A preliminary version of this paper appears in *the 14th International Conference on Security and Cryptography for Networks (SCN 2024).*

Encrypted Multi-map that Hides Query, Access, and Volume Patterns

Alexandra Boldyreva[1⋆] Tianxin Tang[2⋆⋆]

Abstract

We present an encrypted multi-map, a fundamental data structure underlying searchable encryption/structured encryption. Our protocol supports updates and is designed for applications demanding very strong data security. Not only it hides the information about queries and data, but also the query, access, and volume patterns. Our protocol utilizes a position-based ORAM and an encrypted dictionary. We provide two instantiations of the protocol, along with their operation-type-revealing variants, all using PathORAM but with different encrypted dictionary instantiations (AVL tree or BSkiplist). Their efficiency has been evaluated through both asymptotic and concrete complexity analysis, outperforming prior work while achieving the same level of strong security. We have implemented our instantiations and evaluated their performance on two real-world email databases (Enron and Lucene). We also discuss the strengths and limitations of our construction, including its resizability, and highlight that optimized solutions, even with heavy network utilization, may become practical as network speed improves.structured encryption searchable encryption provable security

1 Introduction

Motivation and Prior Work. The growth of cloud storage is soaring at an unprecedented rate. It is predicted that by 2025, 100 zettabytes of data will be stored on the cloud, accounting for 50% of the world's data [1]. Although cloud storage providers are often trusted, the confidentiality of stored data can still be compromised, posing a significant threat to customers and violating regulations such as PCI DSS, HIPAA, EU Data Protection, which mandate data confidentiality.

Adding data confidentiality to cloud applications such as outsourced database is challenging due to the limitations of off-the-shelf encryption schemes. To address this issue, extensive research have been conducted in the areas of searchable symmetric encryption (SSE) and structured encryption (STE) over the past 20 years.

The existing SSE/STE solutions vary significantly in terms of security and efficiency, as these properties often conflict with each other. Earlier works, such as [2]–[5] prioritize efficiency over security. Most of these schemes leak query, access, and volume patterns at the very least,[1] and some solutions, e.g., [3], even leak the data (i.e., database records) equality. Later, various works [6]–[16] have shown

⋆Georgia Institute of Technology, USA. Email: sasha@gatech.edu.

⋆⋆Eindhoven University of Technology, Netherlands. Email: ac.tianxin.tang@gmail.com. Most of this work was conducted while the author was at Georgia Institute of Technology, USA.

This is the full version of an article submitted by the authors to the IACR and to Springer Nature. The version published by Springer Nature appears in the proceedings of SCN 2024.

[1] By query pattern, we mean the information that reveals whether the same keyword has been used repeatedly in search or update queries. The access pattern refers to the information indicating whether the same data is accessed on the server. The volume pattern, on the other hand, is leaked when the amount of communication differs among queries.

that seemingly innocuous leakage can lead to devastating attacks that reveal a significant amount of information about the data.

Accordingly, more recent research [17]–[22] has shifted towards finding solutions with much stronger security, aiming to develop provably-secure solutions that hide query, access, and volume patterns. However, most works do not succeed in providing constructions that provably hide query, access patterns, and volume patterns. These patterns are hidden in Oblix [18], but the protocol relies on secure hardware such as Intel's SGX. Reichert et al. [19] extends Oblix to support range queries and adds differential privacy to the volume pattern. However, as demonstrated in [13], [23], a differentially private volume pattern does not provide sufficient privacy guarantees. Patel et al. [22] propose a volume-hiding construction, but it leaks the query pattern. Their construction is vulnerable to a query-recovery attack by Oya and Kerschbaum [13].

The above constructions have a significant limitation for deployment: they are static and lack support for updates. There has been growing interest in secure dynamic databases [5], [24]–[39], which brings additional requirements of forward security — update queries should not reveal which keywords are involved [24], and backward security — search queries should not reveal removed documents or keywords [25].

We discuss the ones with the strongest security. Chamani et al. [27] propose three constructions: Mitra, Orion, and Horus, with Orion achieving the strongest security among the three. However, none of these schemes are volume-hiding, and the attempt to achieve it through access padding will greatly impact efficiency. In Section 4, we demonstrate that above schemes and SEAL [21] all suffer from significant round complexity overhead from padding ORAM accesses for volume-hiding, even when batched access is allowed. Miers and Mohassel [26] propose an efficient SSE scheme that allows oblivious updates, but it has the drawback of leaking substantial information about the index through the incorporation of non-oblivious reads for efficiency (hence leaks the query and access patterns for search queries). Additionally, as explained in [25], their scheme is not forward-secure.

Finally, George et al. [37] present a leakage suppression framework to eliminate the query (i.e., query-equality) leakage for dynamic schemes while preserving the volume-hiding property. Though the security of their construction is strong, the framework is mostly theoretical and there is no implementation. Our analysis shows that an efficient implementation may not be possible. Although their protocol additionally supports dynamically increasing M, the total number of keywords, we chose not to pursue this for several reasons, which we explain in the remarks. Xu et al. [40] study a folklore approach where the client downloads, updates, and uploads the encrypted index for each query. While this approach is considered secure in a straightforward manner, it suffers from severe bandwidth overhead.

Despite tremendous research progress, a dynamic protocol for outsourced databases that hides query, access, and volume patterns, is both backward and forward secure, and reasonably efficient, remains unavailable. The current state is somewhat stagnant, as it is widely recognized that leakage-abuse attacks can be very dangerous, but at the same time, avoiding leaking volume and other patterns by relying on ORAM and padding is believed to yield prohibitively slow protocols.

Recently there have been significant progress in the related area of private information retrieval (PIR). But PIR is a somewhat more difficult problem for the setting where there are multiple clients and the server owns the data. The most efficient solutions rely on fully-homomorphic encryption and either require computation linear in the database size or massive storage and pre-processing. We discuss while PIR-based solutions are not likely to yield efficient SSE/STE in the full paper [41].

Even though there are no efficient SSE/STE solutions with no leakage, companies are eager to deploy Recently, major database vendor MongoDB, has built and integrated an SSE/STE scheme called queryable encryption (QE) into its database system and made QE available.[2] Not surprisingly, the attacks start to appear [42].

[2] https://www.mongodb.com/docs/manual/core/queryable-encryption/

Should we accept that having leakage and attacks is unavoidable with SSE/STE solutions which are practical or close to be practical? Our work aims to show that it is too early to give up and settle.

Our Focus. Our goal is to explore the possibility of constructing a protocol with extremely strong security that is not prohibitively slow. Our result is a construction that outperforms prior proposals with comparable security. While it is understood that a protocol with such strong security (particularly volume-hiding) may not be highly efficient, our result shows that one does not have to give up on very strong security. As network speed improves, optimized solutions, even with heavy network utilization, may become practical.

In this work, we concentrate on the common scenario where the client performs *exact* match queries to search for documents or database records that match a given keyword. In line with almost all the related work, we use [4]'s database abstraction and focus on constructing an *encrypted multi-map* that maps labels to values. This has direct application to encrypted databases using STE and SSE.

Like previous work, we consider limited client storage, which cannot be linear in either the number of labels or values. This is to ensure that the scheme can be easily adapted to a multi-client setting or to scenarios where the same client uses multiple devices. The client(s) must have the same secret key, and their states must kept in sync to maintain the database while mitigating the query/access leakage. The sublinear client storage also aligns with the scalability requirement necessary for building a potentially large encrypted database. For instance, the feature vectors of media data, such as images can be high-dimensional, and the associated label space can be orders of magnitude larger than the English word space used in email databases.

As for trust consideration, we are only interested in secure solutions that do not rely on the assumption of non-colluding servers or the use of secure hardware, such as SGX. This is because the assumption of non-colluding servers can be difficult to meet in practice, as even if non-colluding cloud providers exist, the service is typically provided by a single company with access to all the data. Furthermore, relying on secure hardware places trust in hardware manufacturers, and the promised security guarantees can be difficult to verify, not to mention the potential privacy risks due to side-channel attacks.

We will compare our protocol with those from[37], [40] as they are the only ones with comparable strong security.

Protocol Overview. In order to build an encrypted multi-map, one cannot simply use oblivious RAM (ORAM) [43] as it is. If we store each label and its associated values in an ORAM block and retrieve the values using the label, the client needs to know its associated block id for ORAM access. However, the limitation of sublinear storage prohibits storing locally the mapping or hash tables (e.g., Cuckoo hashing). Furthermore, using a cryptographically-secure hash function would lead to an ORAM of infeasible size.

Generic Construction. At a high-level, our encrypted multi-map protocol EMM consists of two components: an ORAM, which can be viewed as an encrypted array, and an encrypted dictionary (map); both support oblivious access. The client assigns each label an ORAM index (i.e., block id), and stores the label's associated list of values (padded to some maximum length for volume hiding) in the block indicated by the ORAM index. The ORAM enables the client to retrieve the block using the index, and its security properties guarantee that the server does not learn anything about the client's data or the index, including the query, access, and volume patterns.

We note that while our protocol does not use novel building blocks, our techniques to combine and optimize them are novel, and most importantly lead to a security result that is significantly stronger than most prior work. Our approach to improve efficiency is simple yet effective: we pack all values (e.g., database records or document identifiers) into a single block, avoiding an increase in bandwidth costs and significantly reducing the round complexity in search queries compared with padding ORAM accesses as performed by SEAL to hide volume leakage.

To meet the restriction on the client's storage, we use an encrypted dictionary with oblivious access. The encrypted dictionary stores the mapping between labels and ORAM indices on the server. Thus the client can first retrieve the label's index from the dictionary, then query the ORAM to obtain the list of values. However, this solution is not very efficient, as each ORAM access can incur many rounds of communication.

To improve efficiency, we analyze ORAM operations with a focus on position-based ORAMs. These ORAMs use position maps to maintain mappings between the ORAM indices and physical block addresses on the server. The client keeps the position map secret for oblivious access. To prevent the client's storage from growing linearly with the number of blocks, the position map is typically stored recursively in smaller ORAMs on the server, but this results in high round complexity during access. To resolve this issue, we store the position map in the encrypted dictionary alongside the label-index mapping, which is necessary for access. Our protocol naturally supports batched operations — the client can update one label and the associated values at once, with no additional cost compared with the atomic operation, which updates only one label and value pair per operation.

We also improve the efficiency of our protocol by extending the functionality of the building blocks. Firstly, we extend the encrypted dictionary to support a new operation type that combines standard Get and Put into one. This significantly reduces bandwidth costs and round complexity, especially for less efficient encrypted dictionary instantiations that hide query and access patterns (i.e., oblivious maps). Secondly, we extend the (non-recursive) position-based ORAM syntax to include ReadUp, combining Read and Write into one operation. Some instantiations, such as PathORAM [44] and RingORAM [45] naturally support this modification. These extensions for ORAM and encrypted dictionary can be achieved generically using standard operations, but their efficient realization is specific to certain instantiations.

Instantiations. We present two instantiations, $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$, of our generic encrypted multi-map protocol, along with their operation-type-revealing variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$. These variants allow an attacker to determine whether the client is performing a search or update. All our instantiations utilize standard non-recursive position-based ORAM, such as PathORAM [44] and RingORAM [45], but differ in the encrypted dictionary instantiation. $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ use AVL-tree-based encrypted dictionary OAvlTreeE [46] and AvlTreeE, respectively, hiding the query, access, and volume patterns. AvlTreeE is our modification of OAvlTreeE that allows revealing the operation type. $\mathsf{EMM}_{\mathrm{bskip}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$ are built with encrypted dictionary, instantiated by oblivious BSkiplist with encryption (OBSkiplistE) [47], [48] and its operation-type-revealing variant BSkiplistE, which we propose. Unlike [47], [48], which only provide an oblivious map, our optimization and security relaxation are tailored specifically for the encrypted multi-map, commonly used in SSE/STE. Most existing encrypted multi-map constructions do not hide the operation type. In the applications where it is acceptable, which could be many, $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ (or $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$) permits enhancement in bandwidth costs for Get (or GetUp), and at least $3\times$ (or $2\times$) improvement in round complexity compared with hiding the operation type. For a comprehensive comparison of their efficiency, refer to the complexity analysis (see full paper [41]) and the implementation part (Section 8).

We observe that non-recursive position-based ORAM such as PathORAM and RingORAM (we choose to use PathORAM for simplicity), a crucial building block of our instantiations, satisfies the property we need for optimization — either Read or Write involves block decryption, reading, overwriting (if applicable), re-encryption, and write-back. Thus it supports ReadUp naturally as we can update the block after the read, then perform the standard re-encryption and write-back procedures, incurring negligible communicational or computational overhead compared with Read or Write. Similarly, as two of our encrypted dictionary instantiations use PathORAM managing the dictionary data, they can also be extended to support GetUp without additional costs — find the block storing the matched label (i.e., keyword), update the block value, encrypt the block, and write-back. For more details, we refer to Section 5.

Security Analysis. To analyze the security of our protocol, we adopt the formal security definition for dynamic structured encryption [37], which is also a suitable definition for encrypted multi-maps.

We prove that our protocol satisfies the security definition, with setup leakage limited to the upper-bound size of the multi-map. If we consider each label to be used as a keyword, with value fields storing the document identifiers, such leakage can be interpreted as the maximum number of keywords and documents (associated with each keyword), and the maximum keyword size supported by the system. Our protocol also has access leakage being empty or revealing only the operation type depending on the instantiations. This requires that the underlying encrypted dictionary and non-recursive position-based ORAM be adaptively STE secure and with (almost) no access leakage. Such building blocks exist under the standard cryptographic assumption (i.e., IND-CPA of the blockcipher-based mode of operation used for encryption). Our security results imply that all our instantiations achieve strong backward and forward privacy.

Note that the security level our protocol achieves is extremely strong, and this was not possible for prior constructions with reasonable efficiency. To justify the value of our construction we have to of course analyze its performance.

Performance Analysis. To the best of our knowledge, the two approaches that construct or manage an encrypted multi-map achieving similarly strong security are Xu et al.'s baseline approach [40] and George et al's STE with dynamic leakage suppression [37]. Since [37] does not provide an implementation, we compare the concrete complexity in the full paper [41], showing that our instantiations outperformed the baseline in bandwidth cost and are much more efficient than [37] in bandwidth and round complexity.

Furthermore, the asymptotic complexity analysis provided in the full paper [41] shows that our instantiations outperformed [37] asymptotically in round complexity, and at least equal to and better than theirs in bandwidth cost under the natural assumptions specified in the full paper.

We also implemented all our instantiations, evaluated their performance on two real-world email databases Enron and Lucene, commonly used to evaluate efficiency of SSE protocols, and showed that they provide quite reasonable performance for extremely strong security level (see Section 8 for details). We found the operation-revealing variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$, $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$ outperformed their operation-type-hiding counterparts $\mathsf{EMM}_{\mathrm{avl}}$, $\mathsf{EMM}_{\mathrm{bskip}}$, respectively. The BSkiplist-based instantiations $\mathsf{EMM}_{\mathrm{bskip}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$ performed better than the AVL-tree-based $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ in respective operation-type-hiding and revealing categories, incurring fewer rounds and less query latency.

On Resizability and Document Retrieval. In the full paper, we discuss why we focus on an encrypted multi-map with a bounded size instead of a resizable design. We also discuss how the encrypted multi-map is extended to retrieve large documents in SSE.

1.1 Remarks

On Resizability. As previously mentioned, supporting updates is crucial for many applications. In addition to including operations such as addition and deletion, it is useful to avoid fixing the maximum sizes for the number of keywords and documents. This property is referred as *"full dynamism"* in [49].

We first observe that a truly dynamic scheme also needs to avoid limiting the multi-map tuple length, which corresponds to the maximum number of documents per keyword. Otherwise, an existing keyword cannot be added to newly-inserted documents once the keyword's associated tuple has already reached the maximum length. Yet, all dynamic volume-hiding constructions, including [49], fix the tuple length. We show in Appendix E.1 that allowing a variable tuple length yields a simple file-injection attack to recover the keywords in the encrypted database exploiting the change in the tuple length.

So let us go back to "fully" dynamic schemes with fixed tuple length l, referred to as *resizable*. We could extend our construction to achieve this property by using the existing resizable variants

of PathORAM [50] or RingORAM [51]. Varying the number of ORAM blocks would translate into changing the number of keywords M without the limit. The number of documents N can also be changed but with an upper bound $M \times l$.

However, we observe that attaining resizability in the number of keywords with very strong security properties, such as perfect volume-hiding, brings significant performance challenges. This is because to prevent volume-based attacks, whenever inserting or deleting documents associated with a keyword w, the changes in the size of the encrypted index must be independent of the existence of w in the database. This results in performance issues in prior works, such as an increase in the size of the encrypted index for every search operation in [49], and a fixed size increase for every update operation including deletion in [52]. Even the more efficient operation-type-revealing variants only support fake deletions, and does not reduce the size of the encrypted index. These potentially lead to heavy costs for databases with frequent overwrites. Hence, it is questionable to promote resizable dynamic volume-hiding solutions as practical. In comparison, our construction, with a fixed maximum size on the multi-map, allows for real deletions and updates without size increase.

2 Preliminaries

Notation and Convention. For any $n \in \mathbb{N}$, we let $[n]$ denote the discrete range $[1, n]$. For any vector or ordered set $\texttt{vset}$, we let $\texttt{vset}[i]$ denote the i-th element and let $\#\texttt{vset}$ denote the number of elements in $\texttt{vset}$. We use $|\texttt{vset}|_y$ to denote the size for representing $\texttt{vset}$ in unit y, e.g., $|\texttt{vset}|_2$ stands for $\texttt{vset}$'s size in the number of bits. The algorithms are randomized and polynomial-time (in the security parameter) unless otherwise specified. We use $\mathsf{out} \xleftarrow{\$} A$ to indicate out is generated by some algorithm A using its internal randomness. We use PPT for the short abbreviation of probabilistic-polynomial-time and let $\mathsf{negl}(\cdot)$ to denote the negligible function.

Interactive Algorithm. Our protocol and its building blocks are interactive two-party protocols, so to formally define them we are going to use the following notation. For every (two-party) interactive algorithm, we use the subscripts to denote the participating parties. More specifically, for arbitrary interactive algorithm $\mathcal{I}$ run between A and B, we use the notation $(\mathsf{output}_A, \mathsf{output}_B) \leftarrow [\mathcal{I}_A(\mathsf{input}_A), \mathcal{I}_B(\mathsf{input}_B)]$.

Dictionary & Multi-Map. One of our building blocks is an encrypted dictionary (map), and our protocol is essentially an encrypted multi-map, so we define the corresponding objects here.

Given label space $\mathbb{L}_{\mathsf{dx}}$, value space $\mathbb{V}_{\mathsf{dx}}$, a *dictionary* is an unordered set of pairs denoted by $\mathsf{DX} = \{(\ell_i, v_i)\}_{i \in \mathbb{N}}$, where $\ell_i \in \mathbb{L}_{\mathsf{dx}}$ and $v_i \in \mathbb{V}_{\mathsf{dx}}$. We use $\mathsf{DX}[\ell_i]$ to denote ℓ_i's associated value, i.e., $\mathsf{DX}[\ell_i] = v_i$.

Given label space $\mathbb{L}_{\mathsf{mm}}$, value space $\mathbb{V}_{\mathsf{mm}}$, a *multi-map* is an unordered set of pairs denoted by $\mathsf{MM} = \{(\ell_i, \texttt{vset}_i)\}_{i \in \mathbb{N}}$, where $\ell_i \in \mathbb{L}_{\mathsf{mm}}$ and $\texttt{vset}_i \in \mathbb{V}_{\mathsf{mm}}$ is a tuple (or a set) associated with ℓ_i, i.e., $\mathsf{MM}[\ell_i] = \texttt{vset}_i$.

3 Protocol & Building Block Definitions

Recall that we build an interactive protocol, where the server stores the lists of document identifiers for all keywords in encrypted form, and the client who has the secret key can retrieve the document identifiers for any keyword, as well as update the data. As we mentioned in the Introduction, this is basically an encrypted multi-map protocol, where the multi-map's labels are keywords and the values are lists of document identifiers. Our building blocks are also interactive protocols: ORAM (encrypted array) and encrypted dictionary (map). Definitions for structured encryption (STE) [49], [53] are general enough to cover all these protocols, so we recall the formal definitions below. For each

case, we will specify the label and value spaces of the underlying data structure and also define its functionality.

3.1 Structured Encryption (STE)

Structured encryption STE is defined by the label space $\mathbb{L}$, value space $\mathbb{V}$, operation type space $\mathbb{O}$, functionality $\mathcal{F}$, two algorithms, and one two-party protocol:

- $(K, \mathsf{EDS}) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, z, \mathsf{ds})$: is a randomized algorithm run by client $\mathbf{C}$ that takes as input a security parameter $\kappa \in \mathbb{N}$, auxiliary information $z \in \{0,1\}^*$ (e.g., the maximum size of the data structure), data structure $\mathsf{ds} \in \mathbb{L} \times \mathbb{V}$ [3] with functionality $\mathcal{F}$ and outputs secret key K and encrypted data structure EDS.
- $(v^*, \mathsf{EDS}^*) \leftarrow [\mathsf{Access}_{\mathbf{C}}(K, \mathsf{op}, \ell, v), \mathsf{Access}_{\mathbf{S}}(\mathsf{EDS})]$: is a two-party protocol executed between client $\mathbf{C}$ and server $\mathbf{S}$, where $\mathbf{C}$ inputs secret key K, operation type $\mathsf{op} \in \mathbb{O}$, label $\ell \in \mathbb{L}$, and value $v \in \mathbb{V}$; server inputs EDS; at the end, $\mathbf{C}$ receives $v^* \in \mathbb{V}$ and $\mathbf{S}$ updates the encrypted data structure to EDS^*.
- $\mathsf{ds} \leftarrow \mathsf{Dec}(K, \mathsf{EDS})$: is a deterministic algorithm that takes as input secret key K, encrypted data structure EDS and outputs ds.

Correctness. We define the decryption correctness and Access's operational correctness as follows. For all $\kappa \in \mathbb{N}$, all $z \in \{0,1\}^*$, all data structure (with data) $\mathsf{ds} \subseteq \mathbb{L} \times \mathbb{V}$ with functionality $\mathcal{F}$, we say that EDS *instantiates* ds if and only if for all (K, EDS) output by $\mathsf{Setup}(1^\kappa, z, \mathsf{ds})$, $\mathsf{Dec}(K, \mathsf{EDS}) = \mathsf{ds}$; let $\mathsf{ds}_1 \leftarrow \mathsf{ds}$ and $\mathsf{EDS}_1 \leftarrow \mathsf{EDS}$, after applying an arbitrary polynomial-size sequence of operations $\{(\mathsf{op}_i, \ell_i, v_i)\}_{i \in [q]}$ to EDS_1, where $\mathsf{op}_i \in \mathbb{O}, \ell_i \in \mathbb{L}, v_i \in \mathbb{V}$; for all $i \in [q]$, $(v_{i+1}, \mathsf{EDS}_{i+1}) \leftarrow [\mathsf{Access}_{\mathbf{C}}(K, \mathsf{op}_i, \ell_i, v_i), \mathsf{Access}_{\mathbf{S}}(\mathsf{EDS}_i)]$, we require $v_{i+1} = v'_{i+1}$ and $\mathsf{Dec}(K, \mathsf{EDS}_{i+1}) = \mathsf{ds}'_{i+1}$, where $(v'_{i+1}, \mathsf{ds}'_{i+1}) \leftarrow \mathcal{F}(\mathsf{ds}_i, \mathsf{op}_i, \ell_i, v_i)$.

Remark 1 (On Functionality $\mathcal{F}$). We define encrypted multi-map, encrypted dictionary functionality $\mathcal{F}_{\mathsf{DS}}$ in Definition 32 and ORAM functionality $\mathcal{F}_{\mathsf{RAM}}$ in Definition C1.

3.2 STE Security Definition

We present the standard STE simulation-based adaptive security [49], [53] in our style. The adversary can adaptively choose the data structure (with data), adaptively make updates and queries, and is given the encrypted data structure. As common for works dealing with ORAM, we focus on the semi-honest setting, where the adversary follows the protocol. This models many practical settings where the attacker does not fully corrupt the system but still can influence and observe the communication. We leave the treatment of the stronger malicious security for future work. As usual, we use an abstract leakage profile $\mathcal{L}$ that specifies the information that the protocol leaks, which can be accessed by the simulator in the security definition. We say that an STE scheme is adaptively $\mathcal{L}$-secure if no efficient adversary can distinguish the transcripts from the honest execution of the protocol in the real world and the simulated ones in the ideal world with non-negligible probability. This security notion captures that all efficient honest-but-curious adversaries cannot learn more than that the leakage profile specifies. We will aim at constructions with minimal leakage.

Definition 31 (Adaptive Security for STE)**.** *Given label space $\mathbb{L}$, value space $\mathbb{V}$, and operation type space $\mathbb{O}$, let Π be a STE protocol with functionality $\mathcal{F}$. Let $\mathcal{L}_\Pi = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Access}})$ be the leakage profile describing leakage of Π's algorithms. Let $\kappa \in \mathbb{N}$ be the security parameter. Consider the probabilistic experiments defined in Figure 1.*

[3] Even though STE considers arbitrary data structures, we only focus on those of this type.

$\underline{\mathbf{Real}_{\Pi,\mathcal{A},q}(1^\kappa, z):}$

1: Given $1^\kappa, z$, adversary $\mathcal{A}$ outputs a data structure $\mathsf{ds}_1 \subseteq \mathbb{L} \times \mathbb{V}$.
2: $(K, \mathsf{EDS}_1) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, z, \mathsf{ds}_1)$.
3: $\mathcal{A}$ is given EDS_1.
4: $\mathcal{A}$ adaptively makes q queries in any order,
5: **for** all $t \in [q]$ **do**
6: $\quad \mathsf{Access}(\mathsf{op}_t, \ell_t, v_t)$:
7: $\quad$ where $\mathsf{op}_t \in \mathbb{O}, \ell_t \in \mathbb{L}, v_t \in \mathbb{V}$,
8: $\quad$ returns to $\mathcal{A}$ the transcript of honest
9: $\quad$ execution
10: $\quad [\mathsf{Access}_{\mathbf{C}}(K, \mathsf{op}_t, \ell_t, v_t), \mathsf{Access}_{\mathbf{S}}(\mathsf{EDS}_t)]$,
11: $\quad$ and server $\mathbf{S}$'s output EDS_{t+1}.
12: **end for**
13: Finally, $\mathcal{A}$ outputs bit b.
14: The experiment returns the same bit b.

$\underline{\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{F},q}(1^\kappa, z):}$

1: Given $1^\kappa, z$, adversary $\mathcal{A}$ outputs a data structure $\mathsf{ds}_1 \subseteq \mathbb{L} \times \mathbb{V}$.
2: Given $1^\kappa, z, \mathcal{L}_{\mathsf{Setup}}(\mathsf{ds}_1)$, simulator $\mathcal{S}$ outputs encrypted data structure EDS_1 and sends it to $\mathcal{A}$.
3: Let $\mathbf{S}$ be an honest server and given EDS_1.
4: $\mathcal{A}$ adaptively makes q queries in any order,
5: **for** all $t \in [q]$ **do**
6: $\quad \mathsf{Access}(\mathsf{op}_t, \ell_t, v_t)$:
7: $\quad$ where $\mathsf{op}_t \in \mathbb{O}, \ell_t \in \mathbb{L}, v_t \in \mathbb{V}$,
8: $\quad$ returns to $\mathcal{A}$ the transcript of
9: $\quad [\mathcal{S}(\mathcal{L}_{\mathsf{Access}}(\mathsf{ds}_t, \mathsf{op}_t, \ell_t, v_t)), \mathsf{Access}_{\mathbf{S}}(\mathsf{EDS}_t)]$,
10: $\quad$ and server $\mathbf{S}$'s output EDS_{t+1};
11: $\quad \mathsf{ds}_{t+1} \leftarrow \mathcal{F}(\mathsf{ds}_t, \mathsf{op}_t, \ell_t, v_t)$.
12: **end for**
13: Finally, $\mathcal{A}$ outputs bit b.
14: The experiment returns the same bit b.

Fig. 1: Experiments for Defining STE Adaptive Security.

We say that Π is adaptively $\mathcal{L}_\Pi$-secure if for all $z \in \{0,1\}^*$, there exists a PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$, all q which is a polynomial (in κ), the following is negligible (in κ):

$$\left|\Pr\left[\mathbf{Real}_{\Pi,\mathcal{A},q}(1^\kappa, z) = 1\right] - \Pr\left[\mathbf{Ideal}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{F},q}(1^\kappa, z) = 1\right]\right|.$$

Remark 2 (On Forward and Backward Security). The forward and backward security notions [54] are used to categorize the search and update leakage. Since we propose constructions with very strong security where the leakage is basically absent (two instantiations achieve empty access leakage and two leak at most the operation type), they will trivially satisfy the forward and strongest (BP-I) backward security. Therefore, we skip the formal definitions of forward and backward security here.

Remark 3 (On Randomized Functionality $\mathcal{F}$). Since we only cover data structures that support deterministic functionality, the simplified security definition suffices — the client's output is not given to the adversary as it can compute the correct output itself.

3.3 EMM and EDX Functionality

In Section 3.1, we provided a general STE syntax that covers our encrypted multi-map protocol and its two building blocks — encrypted dictionary and ORAM. Here we complete the definitions by specifying the functionality for each case.

Reactive Functionality. Following the ORAM literature, we say that the functionality is *reactive* if it holds an internal state between the executions, and we assume the functionality discussed throughout this work is all reactive unless specified otherwise. In standard computational RAM model of word size $w \in \mathbb{N}$ (in bits), we assume an (upper-bound) N_{ram} number of memory blocks to store the data structure with block size B_{ram} in bits.

EMM and EDX Functionality. We formally describe the basic operations associated with the data structure that permits accessing, adding and deleting the data. We define the reactive functionality $\mathcal{F}_{\mathsf{DS}}$ and abuse the notations by changing the subscript to indicate whether it is used for $\mathcal{F}_{\mathsf{DX}}$ (dictionary) or $\mathcal{F}_{\mathsf{MM}}$ (multi-map), respectively. In both cases, we set label space $\mathbb{L} = \{0,1\}^*$; set value space $\mathbb{V} = \{0,1\}^*$ for dictionary and $\mathbb{V} = \{\{0,1\}^*\}^*$ for multi-map. We use $*$ to simplify the notations, whereas the length restrictions are usually enforced in practice.

Definition 32 ($\mathcal{F}_{\mathsf{DS}}$)**.** *Given label space* $\mathbb{L}$*, value space* $\mathbb{V}$*, and operation type space* $\mathbb{O} = \{\mathsf{Get}, \mathsf{Put}, \mathsf{Remove}\}$*, we define reactive functionality* $\mathcal{F}_{\mathsf{DS}}$*, for every data structure* $\mathsf{ds} \subseteq \mathbb{L} \times \mathbb{V}$*,* $\mathcal{F}_{\mathsf{DS}}(\mathsf{ds}, \mathsf{op}, \ell, v)$*: where data structure* ds *is stored in the state,* $\mathsf{op} \in \mathbb{O}$*,* $\ell \in \mathbb{L}$*, and* $v \in \mathbb{V}$*,*

1. *If* $\mathsf{op} = \mathsf{Get}$*, if* ℓ *is in* ds *then* $v^* \leftarrow \mathsf{ds}[\ell]$*; otherwise* $v^* \leftarrow \perp$*.*
2. *If* $\mathsf{op} = \mathsf{Put}$*, if* ℓ *is in* ds *then* $v^* \leftarrow \mathsf{ds}[\ell]$*,* $\mathsf{ds}[\ell] \leftarrow v^* \cup v$ [4]*; otherwise* $\mathsf{ds}[\ell] \leftarrow v$*.*
3. *If* $\mathsf{op} = \mathsf{Remove}$*,*
   - *if* ℓ *is in* ds *then*
     $v^* \leftarrow \mathsf{ds}[\ell]$*; if* $v = \perp$ *then* $\mathsf{ds}[\ell] \leftarrow \perp$*;*
     otherwise, $\mathsf{ds}[\ell] \leftarrow v^* \setminus v$*.*
   - *otherwise,* $v^* \leftarrow \perp$*.*
4. *Output* (v^*, ds)*.*

Remark 4. For Get operation in Definition 32, input v is set to $\perp$. For Remove operation, we abuse the notation for the dictionary case as we can interpret the dictionary's value field as a set that only contains single element. We may write $\mathcal{F}_{\mathsf{MM}}$ and $\mathcal{F}_{\mathsf{DX}}$ defined the same as in Definition 32 to indicate the data structure is either a multi-map or a dictionary.

Remark 5. We note that [55] uses a multi-map functionality definition where Put directly overwrites existing values; [35] defines functionality on vectors, introducing *edit* and *append* operations. We instead focus on standard multi-map functionality where each label is associated with a set. Our set operation naturally accommodates atomic operation — operating on a single value can be interpreted as operating on a set with one single element.

3.4 Non-recursive Position-based ORAM

Besides the encrypted dictionary mentioned above, non-recursive position-based ORAM is a crucial building block of our construction. Our definition is new and captures the general non-recursive position-based ORAM. We start with defining a position map and position tag as follows.

Position Tag and Map. We use *position tag* to refer to a set of physical addresses associated with a block id, for example, the position tag of the i-th block is denoted by $\mathsf{pos}_i = \{addr_j\}_{j \in [S]}$, where S is some upper-bound size. For position-based ORAM managing N_{ram} blocks, we define the *position map* as $\mathsf{pmap} = \{(i, \mathsf{pos}_i)\}_{i \in [N_{\mathsf{ram}}]}$.

We refer to Appendix C.1 for syntax, correctness, and security definitions. In particular, we adapt the STE syntax, write the position map explicitly, and incorporate $\mathsf{ORAM.GenPosTag}(1^\kappa, M)$ algorithm, which takes as input security parameter 1^κ, total number of blocks M, and outputs some random position tag.

4 Generic Protocol EMM

Before we present our generic protocol, we note that directly adding (full) volume-hiding to SEAL or other ORAM-based SSE/STE schemes by padding the number of ORAM accesses to the tuple length leads to prohibitive roundtrip latency. For example, suppose the tuple length is 10000 (i.e., number of documents associated with each keyword), and with a batch size equal to 10 for ORAM access (client's stash size grows linearly in the batch size $\times \lceil \log_2 M^* \rceil$), for every query, SEAL will yield 1000 rounds of communication, incurring 30s round latency.

So a different approach is needed, and we propose to pack the document identifiers into a single block, as padding the number of ORAM accesses to the tuple length incurs at least the tuple length of rounds of communication, while costs equal or more bandwidth. We are now ready to present our protocol.

[4] Use $\mathsf{ds}[\ell] \leftarrow v$ for dictionary.

4.1 EMM Protocol Overview

We build our generic encrypted multi-map protocol EMM using an encrypted dictionary EDX that hides the query, access, and volume patterns, and a non-recursive position-based ORAM (with encrypted blocks). We consider the setting where the client keeps a small state, avoiding using storage that is linear in either the total number of labels or values in the multi-map, so that it can be easily adapted to the multi-client or multi-device setting.

As mentioned in the Introduction, our encrypted multi-map protocol allocates an ORAM block storing the associated values for each label in the multi-map. (In our application it will be storing document ids for each keyword.) This is a common straightforward approach. But to achieve practical efficiency, we employ some novel ideas. To help understand our approach, we first briefly recall how position-based ORAM operates.

Position-based ORAM Overview. The client first shuffles the memory blocks and keeps secret the position map — the mappings between the block ids and the position tags (i.e., sets of physical addresses, see Section 3.4). For each ORAM operation (Read or Write), the client will retrieve the blocks, decrypt, re-encrypt, and write back to a set of new random physical addresses and update the local position map. A secure non-recursive position-based ORAM guarantees computational obliviousness — by observing the memory accesses, no efficient adversary can glean information about the underlying blocks that the client accesses. A drawback of non-recursive ORAM is that the position map grows linearly in the total number of memory blocks and may become infeasible to fit the client's local storage. Usually, a recursion technique is proposed — saving the client's storage by recursively storing the position maps in a series of ORAMs so that only the first level's (smaller) ORAM's position map is stored locally by the client, while the last level's ORAM stores the data blocks. The number of levels of recursion is bounded by $\mathcal{O}(\log N_{\mathsf{ram}})$ for managing N_{ram} blocks, which implies a multiplicative factor increase in round complexity and costs more bandwidth compared with the non-recursive one. Thus it is desirable to use a non-recursive position-based ORAM if possible, but how to deal with the linear position map storage if N_{ram} is a significantly large number?

Ideas for Optimization. Our idea is to store each label's associated block id and position tag in an encrypted dictionary on the server side. Since position-based ORAM requires updating the position tag for each block access, we propose novel optimization techniques to enable encrypted dictionary handling the update without additional costs on top of the label lookup.

Protocol Overview. We now describe our generic protocol: given a multi-map and its maximum size, [5] for each label ℓ in the multi-map, we allocate an ORAM block indexed by bid_ℓ to store its values. Then in the encrypted dictionary EDX, at the label ℓ's associated value field, we store the block identifier bid_ℓ and its corresponding position tag $\mathsf{pos}_{\mathsf{bid}_\ell}$.

Performing multi-map operations using EMM is straightforward: for Get operation, we first find in the encrypted dictionary label ℓ's associated block id bid_ℓ and position tag $\mathsf{pos}_{\mathsf{bid}_\ell}$; then read ORAM's bid_ℓ-th block to obtain its associated values with $\mathsf{pos}_{\mathsf{bid}_\ell}$. For Put and Remove operations, our generic EMM protocol supports both atomic operation — updating a label and a single value, and set operation — updating a label and a set of values. The set operation incurs almost negligible computational and communicational overheads compared with the atomic one as we store the label's associated values in a single ORAM block. For each multi-map operation, we only require one EDX operation, followed by two ORAM operations (Read and Write) and one update to EDX refreshing the position tag (and possibly the block id). The volume-hiding property inherits from that both label and value stored in the encrypted dictionary are fixed in size. A significant advantage of our scheme is we use non-recursive position-based ORAM, which improves the efficiency compared with the recursive cases, and at the same time does not sacrifice security as Horus does, applying PRF to labels to generate position tags [56], as we store the position tags in the encrypted dictionary, which can be easily accessed.

[5] For simplicity, we consider the maximum size is fixed after setup.

$\underline{\mathsf{Setup}(1^\kappa, M^* \| B_{\mathsf{ram}}, \mathsf{mm}) :}$

1: $\mathsf{BidStack.Init}(\lfloor \log_2 M^* \rfloor)$
2: $\mathsf{array} \leftarrow ()$
3: **for** each $(\ell_i, \mathtt{vset}_i)$ in mm **do**
4: $\quad \mathsf{block}_i \leftarrow i \| \mathtt{vset}_i$
5: $\quad \mathsf{array}[i] \leftarrow \mathsf{block}_i$
6: $\quad \mathsf{BidStack.IncCtr}()$
7: **end for**
8: $(K_{\mathsf{oram}} \| \mathsf{pmap}, \mathsf{oarray}) \xleftarrow{\$} \mathsf{ORAM.Setup}(1^\kappa,$
9: $\qquad\qquad\qquad\qquad M \| B_{\mathsf{ram}}, \mathsf{array})$
10: $\mathsf{dx} \leftarrow \{\}$
11: **for** each $(\ell_i, \mathtt{vset}_i)$ in mm **do**
12: $\quad \mathsf{dx} \leftarrow \mathsf{dx} \cup \{(\ell_i, (i, \mathsf{pmap}[i]))\}$
13: **end for**
14: $(K_{\mathsf{edx}}, \mathsf{edx}) \xleftarrow{\$} \mathsf{EDX.Setup}(1^\kappa, M, \mathsf{dx})$
15: $K \leftarrow (K_{\mathsf{edx}}, K_{\mathsf{oram}})$
16: $\mathsf{emm} \leftarrow (\mathsf{edx}, \mathsf{oarray})$
17: **return** (K, emm)

$\underline{\mathsf{Dec}(K, \mathsf{emm}) :}$

1: Parse emm as $(\mathsf{edx}, \mathsf{oarray})$
2: Parse K as $(K_{\mathsf{edx}}, K_{\mathsf{oram}})$
3: $\mathsf{dx} \leftarrow \mathsf{EDX.Dec}(K_{\mathsf{edx}}, \mathsf{edx})$
4: $\mathbf{R} \leftarrow ()$
5: $\mathsf{array} \leftarrow \mathsf{ORAM.Dec}(K_{\mathsf{oram}}, \mathsf{oarray})$
6: **for** each block in array **do**
7: $\quad$ **if** $\mathsf{block} \neq \perp$ **then**
8: $\quad\quad$ Parse block as $(i, \mathtt{vset}_i)$
9: $\quad\quad \mathbf{R}[i] \leftarrow \mathtt{vset}_i$
10: $\quad$ **end if**
11: **end for**
12: $\mathsf{mm} \leftarrow \{\}$
13: **for** each $(\ell_i, (i, \mathsf{pmap}[i]))$ in dx **do**
14: $\quad \mathtt{vset} \leftarrow \mathbf{R}[i]$
15: $\quad \mathsf{mm} \leftarrow \mathsf{mm} \cup \{(\ell_i, \mathtt{vset}_i)\}$
16: **end for**
17: **return** mm

Fig. 2: Encrypted Multi-Map Protocol EMM (Part 1).

We optimize our generic EMM protocol by removing the update procedure to EDX after the ORAM accesses and enabling both Read and Write in a single ORAM operation. We achieve the optimization by extending the standard functionality of EDX and ORAM. Specifically, we extend EDX functionality to include a single operation for both Get and Put. This improvement is significant, especially for EDX instantiations suppressing query-equality and access patterns based on oblivious maps, where GetUp only costs the same bandwidth and rounds of communication as Get. Similarly, we extend the functionality of ORAM to combine Read and Write as a single operation.

On Recursive Position-based ORAMs. We also discuss in Appendix D on why our generic protocol EMM and its instantiations do not yield a position-based ORAM that is asymptotically better in efficiency than the existing recursive position-based ORAMs.

4.2 Extended Functionality

Due to the space limit, we present the extended functionality for encrypted dictionary and ORAM we talked about in the previous subsection in Appendix B1 and Appendix C2, respectively.

4.3 Algorithms of EMM Protocol

We are now ready to specify our generic encrypted multi-map protocol EMM. We list its algorithms in Figure 2, 7, 3, and 4. In the generic protocol, we refer to abstract label space $\mathbb{L}$ and value space $\mathbb{V}$ for generality. To use our EMM as part of SSE, we can let the label space be the keyword space, $\mathbb{L} = \{0,1\}^{\mathtt{lenW}}$ for some maximum keyword length $\mathtt{lenW}$; let the value space be the space of sets of document identifiers, $\mathbb{V} = \{\{0,1\}^{\mathtt{lenID}}\}^*$, where $\mathtt{lenID}$ is the maximum length of the document identifier.

The Setup algorithm takes as input security parameter κ, auxiliary information $M^* \| B_{\mathsf{ram}}$, and multi-map mm over $\mathbb{L} \times \mathbb{V}$. M^* is the upper-bound on the number of labels. B_{ram}, the RAM block size, has to be sufficient to store one block id and an upper-bound number of values associated to each label (Section E.1 discusses resizability and fixing maximum size in detail).

One challenge for dynamic updates is to maintain a list of free block ids, which relates to both the Remove operation and the Get operation. ORAM-based SSE/STE schemes usually store, for each

$\underline{\mathsf{Access}_{\mathbf{C},\kappa}(K, \mathsf{op}, \ell, \mathtt{vset}) :}$
1: Parse K as $(K_{\mathsf{edx}}, K_{\mathsf{oram}})$
2: $\mathsf{pos}^* \xleftarrow{\$} \mathsf{ORAM.GenPosTag}(1^\kappa, M)$
3: **if** $\mathsf{op} = \mathsf{Get}$ **then**
4: $\quad \mathsf{op}_1 \leftarrow \mathsf{GetUp}$; $\mathsf{data}_1 \leftarrow \perp \| \mathsf{pos}^*$
5: **else if** $\mathsf{op} = \mathsf{Put}$ **then**
6: $\quad \mathsf{bid} \leftarrow \mathsf{BidStack.GetBid}()$
7: $\quad \mathsf{op}_1 \leftarrow \mathsf{GetUp}$; $\mathsf{data}_1 \leftarrow \mathsf{bid} \| \mathsf{pos}^*$
8: **else** ▷ Condition $\mathsf{op} = \mathsf{Remove}$
9: $\quad$ **if** $\mathtt{vset} \neq \perp$ **then**
10: $\quad\quad \mathsf{op}_1 \leftarrow \mathsf{GetUp}$; $\mathsf{data}_1 \leftarrow \perp \| \mathsf{pos}^*$
11: $\quad$ **else**
12: $\quad\quad \mathsf{op}_1 \leftarrow \mathsf{Remove}$; $\mathsf{data}_1 \leftarrow \perp$
13: $\quad$ **end if**
14: **end if**
15: Run $\mathsf{EDX.Access}_{\mathbf{C}}(K_{\mathsf{edx}}, \mathsf{op}_1, \ell, \mathsf{data}_1)$ with $\mathbf{S}$ and get out_1

$\underline{\mathsf{Access}_{\mathbf{S}}(\mathsf{emm}) :}$
1: Parse emm as $(\mathsf{edx}, \mathsf{oarray})$
15: Run $\mathsf{EDX.Access}_{\mathbf{S}}(\mathsf{edx})$ with $\mathbf{C}$ and get edx^*

Fig. 3: Encrypted Multi-Map Protocol EMM (Part 2).

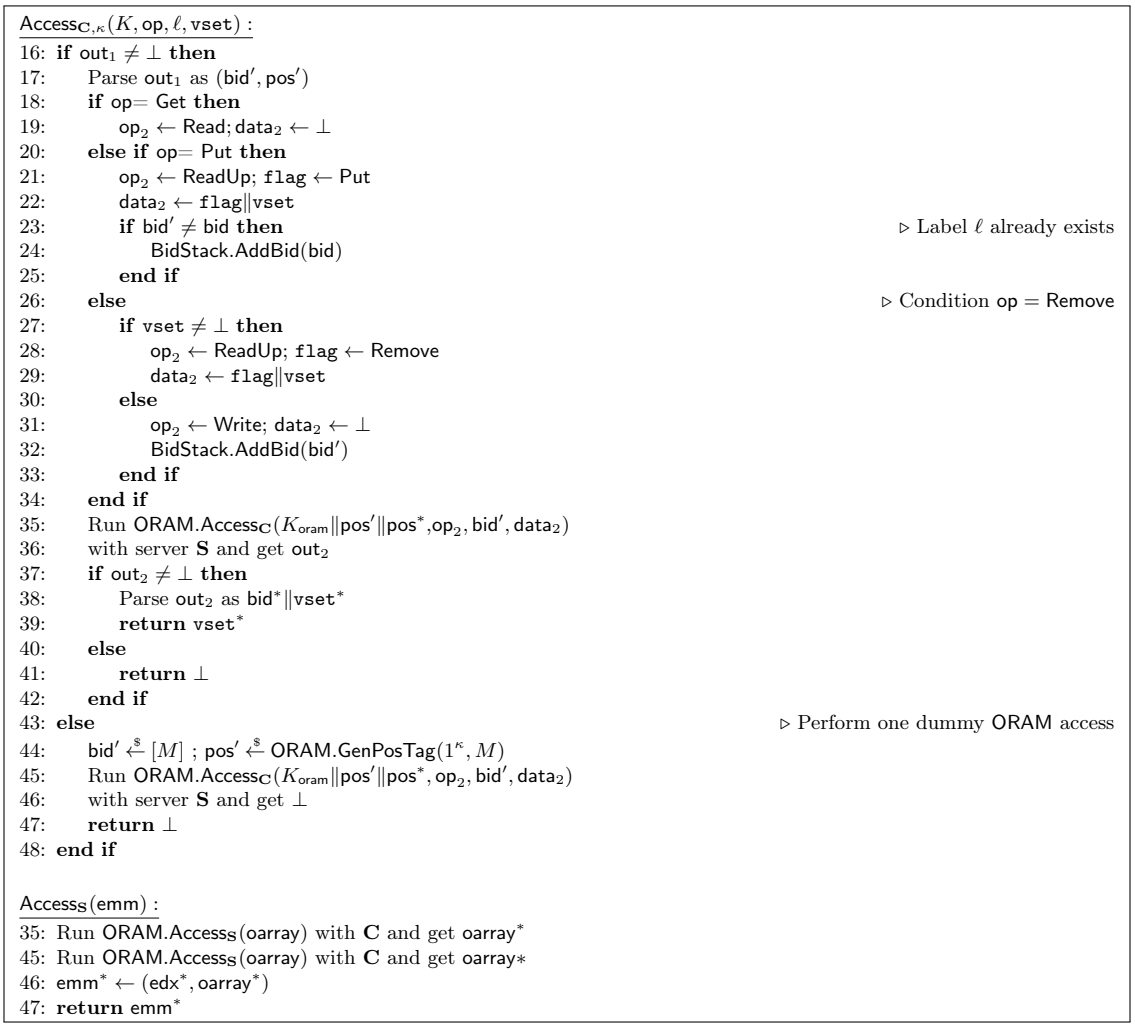
keyword, a set of deleted document produce a set of inserted document identifiers using 1 to `ctr`; and existing document identifiers will be the set difference between the two.

Since EDX and ORAM have the same number of blocks, we directly use the one-to-one mapping between the two, so that we only need to manage the block ids of EDX and use the same block id in ORAM. We manage the block ids using BidStack in Figure 7 (Appendix E) which always keeps the last element in the stack as the global counter, and stores the free block id if a block (i.e., keyword) has been freed. We set the initial stack size to be $\lceil \log_2(M^*) \rceil$ (passed as a parameter depending on an estimate of the frequency of the remove operations). Under the constraint of having as little local storage as possible, as the stack keeping the counter and also the deletion records grows linearly in the number of blocks (number of keywords), the size increase reveals information about the deletion record, BidStack can be instantiated using an oblivious stack and store it together with EDX and ORAM. For operation-type-hiding EMM schemes, the stack will be accessed every time; for operation-type-revealing EMM schemes, the stack will be accessed only for Put and Remove. BidStack can be accessed together with EDX to avoid additional roundtrips.

5 Protocol Instantiations

We present two instantiations of our generic protocol EMM, denoted by $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$ and their corresponding operation-type-revealing variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$. All protocols instantiate the non-recursive ORAM with non-recursive PathORAM [50] for simplicity but differ in the instantiations of the encrypted dictionary. We briefly recall PathORAM (Appendix C.3) and show how it efficiently supports ORAM's extended functionality (Definition C2). We then discuss the encrypted dictionary instantiations — oblivious AVL tree with encryption (OAvlTreeE), oblivious BSkiplist with encryption (OBSkiplistE), and their operation-type-revealing variants AvlTreeE and BSkiplistE, and show how they efficiently support the extended functionality for the encrypted dictionary.

Encrypted Dictionary Instantiations. For our encrypted dictionary we pick two constructions from [57], [58] that were more formally defined in [48]: oblivious AVL-tree with encryption (OAvlTreeE) and oblivious BSkiplist with encryption (OBSkiplistE). In [48] the constructions were formalized as oblivious maps with encryption OMapE. OMapE security is defined using the same STE security



$\underline{\mathsf{Access}_{\mathbf{C},\kappa}(K, \mathsf{op}, \ell, \mathtt{vset}) :}$

16: **if** $\mathsf{out}_1 \neq \perp$ **then**
17: Parse out_1 as $(\mathsf{bid}', \mathsf{pos}')$
18: **if** $\mathsf{op} = \mathsf{Get}$ **then**
19: $\mathsf{op}_2 \leftarrow \mathsf{Read}; \mathsf{data}_2 \leftarrow \perp$
20: **else if** $\mathsf{op} = \mathsf{Put}$ **then**
21: $\mathsf{op}_2 \leftarrow \mathsf{ReadUp}$; $\mathtt{flag} \leftarrow \mathsf{Put}$
22: $\mathsf{data}_2 \leftarrow \mathtt{flag} \| \mathtt{vset}$
23: **if** $\mathsf{bid}' \neq \mathsf{bid}$ **then** ▷ Label ℓ already exists
24: BidStack.AddBid(bid)
25: **end if**
26: **else** ▷ Condition $\mathsf{op} = \mathsf{Remove}$
27: **if** $\mathtt{vset} \neq \perp$ **then**
28: $\mathsf{op}_2 \leftarrow \mathsf{ReadUp}$; $\mathtt{flag} \leftarrow \mathsf{Remove}$
29: $\mathsf{data}_2 \leftarrow \mathtt{flag} \| \mathtt{vset}$
30: **else**
31: $\mathsf{op}_2 \leftarrow \mathsf{Write}$; $\mathsf{data}_2 \leftarrow \perp$
32: BidStack.AddBid(bid')
33: **end if**
34: **end if**
35: Run $\mathsf{ORAM.Access}_{\mathbf{C}}(K_{\mathsf{oram}} \| \mathsf{pos}' \| \mathsf{pos}^*, \mathsf{op}_2, \mathsf{bid}', \mathsf{data}_2)$
36: with server $\mathbf{S}$ and get out_2
37: **if** $\mathsf{out}_2 \neq \perp$ **then**
38: Parse out_2 as $\mathsf{bid}^* \| \mathtt{vset}^*$
39: **return** $\mathtt{vset}^*$
40: **else**
41: **return** $\perp$
42: **end if**
43: **else** ▷ Perform one dummy ORAM access
44: $\mathsf{bid}' \xleftarrow{\$} [M]$; $\mathsf{pos}' \xleftarrow{\$} \mathsf{ORAM.GenPosTag}(1^\kappa, M)$
45: Run $\mathsf{ORAM.Access}_{\mathbf{C}}(K_{\mathsf{oram}} \| \mathsf{pos}' \| \mathsf{pos}^*, \mathsf{op}_2, \mathsf{bid}', \mathsf{data}_2)$
46: with server $\mathbf{S}$ and get $\perp$
47: **return** $\perp$
48: **end if**

$\underline{\mathsf{Access}_{\mathbf{S}}(\mathsf{emm}) :}$

35: Run $\mathsf{ORAM.Access}_{\mathbf{S}}(\mathsf{oarray})$ with $\mathbf{C}$ and get oarray^*
45: Run $\mathsf{ORAM.Access}_{\mathbf{S}}(\mathsf{oarray})$ with $\mathbf{C}$ and get $\mathsf{oarray}*$
46: $\mathsf{emm}^* \leftarrow (\mathsf{edx}^*, \mathsf{oarray}^*)$
47: **return** emm^*

Fig. 4: Encrypted Multi-Map Protocol EMM (Part 3).

(Definition 31) while ensuring the empty Access leakage. Hence we can treat OMapE as a class of encrypted dictionaries with empty access leakage — hiding the query, access, and volume patterns.

We now discuss how we use OAvlTreeE and OBSkiplistE to support efficient GetUp. Both OAvlTreeE and OBSkiplistE rely on underlying non-recursive position-based ORAM (e.g., non-recursive PathORAM) to manage the dictionary data, where each block stores a label and the associated value. They enable oblivious search by making $\mathcal{O}(\log M^*)$ accesses to the underlying ORAM of M^* blocks. By instantiating the underlying ORAM with PathORAM that supports ReadUp, OAvlTreeE and OBSkiplistE can easily support GetUp — when the ORAM block storing the label is found, instead of keeping the same value, re-encrypt, and writeback, we update the value then follow the standard procedure of OAvlTreeE and OBSkiplistE. Hence it improves efficiency compared with using only black-box dictionary operations Get then Put.

Since our $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$ directly instantiate the encrypted dictionary with OAvlTreeE, OBSkiplistE respectively [48], we refer the details to their work. Due to the lack of space, we move the discussion on

AvlTreeE and BSkiplistE to Appendix F.3. The essential idea is to remove the dummy ORAM accesses for Get (or GetUp) used originally in OAvlTreeE and OBSkiplistE for hiding the operation type.

6 Security Analysis

We state the security of the generic protocol EMM, instantiations $\mathsf{EMM}_{\mathrm{avl}}$, $\mathsf{EMM}_{\mathrm{bskip}}$ and their operation-type-revealing variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$.

Security of Generic Protocol EMM. The following theorem states the security guarantee our generic protocol provides. It implies that the protocol inherits the strong security of its building blocks.

Theorem 61 (Main Theorem)**.** *The generic encrypted multi-map protocol* EMM *with building blocks encrypted dictionary* EDX *and non-recursive position-based* ORAM*, is adaptively $\mathcal{L}_{\mathsf{EMM}}$-secure, if* EDX *is adaptively $\mathcal{L}_{\mathsf{EDX}}$-secure and* ORAM *is adaptively $\mathcal{L}_{\mathsf{ORAM}}$-secure (cf. Definition 31), where $\mathcal{L}_{\mathsf{EMM}} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Access}})$ with $\mathcal{L}_{\mathsf{Setup}} = (\mathcal{L}_{\mathsf{EDX.Setup}}, \mathcal{L}_{\mathsf{ORAM.Setup}})$, $\mathcal{L}_{\mathsf{Access}} = (\mathcal{L}_{\mathsf{EDX.Access}}, \mathcal{L}_{\mathsf{ORAM.Access}})$.*

Due to the space limit, we provide the proof sketch in Appendix F.2. It is certainly essential to make sure that our building blocks have strong security.

Security of $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$. Let M^* be the upper-bound on the maximum number of labels of the multi-map. Parameter `nodeSize` is the size of every node in bits; `height` is the height of the structure (i.e., AVL tree or BSkiplist); `bucketSize` is the size of every bucket in bits as part of its underlying ORAM; parameter β is the branching factor of OBSkiplistE's underlying BSkiplist.

Based on previous results [48], [59], we show in Appendix F.1 that PathORAM is adaptively $\mathcal{L}_{\mathsf{PathORAM}}$-secure (cf. Definition 31) assuming the blockcipher-based mode of operation used for encryption is IND-CPA with $\mathcal{L}_{\mathsf{PathORAM.Setup}} = (M^*, B_{\mathrm{poram}}, \mathtt{bucketSize}_{\mathrm{poram}})$ and $\mathcal{L}_{\mathsf{PathORAM.Access}} = \perp$, where B_{poram} is the block size in bits, and $\mathtt{bucketSize}_{\mathrm{poram}}$ is the bucket size in bits. Since PathORAM's Read and Write operations require reading and writing back the encrypted blocks, we show that this property provides optimized ReadUp without sacrificing security (Appendix F.3). All instantiations of encrypted dictionary OAvlTreeE, OBSkiplistE, and their variants AvlTreeE, BSkiplistE are built on PathORAM. The only security difference between OAvlTreeE and AvlTreeE (OBSkiplistE and BSkiplistE) is that the former hides the operation type, while the latter does not. By utilizing underlying PathORAM's optimized ReadUp, all encrypted dictionary instantiations support optimized GetUp, incurring no security loss (see justification in Appendix F.3). The following security statements hold as part of our instantiations: OAvlTreeE is adaptively $\mathcal{L}_{\mathsf{OAvlTreeE}}$-secure (cf. Definition 31) assuming the blockcipher-based mode of operation used for encryption is IND-CPA, with $\mathcal{L}_{\mathsf{OAvlTreeE}} = (\mathcal{L}_{\mathsf{OAvlTreeE.Setup}}, \mathcal{L}_{\mathsf{OAvlTreeE.Access}})$, where $\mathcal{L}_{\mathsf{OAvlTreeE.Setup}} = (M^*, \mathtt{nodeSize}, \mathtt{height}, \mathtt{bucketSize})$, $\mathcal{L}_{\mathsf{OAvlTreeE.Access}} = \perp$; OBSkiplistE is adaptively $\mathcal{L}_{\mathsf{OBSkiplistE}}$-secure (cf. Definition 31) assuming the blockcipher-based mode of operation used for encryption is IND-CPA with $\mathcal{L}_{\mathsf{OBSkiplistE}} = (\mathcal{L}_{\mathsf{OBSkiplistE.Setup}}, \mathcal{L}_{\mathsf{OBSkiplistE.Access}})$, where $\mathcal{L}_{\mathsf{OBSkiplistE.Setup}} = (M^*, \mathtt{nodeSize}, \mathtt{height}, \beta, \mathtt{bucketSize})$, $\mathcal{L}_{\mathsf{OBSkiplistE.Access}} = \perp$.

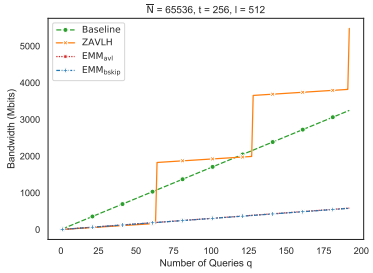
We summarize the security statements for our two instantiations $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$. The proofs follow from Theorem 61 and the above security statements.

Theorem 62. *The encrypted multi-map protocol $\mathsf{EMM}_{\mathrm{avl}}$ is adaptively $\mathcal{L}_{\mathsf{EMM}_{\mathrm{avl}}}$-secure (cf. Definition 31) assuming the blockcipher-based mode of operation used for encryption is IND-CPA, where $\mathcal{L}_{\mathsf{EMM}_{\mathrm{avl}}} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Access}})$ with $\mathcal{L}_{\mathsf{Setup}} = (M^*, \mathtt{nodeSize}, \mathtt{bucketSize}, \mathtt{height}, B_{\mathrm{poram}}, \mathtt{bucketSize}_{\mathrm{poram}})$, and $\mathcal{L}_{\mathsf{Access}} = \perp$.*

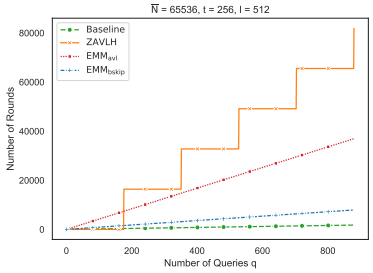
Theorem 63. *The encrypted multi-map protocol $\mathsf{EMM}_{\mathrm{bskip}}$ is adaptively $\mathcal{L}_{\mathsf{EMM}_{\mathrm{bskip}}}$-secure (cf. Definition 31) assuming the blockcipher-based mode of operation used for encryption is IND-CPA, where*

$\mathcal{L}_{\mathsf{EMM}_{\mathrm{bskip}}} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Access}})$ *with* $\mathcal{L}_{\mathsf{Setup}} = (M^*, \mathtt{nodeSize}, \mathtt{bucketSize}, \mathtt{height}, \beta, B_{\mathrm{poram}}, \mathtt{bucketSize}_{\mathrm{poram}})$, *and* $\mathcal{L}_{\mathsf{Access}} = \perp$.

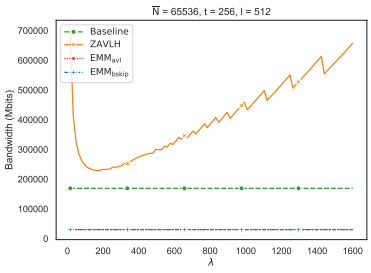
The variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$ have similar security statements as above but differ in the leakage profile — both instantiations leak the operation type in Access, namely, $\mathcal{L}_{\mathsf{Access}} = \mathsf{op} \in \{\mathsf{Get}, \mathsf{Put}, \mathsf{Remove}\}$. Due to the lack of space, we omit the formal security statements for the two operation-type-revealing variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$, and move the explanation that the access leakage contains at most the operation type to Appendix F.4. We remark that, as also mentioned in the Introduction, the setup leakage of all instantiations can be interpreted as the upper-bound size of multi-map, since other data-dependent parameters e.g., `height`, `nodeSize`, `bucketSize`, block size B, β already have been fixed by the upper-bound size and the degree of obliviousness as part of the security parameter.



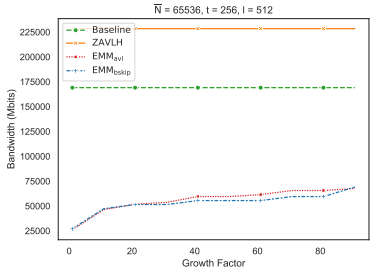
(a) Bandwidth Cost Comparison.



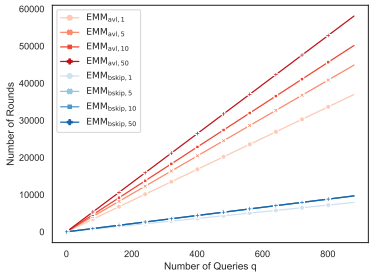
(b) Round Complexity Comparison.



(c) Bandwidth Cost for $q = 10000$.



(d) Bandwidth Cost for $q = 10000$.



(e) Hiding Operation Types.

Parameter	Value
h_{avl}	$1.45 \times \lceil \log_2(M^*+2) \rceil$
h_{bskip}	$\lceil \log_\beta M^* \rceil$
h_{poram}	$\lceil \log_2 M^* \rceil$

Scheme	# Rounds
$\mathsf{EMM}_{\mathrm{avl}}$	$r_1 = 3 \cdot q \cdot h_{\mathrm{avl}} + 3$
$\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$	$r_2 = (\lceil p \cdot q \rceil + 3 \cdot (q - \lceil p \cdot q \rceil)) \cdot h_{\mathrm{avl}} + 3$
$\mathsf{EMM}_{\mathrm{bskip}}$	$r_3 = 2 \cdot q \cdot h_{\mathrm{bskip}} + 3$
$\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$	$r_4 = (\lceil p \cdot q \rceil + 2 \cdot (q - \lceil p \cdot q \rceil)) \cdot h_{\mathrm{bskip}} + 3$

	N	k	l
Enron	30109	15	17303
Lucene	66491	16	49758

Block Size (bytes)	AvlTreeE	BSkiplistE	PathORAM
Enron	64	585	5649
Lucene	64	585	12471

(f) Parameters and Expressions.

Fig. 5: Simulation Results and Parameters.

7 Complexity Analysis

Before analyzing the complexity of our protocols, we first establish the notations and conventions. For any multi-map mm and its arbitrary multi-map mm^* of some upper-bound size, we write $\mathbb{L}_{\mathsf{mm}^*}, \mathbb{V}_{\mathsf{mm}^*}$ to denote the label and value space fixed by mm^*. Let $M^* = \#\mathbb{L}_{\mathsf{mm}^*}$ be the upper-bound size on the total number of labels; let l_{mm^*} be the maximum tuple length, namely, the maximum number of values associated with each label. For round complexity, we use the convention — the number of *rounds* equals the number of interactions between the client and server, where one single round involves sending a message and receiving a message. All our instantiations $\mathsf{EMM}_{\mathrm{avl}}, \mathsf{EMM}_{\mathrm{bskip}}, \mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}, \mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$ have computational and communicational complexity $\mathcal{O}(\log^2 M^* + \log M^* \cdot l_{\mathsf{mm}^*} \cdot \max_{v \in \mathbb{V}_{\mathsf{mm}^*}} |v|_2)$, and round complexity $\mathcal{O}(\log M^*)$. We also provide their concrete expressions in Figure 5f.

Complexity Comparison with [49] and Baseline [52]. We have mentioned that [49] and [52]'s baseline are the only two work proposing EMM constructions that hides query, access, and volume patterns. As [49] does not provide any implementation, we present complexity analysis of their most efficient instantiation ZAVLH using the complexity expressions provided in their full paper, with ours and baseline. In particular, we focus on the concrete complexity comparison and show from the simulation results that our instantiations perform significantly better than theirs in communicational and round complexity, and better than baseline in communicational complexity. We also provide an asymptotic complexity comparison with [49] in Appendix G.2, where we show under the same assumptions as [49], though our instantiations cannot outperform ZAVLH asymptotically in bandwidth complexity, they always have better asymptotic round complexity since ZAVLH's rebuild suffers from high round complexity due to oblivious shuffling.

Before moving to discuss the simulation results, we continue describing the parameters used in the simulation. For arbitrary database represented using a multi-map mm, let N denote the number of documents (i.e., values) in the database, let $\overline{N} = \sum_{\ell \in \mathbb{L}_{\mathsf{mm}}} \#\mathsf{mm}[\ell]$ ($\overline{N}$ is written as N in [49]). Let t denote the number of keywords (i.e., labels), l denote the maximum tuple length, g denote the *growth factor* ($g > 1$) — a parameter that determines the upper-bound size of the multi-map. Specifically, for arbitrary multi-map with $\overline{N}, t, l$ given in the setup, its upper-bound size has $t^* = g \cdot t, l^* = g \cdot l, \overline{N}^* = g^2 \cdot \overline{N}$ (choosing g^2 to compute $\overline{N}^*$ is due to that $\overline{N}$ represents the total number of all keywords' associated documents). In the following experiments, if it is not explicitly specified, g is fixed at 1.5. For an arbitrary sequence of query operations, we assume 10% are Put, 10% are Remove, and the rest 80% are Get. Figure 5f shows the expressions we use for simulation with $p = 80\%$ indicates the percentage of the operations that are Get. Due to the space limit, we continue describing the parameters and expressions used to generate the plots in Appendix G.1.

We are now ready to present our simulation results. Same as in [49], we use $\overline{N} = 65536, t = 256, l = 512$ as default parameters. We also investigate λ's (for every λ operations ZAVLH needs to run one rebuild) influence shown in Figure 5c: for default database, varying λ in between 1 and 1600, and running $10,000$ operations to ensure that at least one rebuild occurs. We found that increasing λ does not always lead to performance improvement of ZAVLH. We use the optimal $\lambda = 176$ we found through the experiment in the rest of the experiments ([49] sets $\lambda = 64$ as default). Both Figures 5a and 5b show that ZAVLH follows a trend with a step-size increase in bandwidth and rounds, reflecting the heavy overhead introduced by the rebuild — a necessary procedure to suppress the leakage in their framework. Unfortunately, it can be seen from the figure that after around 550 queries, the accumulated bandwidth cost of ZAVLH is even greater than the baseline — downloading and updating the encrypted database for every query. Throughout the simulation experiments, we found that the maximum tuple length l plays a more significant role in efficiency than $\overline{N}$. Though the suggested l is relatively small compared with $\overline{N}$ (and possibly N), we show that our instantiations are still practical even when l is close to the total number of documents N in Implementations (Sec 8). We also analyze how the growth factor, namely, the increase in the multi-map's upper-bound size impacts the performance. The multi-map's upper-bound size does not affect the baseline and the ZAVLH's performance for a fixed sequence of operations but affects our instantiations as the underlying ORAM usually occupies non-resizable pre-allocated storage. However, Figure 5d shows that for a database of up to 100 times increase in size, even after $10,000$ operations, the less efficient variants $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$ still perform much better than the baseline, whereas ZAVLH performs the worst.

To offer guidance on selecting the instantiations for practice, Figure 5e shows the round complexity results on a database of the size that is an element in $\{1, 5, 10, 50\}$ times the default size accordingly. We found AVL-tree-based instantiations incur more rounds than BSkiplist-based ones and are more sensitive to the size increase in the database — overlapped blue lines show that the round increase is less sensitive to the size increase. We expand the discussion and provide more figures on bandwidth costs in Appendix G.1.

8 Implementations and Experiments

Implementation. We built our instantiations $\mathsf{EMM}_{\text{avl}}$ and $\mathsf{EMM}_{\text{bskip}}$ and their operation-type-revealing variants $\mathsf{EMM}^{\text{r}}_{\text{avl}}$ and $\mathsf{EMM}^{\text{r}}_{\text{bskip}}$ [6] in C++ using OAvlTreeE and OBSkiplistE's implementations in [48]. Our implementations manage the dataset using RocksDB (optimized for SSD), which suits the typical setting that the storage is more affordable than the memory. Although storing all the data directly in the memory removes the loading time of the SSD-based database, we found for database of at least moderate size, the significant query latency overhead comes from the rounds of communication and the bandwidth cost compared with the computational cost — a major concern in most of the ORAM-based solutions.

Experimental Setup. We conducted our experiments on two public available email databases Enron and Lucene, where each email is treated as a single document, of size 30109 and 66491. We use their processed version, fixing the keyword space to a set of the most frequently-used 3000 keywords, provided by Oya and Kerschbaum [60], [61].

Since our instantiations achieve the volume-hiding property through padding each tuple to the largest size, this specific keyword space captures the worst scenario for evaluation. As in the complexity analysis, we store document identifiers in each PathORAM block using the cheaper storage representation of the two: either a list of l document identifiers where each is a 32-bit integer or a bitvector of N-bits where all 1's indicate the matched document identifiers in the database. Still, we show that our instantiations are practical in this setting. Our experimental results were generated on a commodity laptop with 12-core Intel(R) Core i7-8750H CPU @ 2.20GHz, 16GB RAM, and 360GB SSD, running Ubuntu 20.04.

		$\mathsf{EMM}_{\text{avl}}$	$\mathsf{EMM}^{\text{r}}_{\text{avl}}$	$\mathsf{EMM}_{\text{bskip}}$	$\mathsf{EMM}^{\text{r}}_{\text{bskip}}$
1	# Rounds	70	35	14	10
	Bandwidth (kBs)	885.07	751.65	1623.72	1227.62
	Time (ms)	2170.81	1110.13	549.90	398.20
2	# Rounds	76	38	14	10
	Bandwidth (kBs)	1671.78	1526.93	2387.56	1991.46
	Time (ms)	2413.74	1262.15	611.00	459.32

Table 1: Average Bandwidth and Communicational Cost Comparison for Enron (1) and Lucene (2).

In all EMM instantiations, we use the parameters in Figure 5f, where N denotes the total number of documents, k denotes the maximum keyword length in bytes, and l denotes the maximum tuple length, namely, the maximum number of emails associated with every single keyword. We also fix the growth factor to 1.5. All AVL-tree-based and BSkiplist-based instantiations took less than 30 minutes to set up either Enron or Lucene. Same to the concrete complexity analysis, in our experiments, we assume 80% operations are Get, 10% are Put, and the rest 10% are Remove. We generated the results by running 1000 operations on each dataset, performing Get and Remove operations on keywords and associated document identifiers selected randomly from the corresponding dataset. We also ensure that Remove is performed first and Get after, canceling each other out so that after 1000 operations, the dataset stays the same as the setup one. The average computational time for each operation among all instantiations is below 300 ms. For the more concerned communicational overhead, we simulated the communicational time under network condition with 100 mbps for either uploading or downloading and 30 ms in round latency. Table 1 shows that the communicational overhead is still reasonable for practice among all instantiations. Although the bandwidth costs are comparable between the AVL-tree-based instantiations and BSkiplist-based ones, the round complexity contributes significantly to

[6] Our implementation will be available at `https://gitlab.com/obliviousEMM/emm`.

the query latency. Our experiments show that the BSkiplist-based instantiations and the operation-revealing variants are preferable.

Both Enron and Lucene are commonly used as the target databases to demonstrate the effectiveness of the leakage-abuse attacks. Notably, the email databases are vulnerable to file-injection attacks as everyone can send emails to the target client. Thus evaluating performance on the above two databases carries practical meaning to demonstrate our instantiations' effectiveness minimizing all common SSE/STE leakage patterns while being sufficiently efficient for practice. Though our instantiations cost more bandwidth and rounds than the "leaky" solutions for strong security, we show that they are practically promising in some applications (e.g., email databases).

9 Discussion

We have demonstrated the feasibility of an ORAM-based encrypted index EMM combining an encrypted dictionary EDX with a non-recursive position-based ORAM through simulation and experiments on real-world databases. Due to limited space, we move the discussion to Appendix E.1, on the strengths and limitations of our construction, the trade-offs involved in having a resizable dynamic volume-hiding SSE/STE scheme, and the implications for making our construction resizable (in the number of keywords or database records).

10 Conclusion

We presented a generic encrypted multi-map protocol for the application of searching on outsourced dynamic encrypted databases. Our protocol leverages an encrypted dictionary and an ORAM. Two instantiations and their operation-type-revealing variants were proposed, all using PathORAM, but with different encrypted dictionary instantiations. Our solutions, $\mathsf{EMM}_{\mathrm{avl}}$ and $\mathsf{EMM}_{\mathrm{bskip}}$ provide very strong security, by hiding the information about the data, queries, operation type, and common leakage patterns, namely the query, access, and volume patterns. The operation-type-revealing variants $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$ are more efficient and suitable for scenarios that allow operation-type leakage. Although full padding incurs a high efficiency cost if we wish to prevent any form of volume leakage, under such a strict perfect volume-hiding constraint, our instantiations still outperform the only two known solutions [49], [52] that achieve similar strong security, with improved bandwidth cost and better round complexity than [49]. Since the network latency dominates the efficiency of our solutions, with the increase of network speed (e.g., 1 gbps compared with 100 mbps used in our evaluation), deployment of solutions heavily reliant on network communications may become viable.

Acknowledgement. We thank the anonymous reviewers of SCN '24 for their helpful feedback. Alexandra Boldyreva was supported in part by Cisco Research Award and the National Science Foundation Award No.1946919. Tianxin Tang was funded by an NWO VIDI grant (Project No. VI.Vidi.193.066).

References

[1] C. V. Steve Morgan, "The 2020 data attack surface report," Tech. Rep.
[2] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *2000 IEEE Symposium on Security and Privacy*.
[3] G. Amanatidis, A. Boldyreva, and A. O'Neill, "Provably-secure schemes for basic query support in outsourced databases.," in *DBSec 2007*.
[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *ACM CCS 2006*.

[5] D. Cash, J. Jaeger, S. Jarecki, *et al.*, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014*.

[6] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *ACM CCS 2015*.

[7] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *ACM CCS 2016*.

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

[9] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *ACM CCS 2018*.

[10] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Learning to reconstruct: Statistical learning theory and encrypted database attacks," in *2019 IEEE Symposium on Security and Privacy*.

[11] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted databases: New volume attacks against range queries," in *ACM CCS 2019*.

[12] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution," in *2020 IEEE Symposium on Security and Privacy*.

[13] S. Oya and F. Kerschbaum, "Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption," in *USENIX Security 2021*.

[14] M. Damie, F. Hahn, and A. Peter, "A highly accurate query-recovery attack against searchable encryption using non-indexed documents," in *USENIX Security 2021*.

[15] S. Oya and F. Kerschbaum, "IHOP: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization," in *USENIX Security 2022*.

[16] L. Xu, L. Zheng, C. Xu, X. Yuan, and C. Wang, "Leakage-Abuse Attacks Against Forward and Backward Private Searchable Symmetric Encryption," in *ACM CCS 2023*.

[17] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *CRYPTO 2018, Part I*.

[18] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy*.

[19] L. Reichert, G. R. Chandran, P. Schoppmann, T. Schneider, and B. Scheuermann, "Menhir: An Oblivious Database with Protection against Access and Volume Pattern Leakage," in *AsiaCCS 2024*.

[20] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *EUROCRYPT 2019, Part II*.

[21] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: Attack mitigation for encrypted databases via adjustable leakage," in *USENIX Security 2020*.

[22] S. Patel, G. Persiano, K. Yeo, and M. Yung, "Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing," in *ACM CCS 2019*.

[23] Z. Gui, K. G. Paterson, and S. Patranabis, "Rethinking Searchable Symmetric Encryption," in *2023 IEEE Symposium on Security and Privacy*.

[24] R. Bost, "$\Sigma o \phi o \varsigma$: Forward secure searchable encryption," in *ACM CCS 2016*.

[25] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *ACM CCS 2017*.

[26] I. Miers and P. Mohassel, "IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality," in *NDSS 2017*.

[27] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *ACM CCS 2018*.

[28] S. Sun, X. Yuan, J. K. Liu, *et al.*, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *ACM CCS 2018*.

[29] G. Amjad, S. Kamara, and T. Moataz, "Forward and Backward Private Searchable Encryption with SGX," in *Proceedings of the 12th European Workshop on Systems Security - EuroSec '19*.

[30] G. Amjad, S. Kamara, and T. Moataz, "Breach-Resistant Structured Encryption," in *PoPETs 2019*.

[31] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *NDSS 2020*.

[32] Z. Gui, K. G. Paterson, S. Patranabis, and B. Warinschi, "SWiSSSE: System-Wide Security for Searchable Symmetric Encryption," in *PoPETs 2024*.

[33] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica, "DORY: An Encrypted Search System with Distributed Trust," in *OSDI 2020*.

[34] S.-F. Sun, R. Steinfeld, S. Lai, *et al.*, "Practical non-interactive searchable encryption with forward and backward privacy," in *NDSS 2021*.

[35] G. Amjad, S. Patel, G. Persiano, K. Yeo, and M. Yung, "Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption," in *PoPETs 2023*.

[36] Y. Zhao, H. Wang, and K.-Y. Lam, *Volume-Hiding Dynamic Searchable Symmetric Encryption with Forward and Backward Privacy*, Cryptology ePrint Archive, Report 2021/786.

[37] M. George, S. Kamara, and T. Moataz, "Structured encryption and dynamic leakage suppression," in *EUROCRYPT 2021, Part III*.

[38] P. Grubbs, A. Khandelwal, M.-S. Lacharité, *et al.*, "Pancake: Frequency smoothing for encrypted data stores," in *USENIX Security 2020*.

[39] S. Maiyya, S. Vemula, D. Agrawal, A. E. Abbadi, and F. Kerschbaum, "Waffle: An Online Oblivious Datastore for Protecting Data Access Patterns," in *SIGMOD 2024*.

[40] M. Xu, A. Namavari, D. Cash, and T. Ristenpart, "Searching encrypted data with size-locked indexes," in *USENIX Security 2021*.

[41] A. Boldyreva and T. Tang, *Encrypted multi-map that hides query, access, and volume patterns*, Cryptology ePrint Archive.

[42] Z. Gui, K. G. Paterson, and T. Tang, "Security analysis of mongodb queryable encryption," in *USENIX Security 2023*.

[43] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *J. ACM 1996*.

[44] E. Stefanov, M. van Dijk, E. Shi, *et al.*, "Path ORAM: An extremely simple oblivious RAM protocol," in *ACM CCS 2013*.

[45] L. Ren, C. W. Fletcher, A. Kwon, *et al.*, "Constants count: Practical improvements to oblivious RAM," in *USENIX Security 2015*.

[46] X. S. Wang, K. Nayak, C. Liu, *et al.*, "Oblivious data structures," in *ACM CCS 2014*.

[47] D. S. Roche, A. J. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *2016 IEEE Symposium on Security and Privacy*.

[48] A. Boldyreva and T. Tang, "Privacy-Preserving Approximate k -Nearest-Neighbors Search that Hides Access, Query and Volume Patterns," in *PoPETs 2021*.

[49] M. George, S. Kamara, and T. Moataz, "Structured Encryption and Dynamic Leakage Suppression," in *Advances in Cryptology – EUROCRYPT 2021*.

[50] E. Stefanov, M. van Dijk, E. Shi, *et al.*, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.

[51] L. Ren, C. Fletcher, A. Kwon, *et al.*, "Constants Count: Practical Improvements to Oblivious {RAM}," in *24th USENIX Security Symposium (USENIX Security 15)*.

[52] M. Xu, A. Namavari, D. Cash, and T. Ristenpart, "Searching Encrypted Data with Size-Locked Indexes," in *30th USENIX Security Symposium (USENIX Security 21)*.

[53] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured Encryption and Leakage Suppression," in *Advances in Cryptology – CRYPTO 2018*.

[54] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[55] S. Kamara and T. Moataz, "Computationally Volume-Hiding Structured Encryption," in *Advances in Cryptology – EUROCRYPT 2019*.

[56] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New Constructions for Forward and Backward Private Symmetric Searchable Encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[57] X. S. Wang, K. Nayak, C. Liu, *et al.*, "Oblivious Data Structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[58] D. S. Roche, A. Aviv, and S. G. Choi, "A Practical Oblivious Map Data Structure with Secure Deletion and History Independence," in *2016 IEEE Symposium on Security and Privacy (SP)*.

[59] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal Oblivious RAM," in *Advances in Cryptology – EUROCRYPT 2020*.

[60] S. Oya and F. Kerschbaum, "Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption," in *30th USENIX Security Symposium (USENIX Security 21)*.

[61] *Lucene*, `https://github.com/simon-oya/USENIX21-sap-code`.

[62] M. Naveed, "The Fallacy of Composition of Oblivious RAM and Searchable Encryption," Tech. Rep.

[63] P. Grubbs, A. Khandelwal, M.-S. Lacharité, *et al.*, "Pancake: Frequency Smoothing for Encrypted Data Stores," in *29th USENIX Security Symposium (USENIX Security 20)*.

[64] S. Oya and F. Kerschbaum, "IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization," *arXiv:2110.04180 [cs]*, arXiv: `2110.04180 [cs]`.

[65] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing ORAM and Using It Efficiently for Secure Computation," in *Privacy Enhancing Technologies*.

[66] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan, "Resizable Tree-Based Oblivious RAM," in *Financial Cryptography and Data Security 2015*.

[67] S. Patel, G. Persiano, K. Yeo, and M. Yung, "Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[68] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage," in *29th USENIX Security Symposium (USENIX Security 20)*.

A Applications to Symmetric Searchable Encryption

As we mentioned in the introduction, the encrypted index does not support directly secure search on documents of large size, e.g., photos, videos. An additional round of secure document access is usually suggested after obtaining the document identifiers.

Then how to accomplish the task securely with little leakage? Although retrieving a small number of documents through ORAMs may be feasible, we are interested in other viable methods, especially considering the result by Naveed [62] showing that retrieving matched documents one by one through ORAM costs more bandwidth than transferring the whole encrypted database.

As we have minimized all leakage of the search index in the first stage to obtain the document identifiers, we prefer a practical document access scheme with strong security for the second stage. To the best of our knowledge, Grubbs et al's Pancake scheme [63] can facilitate the needs. It is a frequency-smoothing scheme that introduces fake accesses for queries so that the access pattern is

computationally indistinguishable from uniform distributions. However, their security definition assumes the queries are independent, addressing neither the query-equality leakage nor the adaptive security. We acknowledge the latest query-recovery attack [64], exploiting the query dependence unaddressed by Pancake's non-adaptive security. Thus, it cannot work for the first stage nor directly yield an encrypted multi-map with strong security. We choose it for the second stage is due to the following considerations: first, only document identifiers (e.g., contains no semantic meaning) are sent in the second stage; each document identifier is typically associated with multiple keywords, and these two factors together with Pancake's computationally uniform access pattern mitigate the query-recovery risk. Second, large documents imply data of higher min-entropy, compared with values of small size, e.g., age, posing difficulty to adversaries exploiting query dependence for database reconstruction. Finally, Pancake is highly efficient — within $3-6$ times the cost of the unsecured document access. However, the security-performance trade-off are promoted to the users for choice. A closer examination of adopting this approach in the second stage and its security implication is beyond the scope of this work, and we leave it for future research.

B Extended Functionality for EDX

Definition B1 ($\widetilde{\mathcal{F}}_{\mathsf{DX}}$)**.** *Let label space* $\mathbb{L} = \{0,1\}^*$ *and value space* $\mathbb{V} = \{0,1\}^*$*, where* $\mathsf{dx} \subseteq \mathbb{L} \times \mathbb{V}$*,* $\mathsf{op} \in \{\mathsf{Get}, \mathsf{Put}, \mathsf{Remove}, \mathsf{GetUp}\}$*,* $\widetilde{\mathcal{F}}_{\mathsf{DX}}(\mathsf{dx}, \mathsf{op}, \ell, v)$*: where data structure* dx *is stored in the state,* $\mathsf{op} \in \mathbb{O}$*,* $\ell \in \mathbb{L}$*, and* $v \in \mathbb{V}$*,*

1. *If* $\mathsf{op} \in \{\mathsf{Get}, \mathsf{Put}, \mathsf{Remove}\}$*, define the same as in Definition 32.*
2. *If* $\mathsf{op} = \mathsf{GetUp}$*, parse* v *as* $v_1 \| v_2$*,*
   - *if* ℓ *is in* dx *then* $v^* \leftarrow \mathsf{dx}[\ell]$*; parse* $\mathsf{dx}[\ell]$ *as* $v_1' \| v_2'$*;* $\mathsf{dx}[\ell] \leftarrow v_1' \| v_2$*;*
   - *otherwise:*
     - *if* $v_1 \neq \perp$ *then* $\mathsf{dx}[\ell] \leftarrow v$ *and* $v^* \leftarrow v$*;*
       otherwise $v^* \leftarrow \perp$*.*
3. *Output* (v^*, dx)*.*

Remark 6. The above extended operation GetUp can always be achieved by performing Get then Put as part of the standard functionality $\mathcal{F}_{\mathsf{DX}}$ (Definition 32). However, efficiency is a different issue. For our specific instantiations we will utilize the novel optimization technique that will allow us to have GetUp costing almost the same bandwidth and rounds of communication as performing either Get or Put.

C Non-recursive Position-based ORAM

C.1 Non-recursive Position-based ORAM Definitions

Our definition is new and captures the general non-recursive position-based ORAM. We start with defining a position map and position tag. A position map is a mapping from the memory block ids to sets of physical addresses, known as position tags.

Position-based ORAM Syntax. We adapt the STE syntax for non-recursive position-based ORAM. To manage an N_{ram}-block memory array with block size B_{ram} in bits, fixing label space $\mathbb{L} = [N_{\mathsf{ram}}]$, value space $\mathbb{V} = \{0,1\}^{B_{\mathsf{ram}}}$, operation type space $\mathbb{O}$, we define $\mathsf{ORAM} = (\mathsf{Setup}, \mathsf{Access}, \mathsf{Dec})$. Let $\mathbf{C}$ denote a stateful client, and $\mathbf{S}$ denote the server.

- $\mathsf{pos} \xleftarrow{\$} \mathsf{ORAM.GenPosTag}(1^\kappa, N_{\mathsf{ram}})$: is a randomized algorithm that takes as input security parameter $\kappa \in \mathbb{N}$, number of blocks N_{ram}, and outputs the position tag pos.

- $(K\|\mathsf{pmap}, \mathsf{oarray}) \xleftarrow{\$} \mathsf{ORAM.Setup}(1^\kappa, z, \mathsf{array})$: is a randomized algorithm that takes as input security parameter κ, auxiliary information $z = N_{\mathsf{ram}}\|B_{\mathsf{ram}}$, and $\mathsf{array} \in \mathbb{L} \times \mathbb{V}$ with functionality $\mathcal{F}$. It outputs a secret key K, a position map pmap, and an encrypted array oarray.
- $(v^*, \mathsf{oarray}^*) \leftarrow [\mathsf{ORAM.Access}_{\mathbf{C}}(K\|\mathsf{pmap}[\ell]\|\mathsf{pmap}^*[\ell], \mathsf{op}, \ell, v), \mathsf{ORAM.Access}_{\mathbf{S}}(\mathsf{oarray})]$: is a two-party protocol executed by client $\mathbf{C}$ and server $\mathbf{S}$, where $\mathbf{C}$ inputs secret key K, position tag $\mathsf{pmap}[\ell]$, the updated position tag $\mathsf{pmap}^*[\ell]$, operation type $\mathsf{op} \in \mathbb{O}$, block id $\ell \in \mathbb{L}$, and data $v \in \{0,1\}^{B_{\mathsf{ram}}}$; $\mathbf{S}$ inputs oarray; at the end, $\mathbf{C}$ receives $v^* \in \{0,1\}^{B_{\mathsf{ram}}}$, and $\mathbf{S}$ updates the encrypted array to oarray^*.
- $\mathsf{array} \leftarrow \mathsf{ORAM.Dec}(K, \mathsf{oarray})$: is a deterministic algorithm that takes as input secret key K, encrypted array oarray, and outputs array.

The standard ORAM functionality is defined as follows.

Definition C1 ($\mathcal{F}_{\mathsf{RAM}}$)**.** *For N_{ram}-block memory array with block size B_{ram} in bits, label space $\mathbb{L} = [N_{\mathsf{ram}}]$, value space $\mathbb{V} = \{0,1\}^{B_{\mathsf{ram}}}$, operation type space $\mathbb{O} = \{\mathsf{Read}, \mathsf{Write}\}$, we define reactive functionality $\widetilde{\mathcal{F}}_{\mathsf{RAM}}$, for every $\mathsf{array} \subseteq \mathbb{L} \times \mathbb{V}$, $\mathcal{F}_{\mathsf{RAM}}(\mathsf{array}, \mathsf{op}, \ell, v)$: where array is stored in the state, $\mathsf{op} \in \mathbb{O}$, $\ell \in \mathbb{L}$, and $v \in \mathbb{V}$,*

1. *If $\mathsf{op} = \mathsf{Read}$, if ℓ is in $[N_{\mathsf{ram}}]$ then $v^* \leftarrow \mathsf{array}[\ell]$; otherwise $v^* \leftarrow \perp$.*
2. *If $\mathsf{op} = \mathsf{Write}$, $\mathsf{array}[\ell] \leftarrow v$ and $v^* \leftarrow \perp$.*
3. *Output (v^*, array).*

Correctness. We adapt the correctness definition from Section 3.1. For all $\kappa \in \mathbb{N}$, all $z \in \{0,1\}^*$, all $\mathsf{array} \subseteq \mathbb{L} \times \mathbb{V}$ with functionality $\mathcal{F}$, we say that non-recursive position-based ORAM is correct if and only if the following conditions hold: for all $(K\|\mathsf{pmap}, \mathsf{oarray})$ output by $\mathsf{Setup}(1^\kappa, z, \mathsf{array})$, $\mathsf{Dec}(K, \mathsf{oarray}) = \mathsf{array}$; let $\mathsf{array}_1 \leftarrow \mathsf{array}$, $\mathsf{oarray}_1 \leftarrow \mathsf{oarray}$, and $\mathsf{pmap}_1 \leftarrow \mathsf{pmap}$, after applying an arbitrary polynomial-size sequence of operations $\{(\mathsf{op}_i, \ell_i, v_i)\}_{i\in[q]}$ to oarray_1, where $\mathsf{op}_i \in \mathbb{O}$, $\ell_i \in \mathbb{L}$, $v_i \in \mathbb{V}$; for all $i \in [q]$, all pmap_{i+1} output by $\mathsf{ORAM.GenPosTag}(1^\kappa, N_{\mathsf{ram}})$, $(v_{i+1}, \mathsf{oarray}_{i+1}) \leftarrow [\mathsf{Access}_{\mathbf{C}}(K\|\mathsf{pmap}_i[\ell]\|\mathsf{pmap}_{i+1}[\ell], \mathsf{op}_i, \ell_i, v_i), \mathsf{Access}_{\mathbf{S}}(\mathsf{oarray}_i)]$, we require $v_{i+1} = v'_{i+1}$ and $\mathsf{Dec}(K, \mathsf{oarray}_{i+1}) = \mathsf{array}'_{i+1}$, where $(v'_{i+1}, \mathsf{array}'_{i+1}) \leftarrow \mathcal{F}(\mathsf{array}_i, \mathsf{op}_i, \ell_i, v_i)$.

Security. We show that if a non-recursive position-based ORAM (with encrypted blocks) that supports either standard functionality $\mathcal{F}_{\mathsf{RAM}}$ or extended functionality (used in our generic protocol) satisfying adaptive obliviousness [59], it is secure under STE's adaptive security notion (Definition 31) with an empty leakage profile. We provide a security proof sketch in Appendix F.1.

C.2 Extended Functionality for ORAM

We also specify non-recursive ORAM's extended functionality $\widetilde{\mathcal{F}}_{\mathsf{RAM}}$ to include ReadUp below. The ReadUp operation either removes or adds values in the existing block, indicated by a `flag` (Remove or Put) in the argument. As part of the generic protocol, non-recursive ORAM with $\widetilde{\mathcal{F}}_{\mathsf{RAM}}$ maintains a memory array, of which each block stores a block id concatenated with a set of values.

Definition C2 ($\widetilde{\mathcal{F}}_{\mathsf{RAM}}$)**.** *For N_{ram}-block memory array with block size B_{ram} in bits, let label space $\mathbb{L} = [N_{\mathsf{ram}}]$, value space $\mathbb{V} = \{0,1\}^{B_{\mathsf{ram}}}$, operation type space $\mathbb{O} = \{\mathsf{Read}, \mathsf{Write}, \mathsf{ReadUp}\}$, we define reactive functionality $\widetilde{\mathcal{F}}_{\mathsf{RAM}}$, for every $\mathsf{array} \subseteq \mathbb{L} \times \mathbb{V}$,*

$\mathcal{F}_{\mathsf{RAM}}(\mathsf{array}, \mathsf{op}, \ell, v)$: where array is stored in the state, $\mathsf{op} \in \mathbb{O}$, $\ell \in \mathbb{L}$, and $v \in \mathbb{V}$,

1. *If $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$, define the same as in Definition C1.*
2. *If $\mathsf{op} = \mathsf{ReadUp}$, parse v as $\mathtt{flag}\|\mathtt{vset}$, parse $\mathsf{array}[\ell]$ as $\mathsf{bid}'\|\mathtt{vset}'$, $v^* \leftarrow \mathsf{array}[\ell]$,*
   - *if $\mathtt{flag} = \mathsf{Remove}$, $\mathtt{uset} \leftarrow \mathtt{vset}' \setminus \mathtt{vset}$,*
     $\mathsf{array}[\ell] \leftarrow \mathsf{bid}'\|\mathtt{uset}$.

- *if* $\mathtt{flag} = \mathsf{Put}$, $\mathtt{uset} \leftarrow \mathtt{vset}' \cup \mathtt{vset}$, $\mathsf{array}[\ell] \leftarrow \mathsf{bid}' \| \mathtt{uset}$.

3. *Output* (v^*, array).

Remark 7. The extended operation ReadUp can be generally achieved by performing Read then Write. However, efficiency gains are not always possible. For our instantiations we will apply the optimization technique so that ReadUp cuts the cost up to a half.

C.3 PathORAM

We recall that PathORAM with N_{ram} blocks organizes the encrypted array as a complete binary tree of $\mathtt{height} = \lceil \log_2 N_{\mathsf{ram}} \rceil + 1$, comprising $N_{\mathsf{oram}} = 2^{\lceil \log_2 N_{\mathsf{ram}} \rceil + 1} - 1$ nodes. Each node is a fixed size bucket storing the encrypted blocks. The $\mathtt{bucketSize}$ is chosen based on the degree of obliviousness (part of the security parameter). It suffices to use the leaf node id, an element of the set $[2^{\lceil \log_2 N_{\mathsf{ram}} \rceil}]$, as the position tag since given the leaf node id, the server can find all the blocks on the path from the root to that leaf node. More specifically, $\mathsf{PathORAM.GenPosTag}(1^\kappa, M)$ used as part of EMM is defined as: $x \xleftarrow{\$} [2^{\lceil \log_2 M \rceil}]$ and outputs x. We show how PathORAM naturally supports optimized ReadUp below.

We now show how PathORAM supports efficient extended ReadUp operation. For either Read or Write PathORAM reads and writes back blocks along a tree path. Reading requires decrypting the blocks and storing in the local stash. So instead of either keeping or overwriting the block value, we update the value and perform the same write-back procedure afterwards. The accessed block is mapped to a new random position indicated by the leaf node. All blocks in the local stash on the same path indicated by the old position tag will be padded with dummy blocks, encrypted, and written back to the server. Thus, performing ReadUp compared with Read then Write improves the efficiency by avoiding additional path reading and write-back.

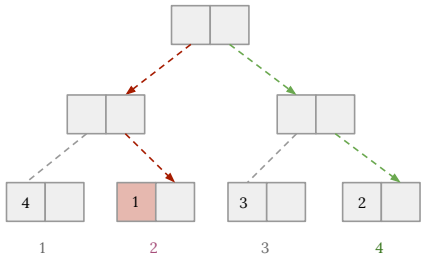


Fig. 6: $N_{\mathsf{ram}} = 4, \mathtt{bucketSize} = 2 \times B_{\mathsf{oram}}$.

Figure 6 shows a toy example of PathORAM to demonstrate ReadUp, where $N_{\mathsf{ram}} = 4, \mathtt{bucketSize} = 2 \times B_{\mathsf{oram}}$, and B_{oram} is the encrypted block size. The node ids are written below the leaf nodes and the block ids are written in the blocks. Other than the numbered blocks, the rest are dummy blocks and all blocks are encrypted. Assume PathORAM.Setup outputs a position map $\mathsf{pmap} = \{(1,2),(2,4),(3,3),(4,1)\}$, where the first field is the block id, and the second field is the leaf node id. To perform ReadUp on block 1 (the red block), similar to Read and Write, generate a random position tag using $\mathsf{ORAM.GenPosTag}(1^\kappa, N_{\mathsf{ram}})$. Suppose 4 is the output. Update $\mathsf{pos}_1^* \leftarrow 4$; fetch and decrypt all the blocks along the red path; perform an update to block 1. Then perform PathORAM's standard lazy write back: the valid blocks and the position tags are kept in the client's local stash. Blocks are written back optimistically if their position tag is matched. Having a stash size of $\mathcal{O}(\log N_{\mathsf{ram}})$ is sufficient to prevent stash overflow. In this example, with $\mathsf{pos}_1^* = 4$, all blocks along the red path

```
Init(s) :
 1: st ← {}
 2: ctr ← 0
 3: stack_size ← s

GetBid() :
 1: if st ≠ ∅ then
 2:     r ← st[0]
 3:     st ← st \ {r}
 4: else
 5:     r ← ctr
 6:     ctr ← ctr + 1
 7: end if
 8: return r

AddBid(x) :
 1: if x = ctr − 1 then
 2:     ctr ← ctr − 1
 3: else
 4:     if #st < stack_size
    then
 5:         st ← st ∪ {x}
 6:     end if
 7: end if

IncCtr() :
 1: ctr ← ctr + 1
```

Fig. 7: Algorithms for Auxiliary Class BidStack.

except block 1 are re-encrypted, padded, and written back to the server. Only when further operations involving accessing the green path, block 1 will be written back to the server.

D Generic Protocol EMM and Recursive Position-based ORAMs

Remark 8 (On Recursive ORAMs). One may ask whether our generic protocol EMM and its instantiations yield a position-based ORAM that is asymptotically better in efficiency than the existing recursive position-based ORAMs, by storing the mappings between block ids and position tags in the encrypted dictionary. The answer is simply no: given N_{ram}, the total number of memory blocks, the recursive ORAM storing the position tags recursively in a series ORAMs in $\mathcal{O}(\log N_{\mathsf{ram}})$ number of levels performs asymptotically the same as looking up a position tag in an oblivious map and then performing non-recursive position-based ORAM access. Interestingly, Wang et al.'s work on oblivious data structures [57] was inspired by how to conduct an efficient search on (recursive) position-based ORAMs in the first place [65]. We emphasize why combining EDX and non-recursive ORAM perfectly fits our setting: for each label, we need to find the block id due to the small client storage constraint, and relying on hashing is not feasible (See Introduction). Therefore, outsourcing the labels and block ids in an encrypted dictionary becomes inevitable. By including additional position tags in the encrypted dictionary, we can obtain (and update) both block id and its position tag on every label look-up. Then, naturally, a non-recursive position-based ORAM to store the label's associated values is a better choice than the recursive one. It costs only two rounds of communication for each ORAM access compared with $\mathcal{O}(\log N_{\mathsf{ram}})$ communication rounds for the recursive one.

E Algorithms for Auxiliary Class BidStack

In Figure 7, we include the algorithms of the auxiliary class BidStack.

E.1 Prices to Pay for Fully Dynamic Volume-hiding SSE/STE

In the following discussion, we say an STE/SSE scheme is "fully dynamic" if it is dynamic and resizable.

On Resizability. We justify our decision not to pursue fully dynamic schemes by showing that all volume-hiding fully dynamic STE/SSE schemes suffer inherently from an unavoidable efficiency overhead: for operation-type-hiding schemes, during Access, every operation needs to yield the same (or computationally indistinguishable) increase in the size of encrypted index, otherwise, the operation type is revealed and also the adversary can exploit the size change to recover the query. Similarly, it follows that even the search and delete need to yield (almost) the same increase in the encrypted index as an insert! As for the operation-type-revealing schemes, though search and delete can now be hidden under the same operation type, without increasing the encrypted size, still, the delete operation is

"fake" that does not actually reduce the size of the encrypted index in order to prevent a volume/size-change attack. A special case of the size-based attack which exploits the different size increases for existing keywords and new keywords (with associated fields are ordered and follow some formatting) was explored by [52].

The above observation also has a direct implication for our construction. If we make our construction resizable in the number of keywords, which can be achieved straightforwardly by instantiating the underlying ORAM with a resizable ORAM (e.g., PathORAM with resizability through techniques discussed in [66]), our construction will leak the exact number of keywords in the encrypted index instead of previously fixed maximum size. This information alone is sufficient to be exploited in the dynamic setting — whether a keyword exists in the encrypted index via keyword/file injection. Thus, for each insertion operation, it must yield (almost) the same size increase regardless of whether the keyword already exists in the index, leaving the scheme with little practical appeal — a curse of fully dynamic volume-hiding SSE/STE schemes given that the adversary has the ability to make the queries and can observe the size change, captured by the standard security definition.

It makes us question whether a fully dynamic volume-hiding STE/SSE scheme with inherent heavy costs for efficiency is worth pursing. Additionally, we will discuss a crucial property that has never been addressed in the literature — varying the tuple length (recall that it refers to the maximum number of documents associated with each keyword), which yields another obstacle to the ultimate goal of designing a fully dynamic volume-hiding SSE/STE scheme.

Since our EMM relies on a single ORAM managing M^* (maximum number of labels) blocks storing the value fields (i.e., document identifiers), then instantiating ORAM with a resizable one only brings the benefits adding dynamism for the keywords, but not the documents (or document identifiers) as the block size is fixed. To our best knowledge, all existing dynamic SSE/STE schemes enforce a (maximum) tuple length on the total number of documents associated with each keyword. However, this constraint is insufficient to address the following practical need: maximum (given) tuple length $l \in \mathbb{N}$, some frequent keyword $\ell \in \mathbb{L}$, after a sequence of document insertions associated with ℓ, clients cannot add ℓ to a newly-inserted document due to that the number of documents associated with ℓ has already reached the maximum length l.

How about not fixing the maximum number of documents N^* while achieving a variable tuple length l? Unfortunately, increasing l is not straightforward in all existing schemes mitigating the volume leakage or achieving the "volume-hiding" property, and we show a simple file-injection-based query-recovery attack on if the scheme allows naively varying l. Though [49] supports unconditionally increasing N, the total number of documents, it is clear that the functionality fails for adding new documents associated with existing keywords when each existing keyword's associated tuple is full.

A Query-Recovery Attack on Variable Tuple Length. We now describe the file-injection keyword-recovery attack against dynamic volume-hiding SSE/STE schemes on variable tuple length. Captured by the standard STE definition 31, the adversary can inject keyword and document (identifier) pairs and can observe the size change in the response transcript or the encrypted index by the adversary. If a SSE/STE scheme reveals the tuple length change in the transcript size or from the size of the encrypted index, then it is vunerable to the following attack.

Consider a multi-map over $\mathbb{L} \times \mathbb{V}$, where keyword set $\mathbb{L} \subseteq \{0,1\}^*$ of size M, document identifier set $\mathbb{V} = [N]$. For simplicity, suppose the initial tuple length $l = 1$, and we choose in the setup stage, a database contains only one pair of keyword and document identifier (ℓ_x, id_x), where $\ell_x \in \mathbb{L}$ and $\mathsf{id}_x \in [N]$. Then, we can adopt the brute-force approach to recover ℓ_x by injecting (adding) pairs $(\ell_i, N+1)$ for all $i \in [M]$ to trigger the change in tuple length. For generality, we allow the client has a local stash of $O(\lambda \cdot l)$, meaning that the client can store λ operation's returned results. We show that it does not affect the effectiveness of the attack for stashless schemes and some schemes with the rebuild functionality as in [49] — after λ operations, the client needs to run the rebuild algorithm. Suppose $\lambda \geq 2$, let $\ell_0 = \ell_1$, for each $i \in [M]$, we perform $\lambda - 1$ search operations on ℓ_{i-1}, followed by

an insertion with $(\ell_i, N+1)$. If resizing occurs (can be detected through the size difference in server storage or encrypted response transcript), then it means a hit on ℓ_x. For $\lambda = 1$ (also implies the typical stashless SSE/STE setting), a similar attack applies — we only need to perform the search operation with ℓ_{i-1} and followed with an insertion with $(\ell_i, N+1)$ for all $i \in [M]$.

On Full Volume Hiding. We comment on the decision to pad the tuple to its maximum length. Random padding can mitigate the leakage to a degree, such as the differentially-private volume padding introduced in [67], but simply averaging the transcript size of repeated queries will make the random padding less effective. SEAL [68] also suggests that for each keyword, padding its associated tuple length to the next power of 2. However, if such a technique is used in the dynamic setting, by adding documents on the same keyword, an adversary can also mount a query-recovery attack exploiting the size change similar to ours. Therefore, one has to accept the efficiency tradeoff due to full-volume-hiding in applications requiring very strong security. An interesting open question is to explore the relaxation of the full volume hiding property but provides meaningful privacy guarantee and the performance gains it allows.

F Security Analysis

F.1 Proof Sketch for Adaptive STE Security of Non-recursive Position-based ORAMs

We show that if a non-recursive position-based ORAM satisfies the adaptive obliviousness [59] and the blocks are encrypted using IND-CPA blockcipher-based mode of operation $\mathcal{SE}$, then it is STE-secure (cf. Definition 31) with empty access leakage and setup leakage containing the public parameters of ORAM — the number of blocks N_{ram} managed by ORAM and block size B_{ram} in bits. For simplicity, assume for each ORAM, we can compute N_{oram}, the number of encrypted blocks using N_{ram}. Let κ denote the security parameter. For any $x \in \mathbb{R}$, we let $\langle\, x \,\rangle$ denote x in the format of a binary string.

We construct a simulator $\mathcal{S}$ that generates a fake encrypted array and simulates transcripts for ORAM.Access with honest server $\mathbf{S}$. We argue that no efficient adversary $\mathcal{A}$ can distinguish the simulated encrypted array and transcripts from those yielded from the real executions of ORAM.Access between honest client $\mathbf{C}$ and honest server $\mathbf{S}$. Our primary technique is that, given the setup leakage — N_{ram} number of blocks, block size B_{ram}, security parameter κ. Fix an IND-CPA symmetric encryption scheme $\mathcal{SE}$ used for block encryption. The simulator $\mathcal{S}$ first generates a random $K_{\mathcal{S}}$ using $\mathcal{SE}$'s key generation algorithm on security parameters κ. Simulator $\mathcal{S}$ then encrypts N_{oram} (computed using N_{ram} and κ) $\langle\, 0 \,\rangle$'s of bit length B_{ram}. Then for $i = 1$ to N_{oram}, $\mathsf{EDS}_1[i] \leftarrow \mathcal{E}_{K_{\mathcal{S}}}(\langle\, 0 \,\rangle)$, where $\langle\, 0 \,\rangle$ is of bit-length B_{ram}. At the end, $\mathcal{S}$ outputs the encrypted array EDS_1 and sends it to $\mathcal{A}$. Adversary $\mathcal{A}$ adaptively makes q queries using ORAM.Access. For each ORAM.Access, simulator $\mathcal{S}$ sends the server $\mathbf{S}$ a transcript which is a list of physical addresses generated by $\mathsf{ORAM.GenPosTag}(1^\kappa, N_{\mathsf{ram}})$. Honest server $\mathbf{S}$ then sends all encrypted blocks associated with those physical addresses to $\mathcal{S}$. Simulator $\mathcal{S}$ then writes back a list of updated encrypted blocks to the same associated addresses back to $\mathbf{S}$. Finally, honest $\mathbf{S}$ will update and output EDS_{t+1}. The "adaptive obliviousness" of ORAM ensures that the physical addresses (i.e., position tags) are computationally indistinguishable in both worlds. Together with the IND-CPA property of encrypted blocks guaranteed by $\mathcal{SE}$, ensure computationally indistinguishable transcripts in both worlds.

F.2 Proof Sketch for Theorem 61

In the generic EMM protocol, we follow the same procedure as in standard non-recursive position-based ORAM for updating the position map but differ in that the position map is kept in an encrypted dictionary EDX instead of on the client's side. Since we require our building block encrypted dictionary to satisfy (strong) adaptive STE security (Definition 31), ideally only with the setup leakage and the

access leakage containing at most the operation type, it follows that the privacy of the position map hold.

We now describe how we construct a simulator $\mathcal{S}$ that uses $\mathcal{S}_e, \mathcal{S}_o$ — simulators for $\mathsf{EDX}, \mathsf{ORAM}$ for their STE security experiments respectively, as subroutines in the EMM's security experiment. During the setup stage, simulator $\mathcal{S}$ runs $\mathcal{S}_e$ and $\mathcal{S}_o$ on EDX and ORAM's setup leakage, outputs the two simulated data structures, and sends them to the adversary $\mathcal{A}$. During EMM.Access, $\mathcal{S}$ first runs $\mathcal{S}_e$ on the EDX's access leakage with honest server $\mathbf{S}$ and outputs the same as $\mathcal{S}_e$. $\mathcal{S}$ then runs $\mathcal{S}_o$ on ORAM's access leakage with honest server $\mathbf{S}$ and outputs the same. The transcripts produced by the honest executions between the client $\mathbf{C}$ and the server $\mathbf{S}$, and the ones simulated by the $\mathcal{S}$ with honest server $\mathbf{S}$ are computationally indistinguishable since EDX and ORAM both satisfy (strong) STE security with their setup and access leakage. Thus we can conclude EMM is adaptively STE secure with a leakage profile that is a union of EDX and ORAM in setup and access, specified in Theorem 61.

F.3 EDX Instantiations and Optimization

EDX Instantiations AvlTreeE and BSkiplistE. We briefly describe AvlTreeE, BSkiplistE — novel variants of OAvlTreeE, OBSkiplistE, allowing the leakage of the operation type of the underlying encrypted dictionary. AvlTreeE modifies OAvlTreeE by removing the dummy accesses for Get and GetUp to ensure operation obliviousness. For each Get (or GetUp), when no nodes are added or removed to the underlying AVL tree, the number of ORAM accesses is reduced from suggested $3 \times \lceil 1.45 \times \log(M+2) \rceil$ in [48], [57] to the maximum height of the AVL tree $\lceil 1.45 \times \log(M+2) \rceil$. It yields $3\times$ efficiency enhancement in rounds of communication and bandwidth costs for Get and GetUp (with no AVL-tree structural change). Similarly, BSkiplistE modifies OBSkiplistE by removing the dummy accesses and thus reduce the total number of ORAM accesses for Get to the maximum height of the BSkiplist $\lceil \log_\beta(M) \rceil$, yielding $2\times$ efficiency enhancement in rounds of communication and bandwidth costs for Get and GetUp (with no BSkiplist structural change).

EDX **with Optimized** GetUp**.** We first show OBSkiplistE and OAvlTreeE support efficient GetUp without affecting their security. Since GetUp performs the same procedures as Put, except instead of overwriting, it performs an update in plaintext on the client's side — from the adversary's view, it is no different from Put. The adaptive obliviousness guarantees that Put, Get, Remove produce computationally indistinguishable transcripts, and thus we reach the same security conclusion for OBSkiplistE and OAvlTreeE with optimized GetUp. As AvlTreeE results from OAvlTreeE after reducing the total number of ORAM accesses to $1.45 \times \lceil \log(M+2) \rceil$ for Get and GetUp (for cases incurring no structural change to the AVL tree), where M is the upper-bound on the total number of labels, it satisfies the same STE security as OAvlTreeE but reveals $\texttt{numAccess}_{\text{avl-oram}}$ — the number of accesses to the underlying ORAM.

PathORAM **with Optimized** ReadUp**.** We show that PathORAM with optimized ReadUp is with the same leakage profile as standard one: ReadUp is computationally indistinguishable from performing a Read since they follow the same procedure of Read except it updates the target block's value instead of keeping the same value, which only occurs in plaintext and on the client's side. Each ReadUp produces a transcript computationally indistinguishable from Read's, and thus computationally indistinguishable from Write's, guaranteed by the adaptive obliviousness. Hence, PathORAM with optimized ReadUp has the same leakage profile as standard PathORAM.

F.4 Access Leakage of $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\mathrm{bskip}}$

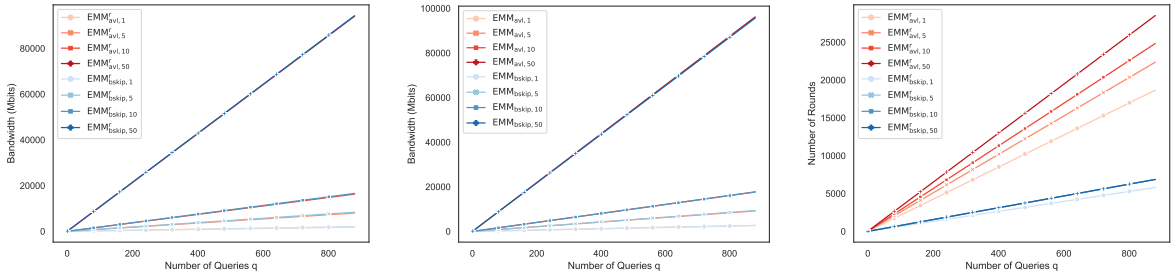
We make the following justification on the access leakage. The encrypted dictionary instantiation AvlTreeE, BSkiplistE are adaptively $\mathcal{L}_{\mathsf{AvlTreeE}}$-secure, adaptively $\mathcal{L}_{\mathsf{BSkiplistE}}$-secure (cf. Definition 31),

having the same setup leakage as OAvlTreeE and OBSkiplistE accordingly. But, both have access leakage instead of empty — $\mathcal{L}_{\mathsf{AvlTreeE.Access}} = \mathtt{numAccess}_{\text{avl-oram}}$ and $\mathcal{L}_{\mathsf{BSkiplistE.Access}} = \mathtt{numAccess}_{\text{bskip-oram}}$, where $\mathtt{numAccess}_{\text{avl-oram}}$ and $\mathtt{numAccess}_{\text{bskip-oram}}$ denote the number of accesses to the underlying ORAM as part of AvlTreeE and BSkiplistE respectively. Though our generic protocol EMM only uses encrypted dictionary operation GetUp and Remove, the number of accesses to the encrypted dictionary's underlying ORAM can differ: for every operation that is either GetUp or Remove, if no node is added or removed to the underlying AVL tree (or BSkiplist), the number of ORAM access for each operation is either $\lceil 1.45 \times \log(M+2) \rceil$ (or $\lceil \log_\beta M \rceil$); otherwise, is fixed at $3 \times \lceil 1.45 \times \log(M+2) \rceil$ (or $2 \times \lceil \log_\beta M \rceil$). Since each EDX instantiation has consistent communicational volume for all accesses to its underlying ORAM, no additional information is leaked from $\mathtt{numAccess}_{\text{avl-oram}}$ (or $\mathtt{numAccess}_{\text{bskip-oram}}$) other than the EDX's operation type and the setup leakage. It follows that partial information about the multi-map operation type is leaked, and thus we include the operation type leakage during EMM.Access for $\mathsf{EMM}^{\mathrm{r}}_{\text{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\text{bskip}}$.

G Complexity Analysis (Continue)

G.1 Concrete Complexity

Parameters and Expressions. Same as in [49], we fix the keyword size to 256 bits and value size (document identifier size) to 64 bits. The branching factor β of the OBSkiplistE is set to 12 as in [48], [58]. Let block size $B_{\text{avl}} = 512$ (for AvlTreeE and OAvlTreeE) and $B_{\text{bskip}} = 2307$ (for BSkiplistE and OBSkiplistE and B_{bskip} may vary based on the degree of obliviousness). The total bandwidth cost for q operations is $\mathtt{Bandwidth}_{\mathsf{edx}} + \mathtt{Bandwidth}_{\mathsf{oram}}$. Let block size for ORAM storing the document identifiers $B_{\mathsf{ram}} = \min\{N, 64 \cdot l\} + 32$, where the document identifiers are represented by either N-bit vector or the maximum tuple of size l containing 64-bit document identifiers; the extra 32-bit is to store the block id. $\mathtt{Bandwidth}_{\mathsf{oram}}$ is consistent with all instantiations for retrieving the document identifiers. To compute the bandwidth, we use $\mathtt{bucketSize}_{\text{avl}} = 4 \cdot B_{\text{avl}}, \mathtt{bucketSize}_{\text{bskip}} = 6 \cdot B_{\text{bskip}}, \mathtt{bucketSize}_{\text{poram}} = 4 \cdot B_{\mathsf{ram}}$. The encrypted bucket size $\mathtt{EBucketSize}_x$ for all $x \in \{\text{avl}, \text{bskip}, \text{poram}\}$ are computed using length padding and IV appending (e.g., AES-based mode of operations). For any $\mathtt{bucketSize}$ in bits, then its encryption has size $\lceil \lceil \mathtt{bucketSize}/8 \rceil / 16 \rceil \cdot 128 + 128$. Then, $\mathtt{Bandwidth}_{\mathsf{oram}} = 2 \cdot (\mathtt{EBucketSize}_{\text{poram}} \cdot (h_{\text{poram}} + 1) + 32)$. $\mathtt{Bandwidth}_{\mathsf{edx}} = 2 \cdot (r_i - 3) \cdot (\mathtt{EBucketSize}_x \cdot (h_{\text{poram}} + 1) + 32)$ for r_i in Figure 5f and $x \in \{\text{avl}, \text{bskip}\}$.



(a) Revealing Operation Types. (b) Hiding Operation Types. (c) Revealing Operation Types.

Fig. 8: Bandwidth Cost and Round Complexity Comparison Hiding/Revealing Operation Types.

Simulation Results. We continue the discussion in Section 7. Figure 8 shows the bandwidth comparison between $\mathsf{EMM}_{\text{avl}}$, $\mathsf{EMM}_{\text{bskip}}$, also between $\mathsf{EMM}^{\mathrm{r}}_{\text{avl}}$ and $\mathsf{EMM}^{\mathrm{r}}_{\text{bskip}}$), using the growth factor in

$\{1, 5, 10, 50\}$ with the default database parameters. We found BSkiplist-based instantiations always outperform the AVl-tree-based ones in round complexity, but the simulated bandwidth does not differ much between the two types of instantiations (AVL tree or BSkiplist), which can be justified by the fact that the bandwidth costs are dominated by retrieving the document identifiers through ORAM. The parameters we used for our implementations (Figure 5f) also reflect this property — the block size for EDX is much less than the ORAM's.

G.2 Asymptotic Complexity Comparison with [49]

Notation. For ease of comparison, we use the nomenclature in [49] in the following discussion, and for simplicity, we focus only on multi-maps and refer the readers to their work for generalized abstraction. Given arbitrary multi-map ds with query space $\mathbb{Q}_{\mathsf{ds}}$, response space $\mathbb{R}_{\mathsf{ds}}$, and update space $\mathbb{U}_{\mathsf{ds}}$; the subscript indicates the space is specific for data structure ds. Let ds_λ be the extended data structure with capacity λ, which can be interpreted as extending the original data structure ds with up to λ operations. Let w be the word size in bits used as the basic data size unit in the RAM computational model, and $|x|_w$ stands for x's size in number of words.

Comparing with Generic Construction [49]. We first summarize the query computational complexity results from [49] and then compare those with our construction's. We use notations consistent with [49] — let time denote the computational complexity, and DDS denote their generic construction.

Assume DDS's base structured encryption (STE) scheme has query computational complexity $\mathcal{O}(\log \#\mathbb{Q}_{\mathsf{ds}})$ and $\lambda = \mathcal{O}(\#\mathbb{Q}_{\mathsf{ds}})$, then the query computational complexity of generic construction DDS is as follows,

$$\begin{aligned}\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} =& \mathcal{O}(\#\mathbb{Q}_{\mathsf{ds}} \cdot \log \#\mathbb{Q}_{\mathsf{ds}}) + \mathcal{O}(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}_\lambda}} |r|_w \cdot \log^2 \lambda) \\ &+ \mathcal{O}(\#\mathbb{Q}_{\mathsf{ds}} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{ds}}). \end{aligned} \tag{1}$$

The query computational complexity of our construction for λ operations is, $\mathsf{time}^{\mathsf{emm}}_{\lambda\mathsf{O}} = \mathcal{O}(\lambda \cdot \log^2 \#\mathbb{Q}_{\mathsf{ds}^*} + \lambda \cdot \log \#\mathbb{Q}_{\mathsf{ds}^*} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}^*}} |r|_w)$.

If we make the same assumption $\lambda = O(\#\mathbb{Q}_{\mathsf{ds}})$ as in [49], then, $\mathsf{time}^{\mathsf{emm}}_{\lambda\mathsf{O}} = \mathcal{O}(\#\mathbb{Q}_{\mathsf{ds}} \cdot \log^2 \#\mathbb{Q}_{\mathsf{ds}^*} + \#\mathbb{Q}_{\mathsf{ds}} \cdot \log \#\mathbb{Q}_{\mathsf{ds}^*} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}^*}} |r|_w)$.

Comparing with asymptotic complexity of DDS in Equation 1, where $\mathcal{O}(\#\mathbb{Q}_{\mathsf{ds}} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{ds}})$ is the dominant term, we can compute that under assumptions $\log^2 \#\mathbb{Q}_{\mathsf{ds}^*} = \mathcal{O}(\log^2 \#\mathbb{Q}_{\mathsf{ds}} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}}} |r|_w)$ and $\log \#\mathbb{Q}_{\mathsf{ds}^*} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}^*}} |r|_w = \mathcal{O}(\log^2 \#\mathbb{Q}_{\mathsf{ds}} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}}} |r|_w)$, our construction is at least as efficient as their generic construction DDS in query computational complexity [49], namely, $\mathsf{time}^{\mathsf{emm}}_{\lambda\mathsf{O}} = \mathcal{O}(\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}})$.

Comparing with Concrete Instantiations [49]. We first recap the complexity results of two concrete instantiations AZL and ZAVLH in [49]. AZL is a perfectly-correct fully-dynamic rebuildable scheme resulting from applying their dynamic leakage suppression framework to the perfectly-correct variant of PBS in [53]. It has the following query complexity, $\mathsf{time}^{\mathsf{azl}}_{\lambda\mathsf{O}+\mathsf{R}} = (\lambda + \#\mathbb{Q}_{\mathsf{ds}}) \cdot \mathsf{time}^{\mathsf{pbs}}_{\mathsf{Q}} + \mathcal{O}\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}_\lambda}} |r|_w \cdot \log^2 \lambda\right) + \mathcal{O}\left(\#\mathbb{Q}_{\mathsf{ds}} \cdot \max_{r \in \mathbb{R}_{\mathsf{ds}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{ds}_\lambda}\right)$, where $\mathsf{time}^{\mathsf{pbs}}_{\mathsf{Q}}$ denotes the query complexity of the base scheme PBS, which is equal to the query complexity of its underlying multi-map encryption scheme.

The other instantiation is ZAVLH — a dynamic rebuildable multi-map encryption scheme results from applying their framework to dynamic AVLH^d [53]. Briefly, let n be the number of bins storing a multi-map of size N, where N is the sum over all labels of the labels' tuple lengths. Each label l is mapped at random to t out of n bins, where t is the maximum tuple length. Assume $t = \mathcal{O}(1)$ and

$n = \mathcal{O}(N/\log N)$, then the query complexity $\mathsf{time}_{\mathsf{Q}}^{\mathsf{zavlh}}$ is $\mathcal{O}(t \cdot N/n) = \mathcal{O}(\log N)$, which yields the query complexity of the whole scheme ZAVLH as follows,

$$\mathsf{time}_{\lambda\mathsf{O}+\mathsf{R}}^{\mathsf{zavlh}} = \mathcal{O}(\#\mathbb{L}_{\mathsf{mm}} \cdot \log N) + \mathcal{O}\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{mm}_\lambda}} |r|_w \cdot \log^2 \lambda\right) + \mathcal{O}\left(\#\mathbb{L}_{\mathsf{mm}} \cdot \max_{r \in \mathbb{R}_{\mathsf{mm}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{L}_{\mathsf{mm}_\lambda}\right).$$

Based on the above results, the two instantiations all perform asymptotically the same as the generic construction. Thus, under the same assumptions as in [49], our construction is (at least) as efficient as the two instantiations.

Round Complexity. Through private conversations with the authors [49], we confirm the round complexity of their rebuild operation used as part of their instantiations is $\mathcal{O}(Q' + Q' \log^2 Q' + \#\mathbb{Q}_{\mathsf{ds}_\lambda} \cdot R)$, where R is the add/edit round complexity of the encrypted data structure and $Q' = \#\mathbb{Q}_{\mathsf{ds}} + \lambda$. The overall round complexity is dominated by the oblivious sorting term $\mathcal{O}(Q' \log^2 Q') = \mathcal{O}\left((\#\mathbb{Q}_{\mathsf{ds}} + \lambda) \log^2(\#\mathbb{Q}_{\mathsf{ds}} + \lambda)\right)$, assuming that the add/edit round complexity of the encrypted data structure is at most linear in the size of the query space. Their round complexity for rebuild for every λ operation is significant, compared with $\mathcal{O}(\log \#\mathbb{Q}_{\mathsf{ds}^*})$ incurred by our generic protocol EMM for each operation.