be combined and placed into the larger encrypted multi-map $\mathsf{EMM}_i^{\mathsf{loc}}$.

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

We present the pseudocode for $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ in Figure 2.

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

Update. For an update operation $(\mathsf{op}, \mathtt{label}, \vec{\mathsf{v}})$, the client chooses a random location x and stores an encryption of the current update operation at $\mathsf{EMM}_u[x]$ after padding $\vec{\mathsf{v}}$ 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 $\mathtt{label}'$ that appears in one of the i downloaded encrypted multi-maps, the client sets $\mathsf{MM}_i[\mathtt{label}'] = (\mathsf{EMM}_0^{\mathsf{loc}}[\mathtt{label}'], \ldots, \mathsf{EMM}_{i-1}^{\mathsf{loc}}[\mathtt{label}'])$. The random location of the current update x is also appended to $\mathsf{MM}_i[\mathtt{label}]$. 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 $\mathtt{label} \in \mathbb{L}$, $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ performs t queries with the server $\mathbb{S}$ to retrieve the locations of all cached update operations for $\mathtt{label}$ in EMM_u. Afterwards, $\mathbb{C}$ uses the same algorithm as $\mathbf{2ch}_{\mathbf{FB}}$.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.

3.4.1 Security. We present the leakage profile for $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ 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 $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ 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 EMMs 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 $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{uop})$ which is also same as $\mathbf{2ch}_{\mathbf{FB}}$. Finally, the query leakage of $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ is similar to $\mathbf{2ch}_{\mathbf{FB}}$ but it also has an extra leakage of queries on $\mathsf{EMM}_i^{\mathsf{loc}}$ that we denote as $\mathcal{L}_{\mathsf{loc}}$. However, $\mathcal{L}_{\mathsf{loc}}$ is a strict subset of the label-equality leakage. Therefore, $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{leq}(O, o), \mathsf{qop}, \mathcal{L}_{\mathsf{loc}})$. As this is essentially the same leakage as $\mathbf{2ch}_{\mathbf{FB}}$, $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ also inherits forward and type-II backward privacy. The security proof of $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ is found in Appendix F.

THEOREM 5. *If* SKE *is IND-CPA secure and* **PiBas*** *is a static, response-hiding* EMM *scheme, then* $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ *is a volume-hiding, forward private and type-II backward private $\mathcal{L}$-secure dynamic,* EMM *scheme.*

3.4.2 Efficiency. We start with the main improvement of $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ over $\mathbf{2ch}_{\mathbf{FB}}$ that is client storage. The client storage of $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ 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 5.1, 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 Appendix G) 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, $\mathbf{2ch}_{\mathbf{FB}}^{\mathbf{s}}$ 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 $u = \mathtt{poly}(n)$. The cost of querying each $\mathsf{EMM}_i^{\mathsf{loc}}$ is equivalent to $O(\log n + u(\mathtt{label}))$ as $O(\log n)$ queries occur and a total of $u(\mathtt{label})$ encrypted values are retrieved in the worst case. Therefore, the worst case communication and computational cost of queries are $O(\ell \log\log n + \ell u(\mathtt{label}) + \log n)$. 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 $O(\ell \log\log n)$.

4 MODIFYING n AND ℓ

In Section 3, we assume that the upper bounds multi-map size (n) and volume (ℓ) never change. We will now present a generic transformations to handle changing upper bounds. In this section, will use n and ℓ as the current size and volume upper bounds respectively. Values n and ℓ will also be inputs for each operation.

```
Let SKE = (Gen, Enc, Dec) be an IND-CPA encryption scheme, $\mathbf{2ch_{FB}}$ be as described in Figure 1 and $\mathbf{PiBas}^*$ (a modified version of $\mathbf{PiBas}$ [9]) be a static, response-hiding, encrypted multi-map scheme as described in [12].
$(\mathsf{st}; \mathsf{EMM}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{params} = (n, \ell, c), \mathsf{MM} = \{(\mathtt{label}_i, \vec{v}_i)\}_{i \in [|\mathtt{label} \in \mathsf{MM}|]})$:
    (1) $\mathbb{C}$ executes $(\mathsf{st2}, \mathsf{EMM2}) \leftarrow \mathbf{2ch_{FB}}.\mathsf{Setup}(1^\lambda, \mathsf{params}, \mathsf{MM})$.
    (2) $\mathbb{C}$ sets $t = 0$ and stores $\mathsf{st} \leftarrow (t, \mathsf{st2})$, and $\mathbb{S}$ stores $\mathsf{EMM} = (\mathsf{EMM2})$.
$(\mathsf{st}'; \mathsf{EMM}') \leftarrow \mathsf{Update}(((\mathsf{op}, \mathtt{label}, \vec{v}), \mathsf{st}), \mathsf{EMM})$:
    (1) $\mathbb{C}$ generates random $x \leftarrow \{0,1\}^\lambda$, pads $\vec{v}$ with dummies until $\vec{v}$ contains $\ell$ values and computes $y \leftarrow \mathsf{Enc}(K_{\mathsf{Enc}}, (\mathsf{op}, \vec{v}))$.
    (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 $\mathsf{MM}_j$.
    (4) For each $\mathtt{label}'$ appearing in at least one of the $\mathsf{MM}_j$, $\mathbb{C}$ computes $\mathsf{MM}_i$ 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 $\mathsf{EMM}$.
$((\mathsf{st}', \vec{v}); \mathsf{EMM}') \leftarrow \mathsf{Query}(((\mathsf{qop}, \mathtt{label}), \mathsf{st}), \mathsf{EMM})$:
    (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 $((\mathsf{st2}', \vec{v}); \mathsf{EMM2}') \leftarrow \mathbf{2ch_{FB}}.\mathsf{Query}(((\mathsf{qop}, \mathtt{label}), \mathsf{st2}); \mathsf{EMM2})$. In the execution, $\mathbb{C}$ uses $L$ as the locations of cached update
        operations for $\mathtt{label}$ in $\mathsf{EMM}_u$ instead of sending a seed to $\mathbb{S}$ to compute these locations.
    (3) $\mathbb{C}$ computes $\mathtt{st}'$ by updating st2 to $\mathsf{st2}'$ and $\mathbb{S}$ computes $\mathsf{EMM}'$ by updating EMM2 to $\mathsf{EMM2}'$.
```

Figure 2: Pseudocode for construction $\mathbf{2ch^s_{FB}}$

4.1 Changing Multi-Map Size n

We start with handling either growing or shrinking the multi-map (i.e., changes to n). To do this, we will leverage a technique used in most common data structure implementations. Consider a dynamic array implementation (such as std::vector in C++). The array is initialized in memory with some fixed capacity upper bound. Once data grows beyond the capacity, the array implementation increases the capacity by some multiplicative factor (such as 2x), allocates new memory for the increased capacity and copies the contents to the new allocation. Our transformation will use the same paradigm.

Consider any dynamic volume-hiding EMM Σ with leakage $\mathcal{L}$ for fixed n and ℓ. We build Σ' with an additional Rebuild functionality.

$(\mathsf{st}', \mathsf{EMM}') \leftarrow \Sigma'.\mathsf{Rebuild}((\mathsf{st}, n, \ell), (\mathsf{EMM}, n, \ell))$:

(1) $\mathbb{C}$ downloads EMM and decrypts using st to get plaintext MM.
(2) $\mathbb{C}$ and $\mathbb{S}$ execute $\Sigma.\mathsf{Setup}(1^\lambda, (n, \ell, \mathsf{params}), \mathsf{MM})$ to receive st' and EMM' respectively.

First, we evaluate the leakage of executing Rebuild. The first step leaks nothing as $\mathbb{C}$ simply downloads EMM. The second step leaks setup leakage $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}, n, \ell)$. So, $\mathcal{L}_{\mathsf{Rebuild}} = \mathcal{L}_{\mathsf{Setup}}$.

When the client reports a change to n, Σ' will first execute Rebuild before running the original algorithm of Σ for either queries or updates. So, there is additional leakage informing when n changes that we will model with N such that $N[i] = \mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}, n, \ell)$ if n changes on the i-th operation and $N[i] = \perp$ otherwise. We choose n to double (halve) to increase (decrease) capacity.

Theorem 6. *Let Σ be a dynamic volume-hiding encrypted multi-map* EMM *with leakage function $\mathcal{L}$ for fixed values of n and ℓ. Then, Σ' is a dynamic volume-hiding* EMM *with leakage* $(\mathcal{L}_{\mathsf{Setup}}, (\mathcal{L}_{\mathsf{Query}}, M), (\mathcal{L}_{\mathsf{Update}}, M))$. *If n is only doubled or halved, Σ' has no increased amortized overhead.*

Proof. For leakage, the simulator can detect when Rebuild is run and simulate setup using M. In terms of efficiency, consider the setting where capacity is doubled. That means, there must have been at least $\Omega(n)$ values added. The cost of Rebuild is $O(n)$ meaning that the amortized overhead is at most $O(1)$ per updated value. A similar argument can be applied if capacity is halved. □

Instantiation with $\mathbf{2ch_{FB}}$ or $\mathbf{2ch^s_{FB}}$. If Σ is chosen to be $\mathbf{2ch_{FB}}$ or $\mathbf{2ch^s_{FB}}$, then $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}, n, \ell) = n$. So, the leakage of Σ' is $\mathcal{L}' = (\mathcal{L}_{\mathsf{Setup}}, (\mathcal{L}_{\mathsf{Query}}, n), (\mathcal{L}_{\mathsf{Update}}, n))$ as M may be upper bounded with $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}, n, \ell) = n$.

Discussion about Approach. At a high level, the transformation is a straightforward approach of downloading, modifying locally and re-uploading the multi-map. To our knowledge, this remains the most efficient technique in the literature. For example, similar techniques were used in [12] for avoiding local storage of large count tables. We also employ similar techniques in $\mathbf{2ch^s_{FB}}$. To our knowledge, techniques with smaller client storage utilize more expensive algorithms including oblivious shuffling or sorting [1, 2, 29, 32]. We leave it as an open problem to improve handling multi-map size changes beyond the straightforward approach.

4.2 Changing Maximum Volume ℓ

For changing values of ℓ, we could also apply the same technique for handling changing n. However, the amortized overhead may be larger as only a small number of keys need to be added to force a change in ℓ. Instead, we present an even simpler transformation that may be applied to either $\mathbf{2ch_{FB}}$ or $\mathbf{2ch^s_{FB}}$.

We augment the Query and Update algorithms in the following way. The state of the client will also include the current maximum volume ℓ. Whenever ℓ changes, the client communicates the new value to the server. Afterwards, the protocols use the new value of ℓ to continue. Combining with techniques in Section 4.1, we get the following theorem that we prove in Appendix H:

Theorem 7. *Let Σ' be either $\mathbf{2ch_{FB}}$ or $\mathbf{2ch^s_{FB}}$ with the above modifications to handle changing n and ℓ with leakage $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}$*

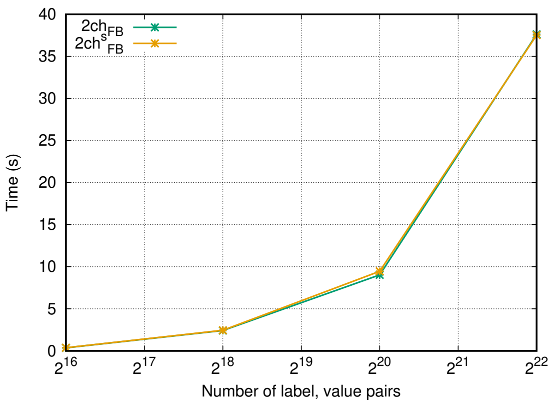


Figure 3: Setup time for $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$

$_{\text{Query}}$, $\mathcal{L}_{\text{Update}}$). *Then,* Σ' *is a dynamic volume-hiding* EMM *with leakage* $\mathcal{L}' = (\mathcal{L}_{\text{Setup}}, (\mathcal{L}_{\text{Query}}, n, \ell), (\mathcal{L}_{\text{Update}}, n, \ell))$.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the practicality of our volume-hiding schemes. First, we describe the experimental setup and our choice of parameters for our constructions. Using these experiments, we aim to answer whether our constructions concretely efficient while providing better privacy and more operations.

5.1 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 our experiments have standard deviations less than 2% of their average and were repeated at least 10 times.

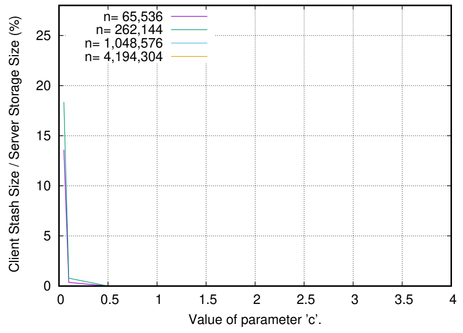


Figure 4: Stash size as a percentage of total encrypted storage size on the server for different values of n and c

Primitives. We use and build on top of cryptographic primitives provided by ring [38] and OPENSSL [41] 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. 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 [12, 35]. 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 [12]. The size of label and value strings will be 20 bytes.

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 $c = 1$. We refer the readers to Figure 3 for a detailed plot of setup times. We also varied the value of c from 0.01 to 4 to study how the value of c effects the stash size when we put up to n values in our encrypted multi-map. This experiment was repeated 100 times and results are plotted in Figure 4. We find that for values of $c \geq 0.1$ the client stash size averaged 0 regardless of the value of n we picked.

Query Protocol. For both our schemes, we computed the total latency taken by the client and the server collectively on average to produce a final query result. We first focus on query times without any updates. 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 5a and 5b show 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 [35] can be made. Here we note that the query times are comparable to the query times in [35] even though our schemes support dynamic operations. The query times of $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^{s}_{FB}}$ are also very comparable even though updates are stored differently in both schemes. Queries for both schemes ranged from 0.027ms to 0.051ms per result.

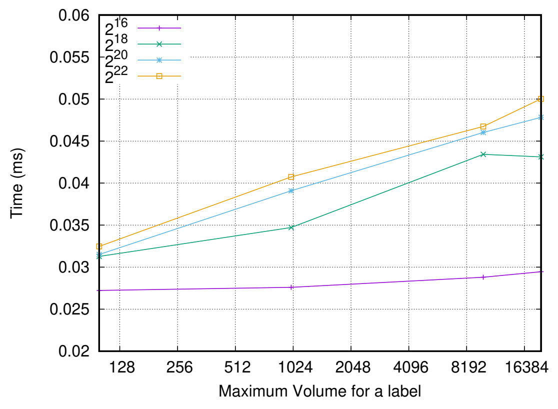
Update Protocol. For $\mathbf{2ch_{FB}}$, the time taken by an update stays under 25ms and for $\mathbf{2ch^{s}_{FB}}$ under 76ms even for maximum volume of 20,000 as shown in Figures 5c and 5d. This is primarily becuase of forward and backward privacy, updates are not directly applied to the two-choice 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 $\mathbf{2ch^{s}_{FB}}$ multiple encrypted structures are downloaded and rebuilt which takes extra time. However, $\mathbf{2ch^{s}_{FB}}$ is still desirable due to smaller permanent client state.

Due to lack of space and for a detailed look at query times interposed with updates to show the effects updates have on a query, we refer the reader to Appendix I.

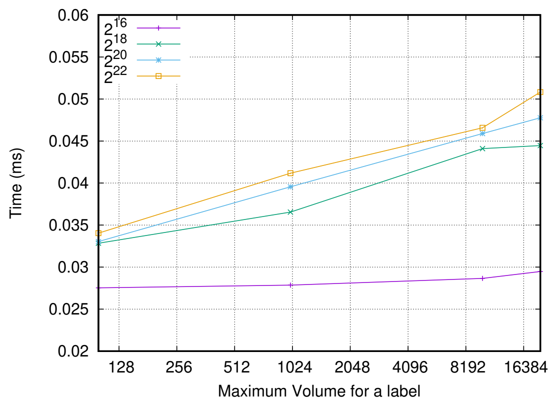
Comparison with DST **[20].** We compare with DST [20]. The other construction based on Pseudo-Random Transform in [20] is lossy in nature and leads to inaccurate query results. Hence, we do not believe a fair comparison is possible there. For DST, we note

	DST				$\mathbf{2ch_{FB}}$				$\mathbf{2ch^s_{FB}}$			
Input MM												
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)	13.8	56.85	220.6	961	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

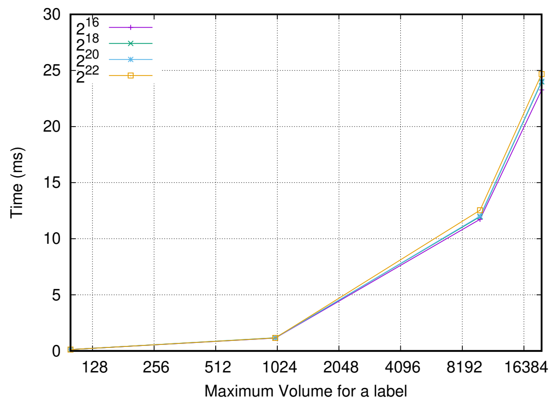
Table 2: Observed sizes of structures. We denote n as the total number of label, value pairs in the input MM.



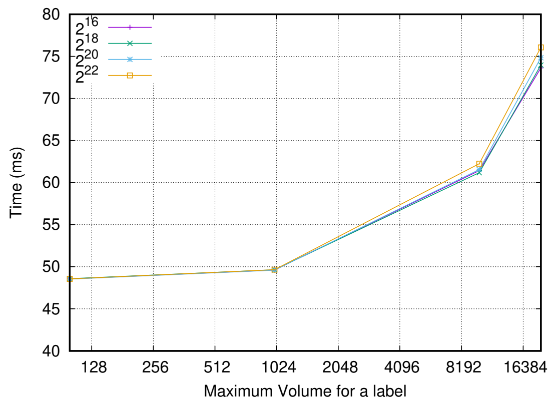
(a) $\mathbf{2ch_{FB}}$ Query



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



(c) $\mathbf{2ch_{FB}}$ Update



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

Figure 5: Query and Update times for different values of ℓ and different database sizes.

that it lacks several features offered by our constructions such as forward privacy and full-dynamicity (see App. A). For a comprehensive treatment, we still present a comparison with DST. We will show that $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ offer the additional functionalities with minimal (or no) increased costs compared to DST.

For updates, DST takes from 7ms-1000ms for ℓ ranging from 128 to 20,000. During an update of a label x, DST downloads all the bins for x, deletes them on the server and re-uploads the edited bins. In comparison Figures 5c and 5d, show that for updates, our schemes have smaller communication and computation than DST. This is not surprising as for an update $\mathbf{2ch_{FB}}$ only uploads a vector of size ℓ to the server and $\mathbf{2ch^s_{FB}}$ rebuilds a series of encrypted structure up until the smallest empty one. For smaller values of ℓ (≤ 1024), $\mathbf{2ch^s_{FB}}$ takes more time than DST during updates due to additional cost of re-executing setup protocols on the underlying data structures dominating the cost incurred due to value of ℓ.

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 as our bins contain 4-5 items each using $c = 1$ whereas bins in DST contain at least 21-32 items (see experiments in [35]). Now taking updates into account during queries, and for different values of ℓ and n, we observed that 12-22 updates on a label before a query on that label would increase the query time to as much as that of DST. This is because downloading these additional updates during our queries makes our communication/computation costs similar to DST (countering our small bin size advantage). This is not surprising as we provide more stronger privacy guarantees. The total cost would still be same or better than DST as this increase in cost of query time is actually amortized over updates in DST.

ACKNOWLEDGMENTS

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 1–9.

[2] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms*. SIAM, 8–14.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *EUROCRYPT 2020*, Anne Canteaut and Yuval Ishai (Eds.). 403–432.

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

[5] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *NDSS 2020*. https://doi.org/10.14722/ndss.2020.23103

[6] Raphael Bost. 2016. Sophos: Forward Secure Searchable Encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1143–1154.

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

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

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

[10] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology - ASIACRYPT 2010*, Masayuki Abe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 577–594.

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

[12] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2020. Dynamic Searchable Encryption with Small Client Storage. In *NDSS*.

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

[14] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. 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*. 315–331.

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

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

[17] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 361–378. https://doi.org/10.1145/3319535.3363210

[18] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. 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*.

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

[20] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *EUROCRYPT 2019*. 183–213.

[21] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and dynamic searchable symmetric encryption. In *International conference on financial cryptography and data security*. Springer, 258–274.

[22] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. 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*. 1329–1340.

[23] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (2009), 1543–1561.

[24] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. 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.

[25] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. 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*. SIAM, 143–156.

[26] Russell WF Lai and Sherman SM Chow. 2017. Forward-secure searchable encryption on labeled bipartite graphs. In *International Conference on Applied Cryptography and Network Security*. Springer, 478–497.

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

[28] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. 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*. 644–655.

[29] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*. Springer, 556–567.

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

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

[32] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2018. CacheShuffle: A family of oblivious shuffles. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[33] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2019. What Storage Access Privacy is Achievable with Small Overhead?. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Amsterdam, Netherlands) *(PODS '19)*. Association for Computing Machinery, New York, NY, USA, 182–199. https://doi.org/10.1145/3294052.3319695

[34] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2020. Leakage Cell Probe Model: Lower Bounds for Key-Equality Mitigation in Encrypted Multi-Maps. In *CRYPTO 2020*.

[35] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 79–93. https://doi.org/10.1145/3319535.3354213

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

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

[38] Brian Smith. 2012. RING. docs.rs/ring/0.17.0-alpha.1/ring/index.html.

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

[40] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*, Vol. 71. 72–75.

[41] docs.rs/openssl/0.10.29/openssl/. 1998. OPENSSL-RUST.

[42] Jiafan Wang and Sherman SM Chow. 2021. Simple Storage-Saving Structure for Volume-Hiding Encrypted Multi-maps. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 63–83.

[43] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption.. In *USENIX Security Symposium*. 707–720.

[44] Yongjun Zhao, Huaxiong Wang, and Kwok-Yan Lam. 2021. Volume-Hiding Dynamic Searchable Symmetric Encryption with Forward and Backward Privacy. Cryptology ePrint Archive, Report 2021/786. https://eprint.iacr.org/2021/786.

A SEMI-DYNAMICITY OF DST [20]

Throughout our work, we refer to the DST 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 [34] shows that, unless one is willing to utilize ORAM-like overheads, leakage of label equality patterns must be revealed by queries. Recall that label equality patterns reveal whether two different operations are performed on the same label or not (see Section 2.4). Using the above approach of replacing an update with two semi-dynamic operations will leak label equality leakage for every update (due to the usage of the semi-dynamic query). This ends up being a significant privacy degradation as label equality leakage during updates 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 LOWER BOUNDS WHEN HIDING ℓ

In this section, we analyze the dynamic volume-hiding definition in [44] that hides the maximum volume ℓ. Additionally, they introduce the notion of (p, ϵ)-correctness meaning that ϵ-fraction of matching values are returned with probability at least p. We refer readers to [44] for both definitions. We show a strong and simple query communication lower bound in this model.

Theorem 8. *Let Σ be a dynamic volume-hiding encrypted multimap EMM according to the definition in [44] that is (p, ϵ)-correct. Then, the sum of the expected query communication and client storage of Σ must be $\Omega(p \cdot \epsilon \cdot n)$.*

Proof. Consider any MM. We show that when MM is input to Σ, the query communication must be $\Omega(\epsilon n)$. To do this, we construct the following adversary $\mathcal{A}$. In the first phase of the definition in [44], $\mathcal{A}$ chooses any label k appearing in MM and constructs MM$'$ with k associated with a value tuple of size $|\mathsf{MM}|$. For operations, $\mathcal{A}$ chooses to query k repeatedly. Note, that the size of the query communication is viewed by the adversary. By the correctness requirement, it must be that $\Omega(\epsilon n)$ values are returned when querying MM$'$ with probability at least p. So, the query communication and client storage must be $\Omega(p \cdot \epsilon \cdot n)$ in expectation. By the volume-hiding requirement and the fact that the adversary sees the size of query communication, this means that queries to MM must satisfy the same requirement. □

For reasonable parameters such as $p \geq 0.5$ and $\epsilon \geq 0.5$ and sublinear client storage, then $\Omega(n)$ expected query communication is required.

C BACKWARD PRIVACY

Definition 7 (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*:*

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

Type-II backward private*:*

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

Type-III backward private*:*

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

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.

D 2ch: PSEUDOCODE AND ANALYSIS

The pseudocode of **2ch** is presented in Figure 6.

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**.

Theorem 9. *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 9, we first prove Theorem 10, Lemma 1 and Lemma 2.

Theorem 10. *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 10. 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 $B_1, \ldots, B_s$ 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) From $\mathsf{leq}(O, o)$, fix smallest i where $\mathtt{label}(o) = \mathtt{label}(O[i])$.
(2) If no such i exists, set $\mathtt{st}[|O| + 1]$ to be a uniformly random string from $\{0, 1\}^\lambda$ and set $i \leftarrow |O| + 1$.

```
Let $F$ and $G$ be PRFs and SKE = (Gen, Enc, Dec) be an IND-CPA encryption scheme.
$(\mathsf{st}; \mathsf{EMM}) \leftarrow \mathbf{2ch}.\mathsf{Setup}(1^\lambda, \mathsf{params} = (n, \ell, c), \mathsf{MM} = \{(\mathtt{label}_i, \vec{v}_i)\}_{i \in [m]})$:
    (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 $j \in [|\vec{v}_i|]$:
            (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 $\mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{label}_i, j, \vec{v}[j]))$ 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 $(\mathtt{label}_i, j, \vec{v}[j])$ 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 $\mathsf{EMM} \leftarrow (B_1, \ldots, B_s)$.
$((\mathsf{st}', \vec{v}); \mathsf{EMM}') \leftarrow \mathbf{2ch}.\mathsf{Query}((\mathsf{st}, (\mathsf{qop}, \mathtt{label})), \mathsf{EMM})$.
    (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 $2\ell$ associated bins that are sent to $\mathbb{C}$.
    (4) $\mathbb{C}$ decrypts all $2\ell$ bins and returns $\vec{v}$ consisting of all values that are tagged with $\mathtt{label}$ in the bins as well as in $\mathtt{Stash}$.
$(\mathsf{st}'; \mathsf{EMM}') \leftarrow \mathbf{2ch}.\mathsf{Update}((\mathsf{st}, (\mathsf{op}, \mathtt{label}, \vec{v}')), \mathsf{EMM})$:
    (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 $2\ell$ associated bins that are sent to $\mathbb{C}$.
    (3) $\mathbb{C}$ decrypts all $2\ell$ bins and compiles $\vec{v}$ 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 $\vec{v}$. All entries corresponding to $\mathtt{label}$ are removed from $\mathtt{Stash}$.
    (5) If op = app, $\mathbb{C}$ appends $\vec{v}'$ to $\vec{v}$. If op = edit, $\mathbb{C}$ sets $\vec{v} \leftarrow \vec{v}'$. If op = del, $\mathbb{C}$ removes the values in $\vec{v}'$ from $\vec{v}$. If op = rm, $\mathbb{C}$ sets $\vec{v} \leftarrow \perp$.
    (6) For $i \in [|\vec{v}|]$:
        (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 $\mathsf{Enc}(K_{\mathsf{Enc}}, (\mathtt{label}, i, \vec{v}[i]))$.
        (c) If both bin $b_0$ and bin $b_1$ contain no nodes with dummy encryptions, add $(\mathtt{label}, i, \vec{v}[i])$ to $\mathtt{Stash}$.
    (7) $\mathbb{C}$ re-encrypts all $2\ell$ bins and sends back to $\mathbb{S}$ for storage.
```

Figure 6: Pseudocode for Construction 2ch

(3) Return $\mathtt{st}[i]$.

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

(1) From $\mathsf{leq}(O, o)$, fix smallest i where $\mathtt{label}(o) = \mathtt{label}(O[i])$.
(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 pseudorandomness of F.
- $\mathbf{Game}_2$ replaces the IND-CPA encryption 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 9. Follows directly from Theorem 10, Lemma 1 and Lemma 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 [33], 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.

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.

E SECURITY PROOF OF $\mathbf{2ch_{FB}}$

For convenience, we present the leakage function $\mathcal{L}$ of $\mathbf{2ch_{FB}}$ (repeated from Section 3.3.1).

- $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}) = n$.
- $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{uop})$.
- $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{leq}(O, o), \mathsf{qop})$.

As a reminder, the above leakage means that only the size of the multi-map is leaked during setup. During the update operation, only the maximum volume and update operation is leaked. Finally, the maximum volume, query operation and label equality leakage pattern are revealed during queries.

Theorem 11. *If* SKE *is an IND-CPA-secure encryption and* F, G *are pseudorandom functions and* H *is modeled as a random oracle, then for every* $n \geq \ell \geq 1$, $\mathbf{2ch_{FB}}$ *is an adaptive* $\mathcal{L}$*-secure dynamic STE scheme for multi-maps.*

Proof. We consider a stateful simulator $\mathcal{S}$ with state st that works as follows:

```
EMM ← $\mathcal{S}$.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 $\perp$ and
      initialize an empty multi-map $\mathsf{EMM}_u$.
  (3) Set st ← $(\mathsf{M}, U)$ where M is an empty map and $U$ is an
      empty array.
  (4) Return $(B_1, \ldots, B_s, \mathsf{EMM}_u)$.
Response ← $\mathcal{S}$.SimQuery($1^\lambda$, qop, $\ell$, leq($O, o$)):
  (1) Using leq($O, o$), find smallest $i$ such that label($o$) = label($O$
      [$i$]) and $O[i]$ is a query.
  (2) If no such $i$ exists, set $i \leftarrow |O| + 1$ and 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
      label($O[j]$) = label($o$). If no such $j$ exists, set $j$ to $-1$. For
      all $m > j$, is label($o$) = 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 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 $(A_{0,i}, A_{1,i})_{i\in[1,\ell]}$.
(st, Response) ← $\mathcal{S}$.SimUpdate($1^\lambda$, uop, $\ell$):
  (1) Compute an $y$ that is an encryption of tuple $(\perp, \vec{v}^\perp)$ where
      $\vec{v}^\perp$ consists of $\ell$ 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 pseudorandomness 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 outputs 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. ◻

Note that the random oracle assumption may be removed by using H as a pseudo-random function PRF and the client sending all PRF evaluations to the server.

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. ◻

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 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. ◻

F SECURITY PROOF OF $\mathbf{2ch^s_{FB}}$

For convenience, we present the leakage function $\mathcal{L}$ of $\mathbf{2ch^s_{FB}}$ (repeated from Section 3.4.1).

- $\mathcal{L}_{\mathsf{Setup}}(\mathsf{MM}) = n$.
- $\mathcal{L}_{\mathsf{Update}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{uop})$.
- $\mathcal{L}_{\mathsf{Query}}(\mathsf{MM}, (O, o)) = (\ell, \mathsf{leq}(O, o), \mathsf{qop})$.

The leakage is identical to $\mathbf{2ch_{FB}}$ for all of setup, updates and queries. We note that the proof will consider the leakage of $\mathcal{L}_{\mathsf{loc}}$ from $\mathsf{EMM}^{\mathsf{loc}}_i$. However, this turns out to be a subset of label equality leakage.

Theorem 12. *If* SKE *is an IND-CPA-secure encryption and* F, G *are pseudorandom functions, then for every* $n \geq \ell \geq 1$, $\mathbf{2ch^s_{FB}}$ *is an adaptive* $\mathcal{L}$*-secure dynamic STE scheme for multi-maps.*

Proof. We will utilize a simulator $\mathcal{S}^{pi}$ for our initialization of each $\mathsf{EMM}^{\mathsf{loc}}_i$ using the $\mathbf{PiBas}^*$ 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 $\mathcal{S}'.\mathsf{SimSetup}(1^\lambda, n)$

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

(1) Using the total number of updates so far, determine encrypted multi-maps $\mathsf{EMM}^{\mathsf{loc}}_i$ that are 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 `label`(o) = `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 `label`$(O[j])$ = `label`(o). If no such j exists, set j to -1. For all $m > j$, if `label`(o) = `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 $(A_{0,i}, A_{1,i})_{i \in [1,\ell]}$.

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

(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 $(\perp, \vec{\mathsf{v}}^{\perp})$ where $\vec{\mathsf{v}}^{\perp}$ 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 pseudorandomness of F, G.
- $\mathbf{Game}_2$ 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.
- $\mathbf{Game}_3$ 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. □

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 5. Follows directly from above. □

G VARIANTS OF $\mathbf{2ch_{FB}}$ AND $\mathbf{2ch^s_{FB}}$

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 $\mathsf{EMM}^{\mathsf{loc}}_i$ 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 $\mathsf{EMM}^{\mathsf{loc}}_i$ that is selected to be locally reconstructed is one that has some space newly freed due to a deletion of an entry from $\mathsf{EMM}^{\mathsf{loc}}_i$ during update resolution in a past query operation.

Discussion about Forward Privacy. We note that the server storage increases as there are more update operations without intermediate query operations. Similar to the growing client storage of $\mathbf{2ch_{FB}}$ discussed in Section 3.3.2, the additional server storage enables providing stronger protection for updates without intermediate queries. We leave it as an open problem to achieve this protecting without additional storage costs.

H PROOF OF THEOREM 7

Proof of Theorem 7. We note that the proof is essentially identical to Theorems 4 and 5. The only modification is that after each operation, the simulator is provided n and ℓ. The simulator will run the same algorithm with these newly provided values. For correctness, both $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$ compact their results such that non-dummy values appear before dummy values. As long as ℓ is a valid upper bound, then all correct values are always returned. □

I EFFECTS OF UPDATES ON QUERIES

We refer to Figure 7 for a detailed look at $\mathbf{2ch_{FB}}$'s query times interposed with its updates to show the effects updates have on a query. In each of the graphs in this figure, 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 graphs, 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 observed that $\mathbf{2ch^s_{FB}}$ (Figure 8) which saves a lot on client storage, 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 than $\mathbf{2ch_{FB}}$. We, however, note that our query times are in order of microseconds ($25\mu s$ to $50\mu s$) per single label, value pairs for both $\mathbf{2ch_{FB}}$ and $\mathbf{2ch^s_{FB}}$. Compared to Figures 5a and 5b, we do see a slight increase as queries now need to apply updates.

J CIRCUMVENTING LABEL EQUALITY LOWER BOUND [34]

In a work by Patel *et al.* [34], it was shown that encrypted multi-map scheme that aims to leak anything less than label equality leakage will inevitably require $\Omega(\log n)$ overhead that is similar to an oblivious RAM (ORAM). Throughout our work, we justify the leakage of label equality leakage as a way to obtain efficiency faster than ORAMs and circumvent this lower bound.

One may wonder whether it is possible to circumvent the lower bound in [34] in other ways without leaking label equality. One attempt may be to restrict the sequence of valid operations to avoid the one that was used to prove the lower bound in [34]. Recall that the proof in [34] considers a hard sequence of k operations with $k/2$ updates with value tuples of length ℓ to unique labels followed by $k/2$ queries to the same labels in any order. For this set of sequences, it was shown that $\Omega(\log(k\ell))$ overhead is required for a wide range of choices for k and ℓ. One obtains the above lower bound by setting $k\ell = n^\alpha$ for any constant $0 < \alpha \leq 1$.

In theory, it is possible to construct an encrypted multi-map that is faster for sequences that are not the above hard sequences without leaking label equality. That is, the construction is faster for non-hard sequence but slower for hard sequences. Unfortunately, the set of hard sequences is large and considers a natural setting of updating $k/2$ different labels followed by querying them. Therefore, the practical benefits of such a construction remain unclear. Nevertheless, we leave it as an interesting open question as to whether this efficiency dichotomy is achievable.

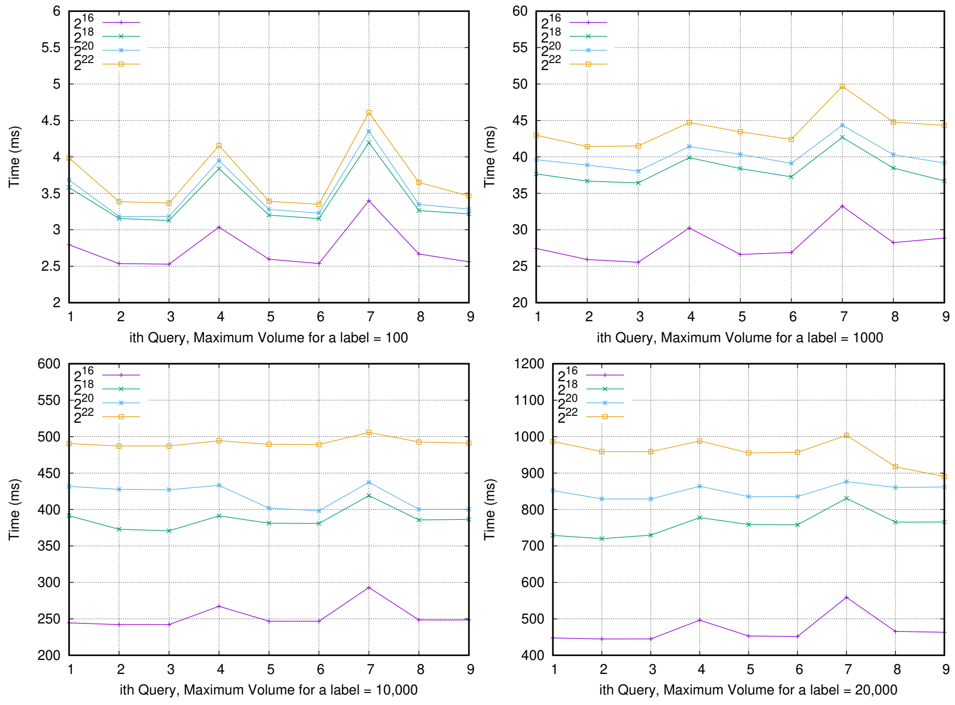


Figure 7: This is the time cost when executing queries in 2ch_{FB} for $\ell \in \{100, 1000, 10000, 20000\}$. For each value of ℓ, we executed queries over varying database sizes as shown in the graphs.

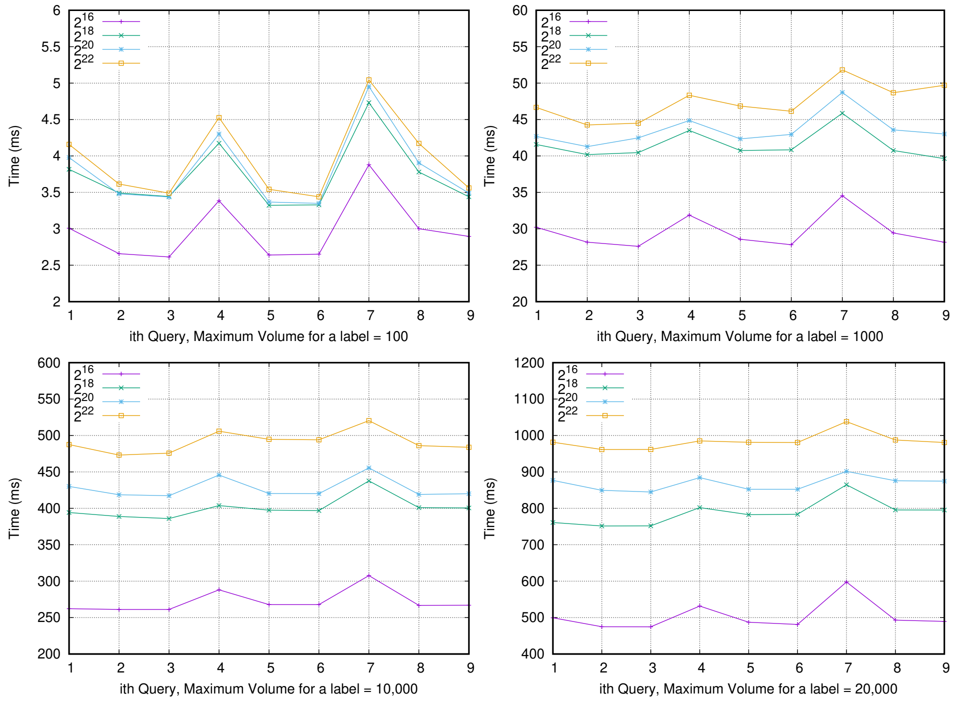


Figure 8: Similar to Fig. 7, this figure represent total time taken when executing queries in $2\text{ch}^{s}_{\text{FB}}$.