POPE: Partial Order-Preserving Encoding

Daniel Roche [*] Daniel Apon [†] Seung Geol Choi [*] Arkady Yerukhimovich [‡]

Abstract

Recently there has been much interest in performing database queries over encrypted data to enable functionality while protecting sensitive data. One particularly efficient mechanism for executing such queries is order-preserving encryption/encoding (OPE) which results in ciphertexts that preserve the relative order of the underlying plaintexts thus allowing range and comparison queries to be performed directly over the ciphertext. In particular, Popa et al. (S&P 2013) recently gave the first interactive, mutable order-preserving encoding scheme achieving the strongest possible security for OPE while allowing for efficient range queries. However, this construction requires the bulk of the work to be performed when inserting data into the database, something that is not desirable for the high insertion rates of today's big data databases.

In this paper, we propose an alternative approach to range queries over encrypted data that is optimized for efficient insert while still maintaining search functionality. Specifically, we propose a new primitive called partial order preserving encoding (POPE) that achieves ideal OPE security while providing extremely fast insertion and efficient (amortized) search. Our scheme is better suited to today's insert-heavy database scenarios. For example, with about one million insertions and one thousand range queries, our POPE scheme is 20X faster than the scheme by Popa et al.

We also propose a new form of frequency-hiding security for POPE, as recently studied by Kerschbaum (CCS 2015) for OPE, and show how to extend our scheme to satisfy it. Altogether, one view of our results is that POPE is a new, fast "gracefully-leaky ORAM" for a particular, randomized interpretation of the (partial) sorting functionality.

1 Introduction

A common workflow in "Big Data" applications is to collect and store a large volume of information, then later perform some analysis (i.e., queries) over the stored data. In many popular NoSQL key-value stores such as Google BigTable [CDG+06] and its descendants, e.g. [DHJ+07, Thea, Theb, Thec], the most important query operation is a *range query*, which selects rows in a contiguous block sorted according to any label such as an index, timestamp, or row id.

In order to support high availability, low cost, and massive scalability, these databases are increasingly stored on remote and potentially untrusted servers, driving the need to secure the stored data. While traditional encryption protects the confidentiality of stored data, it also destroys ordering information that is necessary for the efficient server-side processing, notably for range queries. An important and practical goal is therefore to provide data security for the client while allowing efficient query handling by the database server.

[*]United States Naval Academy, {roche, choi}@usna.edu.
[†]University of Maryland, dapon@cs.umd.edu.
[‡]MIT Lincoln Laboratory, arkady@ll.mit.edu.

Order-preserving encryption (OPE) [AKSX04, BCLO09, BCO11] offers a simple and efficient solution for performing range queries over encrypted data. Specifically, OPE guarantees that $\mathsf{enc}(x) > \mathsf{enc}(y)$ iff $x > y$. Thus, range queries can be performed directly over the ciphertexts in the same way that such a query would be performed over the plaintext data. However, OPE comes with a security cost. None of the original schemes [AKSX04, BCLO09] achieve the *ideal* security goal for OPE of IND-OCPA (indistinguishability under ordered chosen-plaintext attack) [BCLO09] in which ciphertexts reveal no additional information beyond the order of the plaintexts. In fact Boldyreva et al. [BCLO09] prove that achieving a stateless encryption scheme with this security goal is impossible under reasonable assumptions. The existing schemes, instead, either lack formal analysis or strive for weaker notions of security which have been shown to reveal significant amount of information about the plaintext [BCO11].

This impossibility was recently circumvented by Popa et al. [PLZ13] who showed how to build an order-preserving *encoding* scheme[1]. This scheme differs from traditional encryption in two ways. First, the encoding procedure is *interactive* requiring multiple rounds of communication between the data owner (client) and the database (server). Second, the ciphertexts produced are *mutable* so previously encoded ciphertexts may have to be updated when a new value is encoded. Roughly, the Popa et al. scheme works by building a binary search tree containing (encrypted versions of) all the data elements at the leaves. The OPE encoding of a value is just its position in this binary search tree. Thus, encoding works by the client and server jointly traversing this tree with the client decrypting the value at the internal nodes and indicating whether the value being encoded is larger or smaller. If the part of the tree into which the inserted value would go is already full, Popa et al. use a tree rebalancing procedure to make space for the new value. Clearly, this approach requires $O(\log n)$ rounds of communication and total communication of $O(\log n)$ encrypted values per encoding to traverse the binary tree; the same amount of communication is needed per range query to encode the end-values of the range. However, the client only needs enough memory to store two ciphertexts at a time for comparison. A different trade-off between client storage and communication is given by Kerchbaum and Schropfer [KS14] achieving minimal $O(1)$ communication to encode elements (from a uniform random distribution), but requiring *linear*-size, *persistent* client storage — proportional to the storage requirements on the remote database itself.

When used for range queries over encrypted data these two schemes either require require significant communication and work by the server, or significant client storage to handle data insertion in order to achieve efficient range queries that can be executed directly over the OPE encodings. As such, these schemes work well when there are relatively few inserts and many queries performed over the data. However, in many big data applications, the opposite is true with many more values added to the database than are ever searched or retrieved. For example, a typical application might be the collection of data from low-powered sensor networks as in [WGA06], where insertions are numerous and happen in real-time, whereas queries are processed later and on more capable hardware.

Hence, this paper asks the following question:

In the scenario of a huge number of insertions and a moderate number of range queries, can we design a secure range-query scheme with small client storage and much better efficiency?

1.1 Our Work

Our contributions. In this paper we give a positive answer to the above question, proposing an alternative scheme for range queries that we call a partially order preserving encoding or POPE. Specifically, our POPE

[1]We abuse notation and use OPE to refer to both order-preserving encryption and order-preserving encoding.

	Comm. Rounds		Amortized	Client Storage	
	Insert	Query	Communication	Storage Size	Persistence
Here	1	$O(1)$	$O(1)$	$O(n^\epsilon)$	No
[PLZ13]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	No
[KS14]	1	1	$O(1)$	$O(n)$	Yes

Figure 1: Comparison of OPE-based range search schemes. n is the total number of inserts. The communication complexity is given in number of encrypted elements. For our scheme we require at most $O(n^{1-\epsilon})$ total number of queries.

construction satisfies the following properties when storing n items, provided at least $O(n^\epsilon)$ client-side memory and at most $O(n^{1-\epsilon})$ number of range queries for any constant $1 > \epsilon > 0$:

- IND-OCPA security leaking nothing beyond the order of inserted elements.
- Trivial insert operations consisting of 1 message from the client to the server and no computation for the server.
- Expected $O(1)$-round, amortized $O(1)$-bandwidth per range query.
- No persistent client storage between query operations.

See Figure 1 for how this compares to existing schemes. Our scheme is especially suitable for typical big data applications where there are many more inserts than queries. As an example data point, with about one million insertions and one thousand range queries, our POPE scheme is 20X faster than the scheme by Popa et al. See Section 7 for a more detailed comparison.

Our approach. Our main technique to make this possible is *lazy sorting*. Specifically, unlike OPE, we do not sort the encoded values on insert, instead only partially sorting values when necessary during querying. If we regard the actual location in the search tree data structure as an implicit encoding of an encrypted value, our scheme gives partially ordered encoding, and hence the name of our construction POPE (partially order preserving encoding). This allows for extremely efficient insertion and for the cost of sorting encrypted elements to be amortized over the queries performed. In particular, on each query we only need to sort the part of the data that is accessed during the search, leaving much of the data untouched. Additionally, since encodings are sorted during searches, the cost of performing a batch of search queries is often much cheaper than performing these queries individually as later queries no longer need to sort any elements already sorted in earlier queries.

At a high level, our scheme works by building a buffer tree [Arg03] where every node contains an unsorted buffer and a sorted list of elements. The invariant that we maintain is that the elements stored in the sorted list of a node impose a partial order on the values stored (both in the sorted list and unsorted buffer) at that node's children. That is, all values at child i will lie between values $i-1$ and i in the parent's sorted list. Intuitively, this enables the sorted values at each node to serve as an array of simultaneous pivot elements, in the sense of Quicksort [Hoa62]. To maintain this property we make use of client storage to partition a set of unsorted elements according to the values at the parent. Specifically, we require the client to be able to read in a buffer of $O(n^\epsilon)$ encrypted values and to partition them according to these split points. Using this amount of storage we can ensure that the depth of the buffer tree remains $O(1)$ allowing for low amortized latency over batches of client queries.

Frequency-hiding security. Recently, Naveed et al. [NKW15] showed a concrete attack on ideal OPE security (IND-OCPA), using the revealed order information to learn a large fraction of underlying data in a medical database. [NKW15]'s attack highlights the care that must be taken when instantiating real-world systems with the weaker security options (e.g. OPE) of CryptDB [PRZB11]; further discussion on this point may be found in [PZB15]. To counter the [NKW15] attack, Kerschbaum [Ker15] proposed a stronger notion of security that also hides the *frequency* of OPE-encoded elements (i.e. hides equality). To show security, Kerschbaum's scheme requires programming a random oracle in the proof.

In Section 6, we propose a new definition of frequency-hiding for POPE, and show how to extend our scheme to satisfy it without using a random oracle. Further, we argue that the ROM is likely too strong of an assumption in the context of POPE. Indeed, with a programmable random oracle we can also achieve an *even stronger*, "non-committing" notion of POPE security (so strong as to be unrealistic).

1.2 Related Work

OPE alternatives. In addition to OPE there are several other lines of work that enable searching over encrypted data. Typically, these works stronger security than provided by OPE; in particular they do not reveal the full order of the underlying data as happens with OPE. However, this comes at a significant performance cost with even the latest schemes being one to two orders of magnitude slower than the latest OPE-based implementations [PRZB11, KGM$^+$14].

Symmetric searchable encryption (SSE) was first proposed by Sawn, Wagner, and Perrig [SWP00] who showed how to search over encrypted data for keyword matches in sub-linear time. Goh [Goh03] showed how to support conjunctive queries, Curtmola et al. [CGKO06] showed how to support multiple querying clients, and Kamara et al. [KPR12] showed how to allow for stateful modifications to the database. Recent works [PKV$^+$14,CJJ$^+$13,FJK$^+$15] achieve performance within a couple orders of magnitude of unencrypted databases for rich classes of queries including boolean formulas over keyword, and range queries. We refer interested readers to the survey by Bosch et al. [BHJP14] for an excellent overview of this area.

Oblivious RAM [Gol87,SCSL11,SvDS$^+$13,WCS15] and oblivious storage schemes [GMOT12,AKST14, DvDF$^+$16,MMB15] can be used for the same applications as OPE and POPE, but achieve a stronger security definition that additionally hides the access pattern, and therefore incur a larger performance cost than our approach. Indeed, one view of our results is that POPE is a (fast) "gracefully-leaky ORAM" for a certain, randomized interpretation of the (partial) sorting functionality.

Finally, we note that techniques such as fully-homomorphic encryption [Gen09], public-key searchable encryption [BCOP04, BW07, SBC$^+$07], and secure multi-party computation [Yao86, BGW88, GMW86] can enable searching over encrypted data while achieving the strongest possible security. However, these approaches would require performing expensive cryptographic operations over the entire database on each query and are thus prohibitively expensive. Very recently cryptographic primitives such as order-revealing encryption [BLR$^+$15] and garbled random-access memory [GLOS15] have offered the potential to achieve this level of security for sub-linear time search. However, all constructions of these primitive either rely on very non-standard assumptions or are prohibitively slow.

Lazy data structures and I/O complexity. The data structure that forms the basis for our construction is similar in concept to the Buffer Tree of [Arg03]. Their data structure delays insertions and searches in a buffer stored at each node, which are cleared (thus executing the actual operations) when they become sufficiently full. The main difference here is that our buffers contain only insertions, and they are cleared only when a search operation passes through that node.

We also point out an interesting connection to I/O complexity regarding the size of local storage. In our construction, as in [PRZB11], the client is treated as an oracle to perform comparisons of ciphertexts. If we think of the client's memory as a "working space" of size L, and the server's memory as external disk, then from [AV87] it can be seen that performing m range queries on a database of size $n \geq m$ requires a total transfer bandwidth of at least $\Omega(m \log_L m)$ ciphertexts. (This is due to the lower bound on the I/O complexity of sorting, and the fact that m range queries can reveal the order of a size-m subset.) In particular, this means that the mOPE construction from [PRZB11] cannot be improved without either limiting the number of queries, or increasing the client-side storage, both of which we will do.

2 System Settings and Considerations

Data format. We view the data elements inserted into the database as blocks consisting of a label ℓ and a value v. All range searches are performed over the labels and the corresponding values are returned for the matching labels.

Participants. Our range query system consists of three entities: the client, the server, and the comparison oracle.

- The client C has no relevant storage or computational capability other than encryption and decryption. The client initiates all the operations on the database by sending encrypted queries to the cloud server (and decrypting the responses). C must have the ability to encrypt both labels and values; i.e., C must hold all cryptographic keys. There can be multiple clients sharing the same cryptographic keys as long as they are able to coherently decrypt and encrypt labels and values.
- The cloud server S stores the entire encrypted database, and is responsible to execute all operations efficiently. S holds no encryption or decryption keys, and should learn nothing about the database contents except an (partial) order on the plaintext labels.
- The comparison oracle O holds the decryption key for labels (but not values) in the database, and is therefore able to decrypt the labels, perform comparison-based computations at the request of the server S, and return the results. The operations of O are constrained so that nothing more is learned by S than the order of the encrypted labels.

We generally assume that the server S has the largest computational and storage capabilities, followed by the oracle O which has polynomially smaller storage and constrained computational costs, and the client C which has finite (i.e. $O(1)$) storage and no computational responsibility beyond encryption and decryption.

Possible deployment scenarios. Introducing the comparison oracle as a separate entity allows us to consider several different scenarios:

- If C and O are the same (and S is distinct), then we have the typical single-user cloud scenario described in the introduction. The client and server will execute a protocol in order to insert and retrieve database entries without revealing too much information to the server about the contents of the database. The client has limited space and computational resources, and we want to develop efficient protocols under these restrictions.
- If C, S, and O are all distinct, then we have the multi-user cloud scenario, where O is a semi-trusted third party with limited resources.
- If S and O are the same (and C is distinct), then the client trusts the cloud server with a decryption key for the labels, in order to provide greater efficiency. The client's data is secure from any third parties that access the server's long-term (encrypted) storage. Note that this seems to be a typical

use-case in currently-available cloud platforms, where data is stored encrypted but (some) decryption is performed remotely.

- If C, S, and O are all the same machine, then we may consider the server's memory to be high-volume persistent memory (such as a hard disk), and O's storage to be the erasable, secure memory (such as RAM or an encrypted thumb drive). Then the database contents will be hidden from an attacker who can access the persistent memory but not the erasable memory.

Performance goals and relevant parameters. In our analysis we assume the storage capabilities of the oracle O are fixed and known. We wish to minimize (in decreasing order of priority)

- the amount of communication between S and O,
- the amount of computation required by O,
- the amount of storage required on the server S,
- and finally, the amount of computation required by S.

These costs will be expressed in terms of three parameters:

- n: The total number of items inserted into the database.
- m: The total number of range search (or range delete) operations performed.
- L: The total number of encrypted labels that the oracle O can store in its local memory.

3 Partial Order Preserving Encoding

In this section, we define partial order preserving encoding schemes and their security requirements. We assume that readers are familiar with security notions of standard cryptographic primitives [KL07]. Let λ denote the security parameter.

Syntax. We first define the syntax for POPE schemes. Let $\Pi_\ell = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and $\Pi_v = (\mathsf{vKeyGen}, \mathsf{vEnc}, \mathsf{vDec})$ be both standard private-key encryption schemes. A POPE scheme is defined as

$$(\Pi_\ell, \Pi_v, \mathsf{InitState}, \mathsf{Cmp}, \mathsf{Insert}, \mathsf{Search}, \mathsf{Delete}).$$

Here, Π_ℓ and Π_v are used when the client encrypts labels and values respectively. Moreover, after the client creates the encryption keys (k, k') for both labels and variables, she sends the label encryption key k to the comparison oracle. We will use $\bar{\ell}$ and $\bar{v}$ to denote the encrypted version of label ℓ and value v. The other algorithms are defined as follows:

- $\mathsf{st} \leftarrow \mathsf{InitState}(1^\lambda)$. The server S will run this algorithm, which takes as input the security parameter and outputs an initial state st.
- $\mathsf{Cmp}(k, (\bar{\ell}_1, \ldots \bar{\ell}_q))$. The comparison oracle will run this comparison algorithm that takes as input the label encryption key and an arbitrary number q of encrypted labels. The output of the algorithm will be determined by specific POPE constructions; the only restriction is that the output should be determined only by *the input encrypted labels and the order of the labels*, that is, $(\bar{\ell}_1, \ldots, \bar{\ell}_q)$ and $(j_1, \ldots, j_q)$ where ℓ_i is the j_i-th largest element in the sequence.

- $\mathsf{st}' \leftarrow \mathsf{Insert}^{\mathsf{Cmp}(k,\cdot)}((\bar{\ell}, \bar{v}), \mathsf{st})$. The server runs this *insertion algorithm* to insert the encrypted label-value pair in its state st. The algorithm outputs the updated state st′. Note this algorithm may potentially query the oracle $\mathsf{Cmp}(k, \cdot)$, which captures the interactions between the server S and the comparison oracle O.
- $(\mathsf{st}', \{(\bar{\ell}'_j, \bar{v}'_j)\}_j) \leftarrow \mathsf{Search}^{\mathsf{Cmp}(k,\cdot)}(\bar{\ell}_1, \bar{\ell}_2, \mathsf{st})$. The server will run this *search algorithm* with the help of the oracle Cmp, i.e., we will have $\ell_1 \le \ell'_j \le \ell_2$.
- $\mathsf{st}' \leftarrow \mathsf{Delete}^{\mathsf{Cmp}(k,\cdot)}(\bar{\ell}_1, \bar{\ell}_2, \mathsf{st})$. The server will run this *deletion algorithm* with the help of the oracle Cmp. All encrypted items $\{(\bar{\ell}'_j, \bar{v}'_j)\}_j$ such that $\ell_1 \le \ell'_j \le \ell_2$ will be deleted from the server database.

We remark that the above syntax is for the sake of generality, and Insert (resp., Search, Delete) may not use the comparison oracle or randomness. In fact, Insert in our construction is a deterministic algorithm which doesn't query Cmp.

Security. We now give the formal definition of security for POPE.

Definition 1 (IND-OCPA security)**.** *Security of POPE is defined through the following experiment. For an algorithm* F *with access to oracles, define* the view of F's execution, *written* VIEW(F)*, as the collection of the input to* F*, the randomness used by* F*, and the answers from the oracles.*

```
***Experiment*** $\mathsf{EXP}^{\text{ind-ocpa}}_{\mathcal{A}}(\mathsf{POPE}, \lambda, b)$
$k \leftarrow \mathsf{KeyGen}(1^\lambda)$; $k' \leftarrow \mathsf{vKeyGen}(1^\lambda)$; $\mathsf{st} \leftarrow \mathsf{InitState}(1^\lambda)$;
Send st to $\mathcal{A}$;
$i \leftarrow 1$;
For $j = 1, \ldots, poly(\lambda)$:
    $op_j \leftarrow \mathcal{A}$
    If $op_j$ is 'insert':
        $\mathsf{F}_j \leftarrow \mathsf{Insert}$
        $((\ell_i^0, v_i^0), (\ell_i^1, v_i^1)) \leftarrow \mathcal{A}$
        $\bar{\ell}_i \leftarrow \mathsf{Enc}_k(\ell_i^b)$; $\bar{v}_i \leftarrow \mathsf{vEnc}_{k'}(v_i^b)$; $\mathsf{F}_j^{\mathsf{Cmp}(k,\cdot)}(\bar{\ell}_i, \bar{v}_i, \mathsf{st})$
        $i \leftarrow i + 1$
    Else if $op_j$ is 'search' or 'delete':
        $\mathsf{F}_j \leftarrow$ Search or Delete depending on $op_j$.
        $((\ell_i^0, \ell_{i+1}^0), (\ell_i^1, \ell_{i+1}^1)) \leftarrow \mathcal{A}$
        $\bar{\ell}_i \leftarrow \mathsf{Enc}_k(\ell_i^b)$; $\bar{\ell}_{i+1} \leftarrow \mathsf{Enc}_k(\ell_{i+1}^b)$; $\mathsf{F}_j^{\mathsf{Cmp}(k,\cdot)}(\bar{\ell}_i, \bar{\ell}_{i+1}, \mathsf{st})$
        $i \leftarrow i + 2$
    Send VIEW($\mathsf{F}_j$) to $\mathcal{A}$.
    Update st from the output of $\mathsf{F}_j$.
Output what $\mathcal{A}$ outputs.
```

Note in the above experiment, we assume the adversary $\mathcal{A}$ is stateful, and the variable i is used as a counter for the label encryptions. We call the adversary $\mathcal{A}$ admissible *if $\mathcal{A}$ always chooses labels and values such that all labels are of the same length, and all values are of the same length, $\ell_i^0 < \ell_j^0$ iff $\ell_i^1 < \ell_j^1$ for all i, j. We say that a POPE scheme is IND-OCPA if for any admissible PPT adversary $\mathcal{A}$, the following ensembles are computationally indistinguishable:*

$$\{\mathsf{EXP}^{\text{ind-ocpa}}_{\mathcal{A}}(\mathsf{POPE}, \lambda, 0)\}_\lambda \approx_c \{\mathsf{EXP}^{\text{ind-ocpa}}_{\mathcal{A}}(\mathsf{POPE}, \lambda, 1)\}_\lambda.$$

Remark on VIEW(F). The *view* of F's execution is essentially the view of the semi-honest server. The input label encryption(s) corresponds to the message from the client, and the input state corresponds to the data structure containing the label encryptions. The randomness used by F is the randomness used by the server, and the oracle answer is the message from the comparison oracle.

4 Main Construction

Recall that the parameter n represents the total number of items inserted into the database, and the parameter m represents the total number of range search (or range delete) operations performed. The comparison oracle Cmp can store $L + O(1)$ labels in its local memory. Let $L = n^\epsilon$ for constant $\epsilon > 0$.

In our system, the client C and comparison oracle Cmp share a secret key k for the label encryption scheme Π_ℓ. Whenever the client must communicate a block (ℓ, v) to the server, C sends $(\bar{\ell}, \bar{v})$ to S, where $\bar{\ell} \leftarrow \mathsf{Enc}_k(\ell)$ and $\bar{v} \leftarrow \mathsf{vEnc}_{k'}(v))$ with a value encryption key k held only by the client. Whenever the server must communicate an encrypted label $\bar{\ell}$ to the oracle, S forwards the ciphertext $\bar{\ell}$ to Cmp, which internally uses shared key k to operate on plaintext label ℓ directly.

However for the remainder of the construction description, we will suppress explicit notation for encryption and decryption, deferring discussion of the properties required (or desired) of schemes Enc and vEnc to the subsequent analysis in Section 5.

Server memory layout. The server S maintains a type of lazy buffer tree $\mathcal{T}$, which is a balanced tree with root r where every node has at most $L + 1$ children.

- Each *non-leaf* node u of the tree will be associated with an unsorted buffer and a sorted list. An unsorted buffer can hold an a-priori unbounded number of blocks $\{(\ell_1, v_1), (\ell_2, v_2), ...\}$, and a sorted list can hold up to L labels $(\ell_1, ..., \ell_L)$.
- Each *leaf* node u_{leaf} has only an unsorted buffer $\{(\ell_1, v_1), (\ell_2, v_2), ...\}$.

We remark that, in general, for every distinct label ℓ, there could be many distinct blocks $(\ell, v_1), (\ell, v_2), ...$ stored in $\mathcal{T}$. However, in the presentation of the Split algorithm below, we will restrict to the special case when for each label ℓ there is at most one block (ℓ, v) in $\mathcal{T}$ in order to convey the main ideas more clearly.

Main invariant of tree $\mathcal{T}$. We will enforce the following order-invariant on tree $\mathcal{T}$:

> Let ℓ_{j-1} and ℓ_j be the $(j-1)$th and jth sorted labels at some non-leaf node u in $\mathcal{T}$. Then, for all labels ℓ in the sub-tree $\mathcal{T}_{u_j}$ rooted at the jth child u_j of u, we have $\ell_{j-1} < \ell \leq \ell_j$.

Intuitively, this guarantee of global partial ordering enables the L sorted labels $\ell_1, ..., \ell_L$ at each node u to serve as an array of simultaneous *pivot elements*, in the sense of Quicksort [Hoa62], for the $L + 1$ sub-trees rooted at u's (at most) $L + 1$ children $u_1, ..., u_{L+1}$. We will use this idea plus the parameter setting $L = n^\epsilon$, implying $\mathcal{T}$ has depth $\lceil 1/\epsilon \rceil = O(1)$, to enable the server to traverse and maintain the tree $\mathcal{T}$ with low amortized latency over repeated batches of client-queries.

Organization. For simplicity of presentation, we separate the various interfaces into **(i)** client-server operations and **(ii)** server-oracle subroutines. We assume that action begins with some client request to the server, selected from the list of available operations. During the execution of some operation, the server may invoke one or more subroutine calls involving server-queries to the comparison oracle Cmp.

In particular, the final subroutine, Split, is at the core of our construction, but is quite detailed. For the reader's sake, we devote extra care to explaining in detail how this subroutine behaves.

4.1 Client-Server Operations

There are two basic operations – Insert (for batches of blocks) and Search (for ranges of blocks) – that are defined recursively at each node u of the tree $\mathcal{T}$. *Without loss of generality, the client C will always* *initiate requests to the server S as "$\mathsf{Insert}(\cdot, r)$" and "$\mathsf{Search}(\cdot, r)$" for* <u>*current*</u> *root r of tree $\mathcal{T}$*. However, the server may recursively call these operations on non-root nodes $u \in \mathcal{T}$. On receipt of a valid client operation-request, i.e. for the current root r of $\mathcal{T}$, the server S will initiate a sequence of interactive queries to the comparison oracle Cmp in order to update/navigate the tree $\mathcal{T}$ and (for searches) will also return any identified blocks $\{(\ell'_1, v'_1), (\ell'_2, v'_2), ...\}$ to C.

We also show how to efficiently implement a third operation, Delete (for ranges of blocks), by making minor changes to our implementation of Search. Altogether, this defines the tuple of operations $\mathsf{Op} = (\mathsf{Insert}, \mathsf{Search}, \mathsf{Delete})$.

The Insert **operation.** We use the following syntax for inserting unstructured batches of blocks[2]:

$$\mathsf{Op.Insert}^{\mathcal{T}}\left(\{(\ell_i, v_i)\}_i, u\right),$$

where the sub-tree $\mathcal{T}_u \subseteq \mathcal{T}$ rooted at u is updated with u'. To insert blocks at node $u \in \mathcal{T}$:

Algorithm 1: Insert Operation

```
let $u'$ be $u$ but with blocks $\{(\ell_1, v_1), (\ell_2, v_2), ...\}$ appended to its buffer;
store $u'$ in $\mathcal{T}$ instead of $u$.
```

The Search **operation.** We use the following syntax for retrieving ordered ranges of blocks:

$$\{(\ell'_j, v'_j)\}_j \leftarrow \mathsf{Op.Search}^{\mathcal{T}, \mathsf{Cmp}(k,\cdot)}\left(\ell_1, \ell_2, u\right),$$

where the sequence of blocks $\{(\ell'_1, v'_1), (\ell'_2, v'_2), ...\}$ is returned. If the search originated at root $r \in \mathcal{T}$, the sequence of blocks is returned *to the client*; else, the server returns a (sub)sequence to its own recursive call. To search for blocks in the range $[\ell_1, \ell_2]$ contained in the sub-tree $\mathcal{T}_u \subseteq \mathcal{T}$ rooted at $u \in \mathcal{T}$:

Algorithm 2: Search Operation

```
let $\big((\ell^*_1, v^*_1), \mathsf{found}_1\big) := \mathsf{SubR.Split}^{\mathcal{T}, \mathsf{Cmp}(k,\cdot)}(\ell_1, u, \mathsf{left})$;
let $\big((\ell^*_2, v^*_2), \mathsf{found}_2\big) := \mathsf{SubR.Split}^{\mathcal{T}, \mathsf{Cmp}(k,\cdot)}(\ell_2, u, \mathsf{right})$;
let $\{(\ell'_1, v'_1), (\ell'_2, v'_2), ...\} := \mathsf{SubR.Traverse}^{\mathcal{T}}(\ell^*_1, \ell^*_2)$;
output $\{(\ell'_1, v'_1), (\ell'_2, v'_2), ...\}$.
```

Roughly speaking, the output (ℓ^*, v^*) from $\mathsf{SubR.Split}$ is the leftmost (resp. rightmost) block matching the *split-point* ℓ_1 or ℓ_2. See below for the actual detailed description of the subroutine.

The Delete **operation.** We use the following syntax for deleting ordered ranges of blocks:

$$\mathsf{Op.Delete}^{\mathcal{T}, \mathsf{Cmp}(k,\cdot)}\left(\ell_1, \ell_2, u\right),$$

To remove all blocks in the range $[\ell_1, \ell_2]$ contained in the sub-tree $\mathcal{T}_u \subseteq \mathcal{T}$ rooted at $u \in \mathcal{T}$:

[2]We suppress the Cmp oracle notation for Insert, since the oracle is unused.

```
Algorithm 3: Delete Operation
  let $\{(\ell'_j, v'_j)\}_j := \mathsf{Op.Search}^{\mathcal{T},\mathsf{Cmp}(k,\cdot)}(\ell_1, \ell_2, u)$;
  remove every block in the sequence $\{(\ell'_1, v'_1), (\ell'_2, v'_2), ...\}$ from $\mathcal{T}$.
```

4.2 Server-Oracle Subroutines

The server S locally invokes subroutines and queries the comparison oracle Cmp to complete them. We define the tuple of subroutines $\mathsf{SubR} = (\mathsf{Traverse}, \mathsf{Rebalance}, \mathsf{Split})$.

Comparison oracle. The comparison oracle is parameterized by public $L \in \mathbb{N}$, and we have $q \leq L$; it stores secret key k to access labels ℓ of ciphertexts $\bar{\ell}$. We use the notation

$$\mathsf{Cmp}[\ell_1, ..., \ell_q](k, \ell)$$

to describe the oracle. We differentiate between two modes of operation: *sorting* and *streaming*.

- When *sorting*, the input is $(\ell_1, ..., \ell_q, \ell)$, and the output is a sorted list $(\ell_{i_1}, \ldots, \ell_{i_q})$ and $(j, \mathsf{found}) \in [q+1] \times \{0, 1\}$, where j is such that $\ell_{i_{j-1}} < \ell \leq \ell_{i_j}$, with ℓ_{i_0} and $\ell_{i_{q+1}}$ defined as $-\infty$ and ∞ respectively, and $\mathsf{found} = 1$ iff $\ell = \ell_j$ for some $j \in [q]$.
- When *streaming*, semi-persistent inputs $(\ell_1, ..., \ell_q)$ are first uploaded to the oracle, and then input ℓ is received from a stream. In the streaming mode, only (j, found) is returned as output.

The Traverse **subroutine.** We use the following syntax for traversing the tree $\mathcal{T}$ from ℓ_1 to ℓ_2:

$$\{(\ell'_1, v'_1), (\ell'_2, v'_2), ...\} \leftarrow \mathsf{SubR.Traverse}^{\mathcal{T}}(\ell_1, \ell_2),$$

where a sequence of every block (ℓ'_i, v'_i) found between ℓ_1 and ℓ_2 in the partial ordering of $\mathcal{T}$ is returned to the server. Traversals may be implemented by any common tree-traversal algorithm.

The Rebalance **subroutine.** We use the following syntax for rebalancing the tree $\mathcal{T}$ at a node u:

$$\mathsf{SubR.Rebalance}^{\mathcal{T}}(u),$$

We demand the following, additional invariants on calls to $\mathsf{SubR.Rebalance}(u)$:

1. u is a non-leaf node,
2. u has an empty unsorted buffer,
3. and u contains at most $2L$ sorted labels and $2L + 1$ children.

To rebalance the tree $\mathcal{T}$ at node u, proceed as in Algorithm 4.

4.3 The Split Subroutine

Intuition — *How to Split a Tree*. The Split subroutine simultaneously services two primary objectives:

1. Search for an end-point logical position, e.g. in order to answer some client-generated range query

Algorithm 4: Rebalance Subroutine

```
if $u$ has $L$ or fewer labels :
    halt
if $u$ has no parent :
    create a new root node $r$;
choose the median label of $u$ to promote to the list of the parent of $u$;
split labels and children of $u$ accordingly, creating a new sibling internal node $u^*$;
call $\mathsf{SubR.Rebalance}^{\mathcal{T}}(\mathsf{parent}(u))$;
```

2. Re-arrange the tree to *speed up* subsequent calls to Split (and achieve good amortized costs).

The idea is best explained by illustrating the desired behavior/effect of splitting the tree at different stages of its lifecycle. Details follow.

Prior to the first split of the tree $\mathcal{T}$, a (potentially large) sequence of unsorted ciphertexts have been (blindly) inserted into the lone buffer of the single, initial node/root $r \in \mathcal{T}$. So immediately prior to the first split of the tree, we can picture the state of the server's data as the monolithic node r in Figure 2.

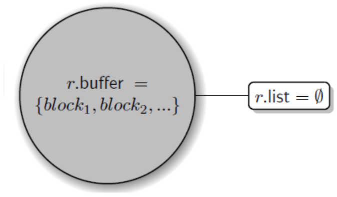

Figure 2: The server's tree $\mathcal{T}$ immediately prior to its first split: just an unsorted buffer.

In the first split of the sole node $r \in \mathcal{T}$, L random labels are chosen uniformly without replacement from the buffer of r. These are sent to the oracle, who sorts them. A new root node r' is created with new children $u_1, ..., u_{L+1}$. (It is possible to view the old node r as being equal to one of the otherwise-newly-created u_i.)

The server then streams every block's label ℓ in the buffer of r through the oracle, and learns the *sorted* position $i \in [L+1]$ of each such block. All blocks originally in the buffer of r are inserted into the buffer of their respective u_i matching revealed order $i \in [L+1]$. (During the course of this procedure, the searched-for end-point label ℓ^* will be identified to the server by the oracle when it is streamed through the oracle's view.)

The new state of the tree $\mathcal{T}$, immediately after the *first* Split call, is depicted in Figure 3.

In subsequent splits of the tree $\mathcal{T}$, the split is recursively evaluated *node-wise* on the path beginning at the current root $r^* \in \mathcal{T}$ and terminating in the particular leaf node u^* that contains the search term ℓ^* (if it exists in the tree).

For these later queries, the splitting behavior depends on whether the current action is on an *internal* (i.e. non-leaf) node of the tree, or on a *leaf* node of the tree. (It also depends on whether the search-label ℓ^* is encountered prior to reaching a leaf, as in this case, we will terminate this path-wise recursion of Split, and just output the answer to the client's query more quickly.)

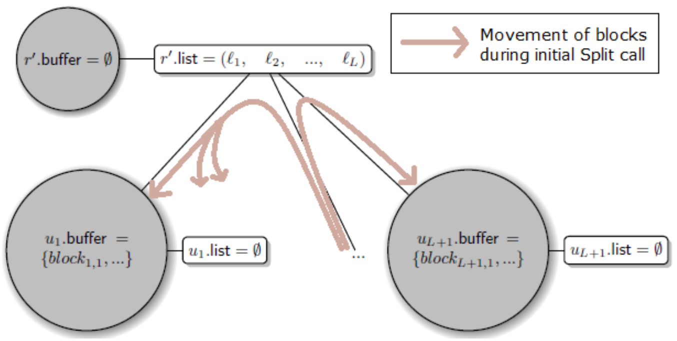
r'.buffer $= \emptyset$
r'.list $= (\ell_1, \ \ell_2, \ ..., \ \ell_L)$
Movement of blocks during initial Split call
u_1.buffer $= \{block_{1,1}, \ldots\}$
u_1.list $= \emptyset$
u_{L+1}.buffer $= \{block_{L+1,1}, \ldots\}$
u_{L+1}.list $= \emptyset$

Figure 3: The server's tree $\mathcal{T}$ after one split: a root, and $L + 1$ leaves — i.e. "Splitting" a leaf.

At internal nodes u, subsequent touches by Split will emphasize "clearing the buffer of u," which involves streaming every block stored at u to the oracle, and partitioning these blocks into the $L + 1$ children of u according to the oracle's index-responses. (Note, however, we do not need to pick L random labels to promote here, since the node u already contains a *fixed* list of L labels that we have previously promoted to u.)

This behavior of subsequent splits of internal nodes is illustrated in Figure 4.

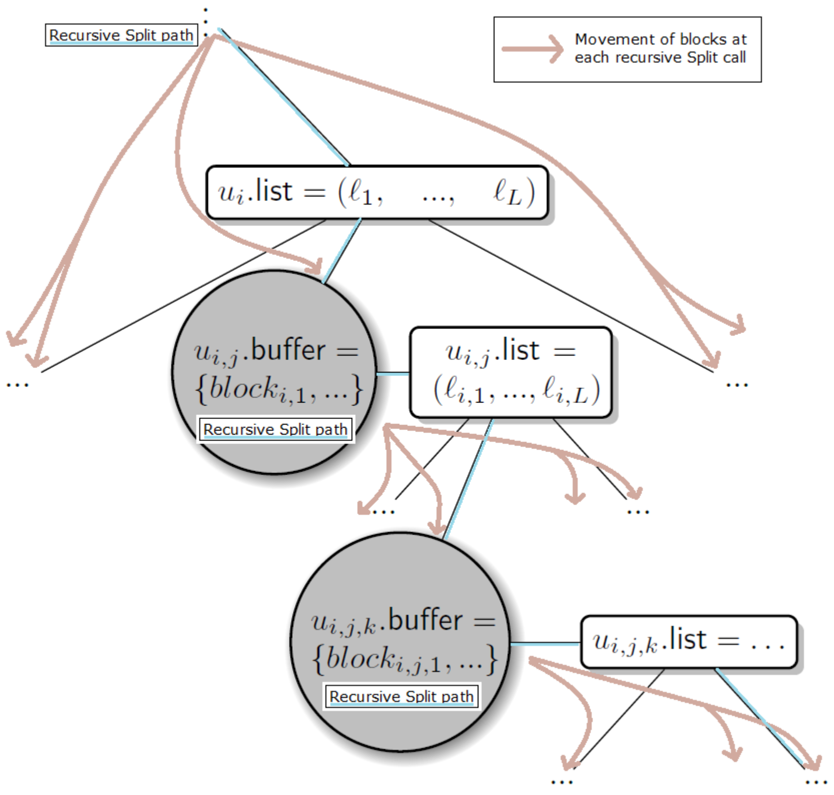
Recursive Split path
Movement of blocks at each recursive Split call
u_i.list $= (\ell_1, \ ..., \ \ell_L)$
$u_{i,j}$.buffer $= \{block_{i,1}, \ldots\}$
Recursive Split path
$u_{i,j}$.list $= (\ell_{i,1}, ..., \ell_{i,L})$
$u_{i,j,k}$.buffer $= \{block_{i,j,1}, \ldots\}$
Recursive Split path
$u_{i,j,k}$.list $= \ldots$

Figure 4: Internal blocks in subsequent splits of the server's tree $\mathcal{T}$ — i.e. "Flushing" (recursively).

At leaf nodes u, the recursive procedure of path-splitting naturally terminates, since the global Split evaluation either must find the search-label ℓ^* by this point, or conclude no such ℓ^* exists in $\mathcal{T}$ (and instead find the best-possible logical address for the corresponding range query). However, we enforce an additional (more stringent) requirement on termination of Split at leaf nodes of the tree — namely, that the leaves of the tree should be approximately **balanced** both in the number of blocks stored, and their depth in the tree. (Fortunately, it will turn out to be very cheap to achieve this.)

More concretely, if the leaf node u is low weight (i.e. contains less than L blocks), then we are content and halt Split with the output discovered at u. If on the other hand, the leaf node u is *high weight* (i.e. contains more than L blocks), then the server will *internally and independently* (without additional interaction with the comparison oracle) rebalance the structure *while maintaining the partial order invariant of the tree*. As (infrequently) required, this rebalancing may (recursively) spawn a fresh root r^{**} of the tree (and split the extra blocks across the two new, equal-size sub-trees) More frequently, the server will simply re-arrange its logical positioning of blocks laterally.

Formal Split **syntax.** We use the following syntax for splitting sub-trees $\mathcal{T}_u \subseteq \mathcal{T}$ along label ℓ^*:

$$\big((\ell^*, v^*), \mathsf{found}\big) \leftarrow \mathsf{SubR.Split}^{\mathcal{T},\mathsf{Cmp}(k,\cdot)}(\ell, u, \mathsf{side}).$$

for side $\in \{\mathsf{left}, \mathsf{right}\}$. The subroutine updates subtree $\mathcal{T}_u$ so that it maintains a partial ordering of blocks in $\mathcal{T}_u$ with respect to ℓ. The output block (ℓ^*, v^*) is (if possible) the leftmost (resp. rightmost) block best-matching the *split-point* ℓ. If ℓ^* and ℓ match, found $= 1$.

Formal Split **pseudocode.** For ease of presentation, we partition the pseudocode of Split into three possible branches "at the top level" of the global Split subroutine's logic. Concretely, the server's (stateful) execution of a request to $\mathsf{SubR.Split}^{\mathcal{T},\mathsf{Cmp}(k,\cdot)}(\ell^*, u, \mathsf{side})$ will begin by branching based on which one of the three following conditions is presently true of the node $u \in \mathcal{T}$:

If u is an internal (i.e. non-leaf) node: Run Algorithm 5, as depicted in Figure 4.

```
Algorithm 5: Split Subroutine: Case (1)
(when $u$ is an internal node)

if $u$ is an internal node :      ▷ clear the buffer of $u$
    upload query $\ell^*$ and the sorted labels $(\ell^*, \ell_1, ..., \ell_L)$ at $u$ to Cmp;
    for each block $(\ell, v)$ in the buffer of $u$ do
        as a stream, send $\ell$ to the oracle $\mathsf{Cmp}[\ell^*, \ell_1, ..., \ell_L](k, \cdot)$;
        for each label $\ell$ in the stream do
            Cmp returns a pair $(j, \mathsf{found})$;
              ▷ index $j \in [L+2]$ is order of $\ell$ in $\{\ell^*, \ell_1, ..., \ell_L, \ell\}$
            if found $= 1$ :
                halt and output $(\ell^*, v^*) := (\ell, v)$;
            else:
                call $\mathsf{Op.Insert}^{\mathcal{T}}(\ell, u_j)$ for node $u_j$ the $j$-th child of $u$;
    query $\mathsf{Cmp}[\ell^*, \ell_1, ..., \ell_L](k, \ell^*)$ to obtain $(j^*, 1)$;
    call and output from $\mathsf{SubR.Split}^{\mathcal{T},\mathsf{Cmp}(k,\cdot)}(\ell^*, u_{j^*})$ for $j^*$th child $u_{j^*}$;
```

If u a leaf node and $|u| \leq L$: Run a brute-force search; i.e. Algorithm 6.

Algorithm 6: Split Subroutine: Case (2)
(when u is a leaf node and stores at most L blocks)

```
else if $u$ is a leaf node :
    if $u$ contains at most $L$ blocks :    $\triangleright$ find output
        upload all $\ell \in u$ and $\ell^*$ to Cmp;
        if $\ell^* = \ell_{j^*}$ is in the list :
            Cmp returns a sorted list $(\ell_1, \ell_2, ..., \ell_s)$,
            plus the pair $(j^*, \mathsf{found})$ for $j^* \in [s]$;
        else:
            Cmp returns the pair $(j', \mathsf{not\ found})$ for $j' \in \{0\} \cup [s]$
            immediately preceding $\ell^*$ in the sorted order;
        let $u'$ be the node with (temporarily) sorted buffer $(\ell_1, ..., \ell_s)$;
        halt and output $(\ell_{j^*}, v)$ (resp. $(\ell_{j'}, \perp)$);
```

If u a leaf node and $|u| > L$: Run Algorithm 7, as depicted in Figure 3 (plus rebalancing as in Algorithm 4).

Algorithm 7: Split Subroutine: Case (3)
(when u is a leaf node and stores more than L blocks)

```
else if $u$ contains more than $L$ blocks :    $\triangleright$ then split $u$
    uniformly sample $L$ labels $\{\ell'_{r_1}, ..., \ell'_{r_L}\}$ from $u$;
    send $\{\ell'_{r_1}, ..., \ell'_{r_L}\}$ to $\mathsf{Cmp}(k, \cdot)$; receive the sorted list $(\ell'_1, ..., \ell'_L)$;
    if $u$ is the $j$-th child of parent $p$ with $L' \leq L$ sorted labels :
        expand $p$'s list to contain $(\ell_1, ..., \ell_{j-1}, \ell'_1, ..., \ell'_L, \ell_j, ..., \ell_{L'})$;
        query $\mathsf{Cmp}[\ell'_1, ..., \ell'_L](k, \ell^*)$ to obtain $(j^*, \mathsf{found})$ and let $u$ be the $(j + j^* - 1)$th child of $p$;
        as a stream for $\ell \in u$,
            $(j', \mathsf{found}) := \mathsf{Cmp}[\ell'_1, ..., \ell'_L, \ell^*](k, \ell)$;
        for each $(\ell, j')$ pair:
            call $\mathsf{Op.Insert}^{\mathcal{T}}((\ell, v), u_{j+j'-1})$;
        then call $\mathsf{SubR.Rebalance}^{\mathcal{T}}(p)$;
        then call $\mathsf{SubR.Split}^{\mathcal{T}, \mathsf{Cmp}(k,\cdot)}(\ell^*, u, \mathsf{side})$;
    else if $u$ has no parent (i.e. it is the only node in the tree) :
        create a new root $r$ with empty buffer and sorted list $(\ell'_1, ..., \ell'_L)$;
        query $\mathsf{Cmp}[\ell'_1, ..., \ell'_L](k, \ell^*)$ to obtain $(j^*, \mathsf{found})$ and let $u$ be the $j^*$th child of $r$;
        as a stream for $\ell \in u$,
            $(j, \mathsf{found}) := \mathsf{Cmp}[\ell'_1, ..., \ell'_L, \ell^*](k, \ell)$;
        for each $(\ell, j)$ pair: if $\mathsf{found} = 1$ :
            halt and output $(\ell^*, v^*) := (\ell, v)$;
        else:
            call $\mathsf{Op.Insert}^{\mathcal{T}}(\ell, u_j)$ for node $u_j$ the $j$-th child of $r$;
```

5 Analysis

5.1 Cost Analysis

We derive amortized upper bounds on **server/oracle** rounds and bandwidth per operation.

Theorem 1. *After n insertions and m searches with local storage of size L, our scheme has the following server/oracle query costs:*

1. Insert *is free,*
2. Search *requires $O(\log_L n)$ rounds in expectation. Since $L = n^\epsilon, \epsilon > 0$, this is $\underline{O(1) \text{ rounds in expectation}}$.*
3. *The total bandwidth over all $(n+m)$ operations is $O(mL\log_L(n) + n\log_L(m) + n\log_L(\log(n)))$.*
4. *Without restriction, the amortized bandwidth per operation is $O((L/\log(L))\log(n))$.*
5. *When $mL = O(n)$, the amortized bandwidth per operation is $O(\log(n)/\log(L))$.*
6. *When $mL = o(n)$, the amortized bandwidth per operation IS $O(\log(m)/\log(L) + \log\log(n)/\log(L))$.*
7. *For $mL \leq n, L = n^\epsilon$, constant $\epsilon > 0$ — i.e. $m \leq n^{1-\epsilon}$ — the $\underline{\text{amortized bandwidth per operation is } O(1)}$.*

Proof. We use the following parameters for the cost analysis:

- n, the total number of blocks inserted by the client
- m, the total number of search queries by the client
- L, local storage capacity of the oracle ($\pm O(1)$)
- k, the number of blocks at a fixed leaf

Choose n, ϵ so that $L = n^\epsilon > 16$.

The case of Insert is trivial to analyze: The server never communicates with the oracle.

Analyzing Search (resp. Delete) is more involved. We begin by examining the most complicated component of the construction first, i.e. the Split subroutine. At its core, this subroutine takes a leaf node with $> L$ blocks, and makes it smaller, creating new leaf nodes in the process. We reason about Split using the following procedure, which is strictly worse than the actual construction but easier to analyze:

Say a leaf node has $k > L$ blocks. Choose L labels uniformly to promote and partition all k blocks by the $(L+1)$-way split of these labels as in Split. Now consider another partition of all k items into $2L/\log(L)$ sorted sub-lists, each of size roughly $k\log(L)/(2L)$. After the $(L+1)$-way partition, we find the index of each of the L promoted labels in this sorted order.

Repeat this process (choosing L labels and partitioning), without creating new nodes, until at least one of the L labels is found in each of the $2L/\log(L)$ sorted sub-lists. By the Coupon Collector's Problem, this event occurs after a single iteration with constant probability, so the expected number of iterations until success is $O(1)$ and the expected server/oracle bandwidth is $O(k)$.

After this process terminates, select one label in each of the $2L/\log(L)$ sorted sub-lists, and then split the leaf node according to these chosen labels; i.e. these $2L/\log(L)$ labels are "actually" promoted to the parent node. The result is $2L/\log(L) + 1$ sibling leaf nodes, each of size at most $k\log(L)/L$. Recurse on the node that contains the label being searched for until the size that node is at most L.

Observe that whenever $L > 4$, $2\log(L)/L < 1$, so each Split call reduces the size of the leaf node by at least a constant factor. More specifically, the maximum number of splits required to reach a leaf of size at most L is at most

$$\frac{\log(k)}{\log(L/\log(L))}, \tag{1}$$

which is at most $2\log(k)/\log(L) = 2\log_L(k)$ whenever $L \geq 16$.

The number of sequential Split calls on a given leaf determines the round complexity of Search, so there are $O(\log_L k)$ rounds of communication starting with a size-k leaf node. Note the total bandwidth over all Split calls is still $O(k)$.

Given these bounds on Split, we examine the resulting tree $\mathcal{T}$, whose structure is populated entirely by calls to Split. Since $k < n$, the total number of Split calls over all m Search operations is at most $2m\log(n)/\log(L)$. Consider the sorted labels in non-leaf nodes of the tree. Each such label is inserted by a Split operation from a leaf, and each Split inserts at most $2L/\log(L)$ labels. Therefore, the total number of labels stored in the sorted, non-leaf portion of the tree $\mathcal{T}$ is at most $mL\log(n)$.

So, more concretely, the sorted labels in the non-leaf nodes of the tree form a B-tree with between L and $L/2$ labels per node. Therefore, the maximum height of this portion of the tree is $\log(mL\log(n))/\log(L/2)$ at most, which implies the height of the entire tree is bounded by

$$O(\log_L(m) + \log_L(\log(n))). \tag{2}$$

Given the bounds on the number of total Split calls in Equation (1) and the height of the tree in Equation (2), we find that the (non-amortized) expected round complexity for every Search operation is $O(\log_L n)$. It remains to count the total bandwidth.

Each round of a Search call involves uploading at most L labels to serve as partition indices, incurring a total bandwidth of

$$O(mL\log_L(n)) \tag{3}$$

for this component. In addition, all the labels in buffers along the search path was sent to the comparison oracle – some more than once. Observe that blocks in buffers only move to a lower buffer, or laterally from leaf nodes to leaf nodes during Split operations. Therefore, the expected total number of blocks in any *non-leaf* buffer, across all Search operations, is

$$O(n \cdot \mathsf{height}(\mathcal{T})) = O(n\log_L(m) + n\log_L(\log(n))). \tag{4}$$

Finally, we consider the total size of all leaf nodes encountered during Search operations. The worst-case scenario for the construction is when all n insertions happen before all m searches, and each search's splits land in the largest remaining leaf node(s). Using the procedure described at the beginning of the analysis, the largest leaf nodes have the following sizes:

- 1 node of size n,
- $2L/\log(L)$ nodes of size at most $n\log(L)/L$ and total size $< n$,
- $4L^2/\log^2(L)$ nodes of size at most $n\log^2(L)/L^2$ and total size $< n$, etc.

Therefore, we encounter the m largest leaf nodes within at most $\log_{L/\log(L)}(m) \leq 2\log_L(m)$ rounds. So the total size of the m largest leaf nodes is bounded by

$$O(n\log_L m). \tag{5}$$

From Equations (3), (4), and (5), we find that the total bandwidth over all $(n + m)$ operations is $O(mL\log_L(n) + n\log_L(m) + n\log_L(\log(n)))$, and Theorem 1 follows. ■

5.2 Security Analysis

Bellare et al. [BKN02] introduced a security notion called indistinguishability under *distinct* chosen ciphertext attack (IND-DCPA). This security notion is strictly weaker than IND-CPA (i.e. semantic security), since the plaintexts chosen by the adversary should be distinct. We remark that a pseudorandom permutation (PRP) meets this security notion [BBO07]. Jumping ahead, we will use a *deterministic* IND-DCPA-secure symmetric encryption to encrypt labels; this weaker security notion is enough for us, since we allow the order of labels to be leaked.

Let $\Pi = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$ be a symmetric encryption scheme. Let $\mathcal{LR}(\cdot, \cdot, b)$ be the function that on inputs m_0, m_1 returns m_b. Consider the following experiment.

Experiment $\mathsf{EXP}^{\mathsf{ind\text{-}dcpa}}_{\mathcal{A}}(\Pi, \lambda, b)$
$k \leftarrow \mathsf{gen}(1^\lambda)$
$b' \leftarrow \mathcal{A}^{\mathsf{enc}_k(\mathcal{LR}(\cdot,\cdot,b))}$
If $b = b'$, return 1; otherwise return 0

We call the adversary $\mathcal{A}$ *admissible* if all left messages of $\mathcal{A}$'s queries are unique and all right messages of $\mathcal{A}$'s are unique. We define the advantage of the adversary $\mathcal{A}$ in the experiment above as:

$$\begin{aligned}\mathbf{Adv}^{\mathsf{ind\text{-}dcpa}}_{\mathcal{A}}(\Pi, \lambda) = \big|\, &\Pr[\mathsf{EXP}^{\mathsf{ind\text{-}dcpa}}_{\mathcal{A}}(\Pi, \lambda, 0) = 1] \\ &- \Pr[\mathsf{EXP}^{\mathsf{ind\text{-}dcpa}}_{\mathcal{A}}(\Pi, \lambda, 1) = 1]\big|.\end{aligned} \tag{6}$$

We say that a symmetric encryption scheme Π is IND-DCPA secure if for any sufficiently large λ and any PPT admissible adversary $\mathcal{A}$, there is a negligible function negl such that $\mathbf{Adv}^{\mathsf{ind\text{-}dcpa}}_{\mathcal{A}}(\Pi, \lambda) \leq \mathsf{negl}(\lambda)$.

Theorem 2. *If Π_ℓ is a deterministic private encryption scheme with IND-DCPA security, and Π_v is an IND-CPA secure private-key encryption scheme, our POPE construction is IND-OCPA secure.*

Proof. We prove security of our scheme using a standard hybrid argument. Given a sequence of messages $(x_1, \ldots, x_i)$, let $\mathsf{first}(x_1, \ldots, x_i)$ denote the position in which the last element x_i appears for the first time in the given sequence. For example, we have $\mathsf{first}(1, 5, 2, 4) = 4$ and $\mathsf{first}(1, 5, 2, 4, 2) = 3$.

We consider the following hybrids:

- HYBRID$_0$: It's $\mathsf{EXP}^{\mathsf{ind\text{-}ocpa}}_{\mathcal{A}}(\mathsf{POPE}, \lambda, 0)$.
- HYBRID$_1$: It's the same as HYBRID$_0$ except that each encrypted value $\bar{v}_i$ is computed by performing $\mathsf{vEnc}(0^{|v_i^0|})$.
- HYBRID$_2$: It's the same as HYBRID$_1$ except that the comparison oracle answers are simulated by associating each encrypted label $\bar{\ell}_i$ with ℓ_i^0 rather than actually performing the decryption.
- HYBRID$_3$: It's the same as HYBRID$_2$ except that each encrypted label $\bar{\ell}_i$ is computed by performing $\mathsf{Enc}(\mathsf{first}(\ell_1^0, \ldots, \ell_i^0))$. The comparison oracle answers are still simulated by associating each encrypted label $\bar{\ell}_i$ with ℓ_i^0 rather than actually performing the decryption.
- HYBRID$_4$: It's the same as HYBRID$_3$ except that each encrypted label $\bar{\ell}_i$ is computed by performing $\mathsf{Enc}(\mathsf{first}(\ell_1^1, \ldots, \ell_i^1))$. The comparison oracle answers are simulated by associating each encrypted label $\bar{\ell}_i$ with ℓ_i^1 rather than actually performing the decryption.
- HYBRID$_5$: It's the same as HYBRID$_4$ except except that each encrypted label $\bar{\ell}_i$ is computed by performing $\mathsf{Enc}(\ell_i^1)$. The comparison oracle answers are still simulated by associating each encrypted label $\bar{\ell}_i$ with ℓ_i^1 rather than actually performing the decryption.

- HYBRID$_6$: It's the same as HYBRID$_5$ except that it uses a real comparison oracle.
- HYBRID$_7$: It's the same as HYBRID$_6$ except that each encrypted value $\bar{v}_i$ is computed by performing $\mathsf{vEnc}(v_i^1)$. Note that HYBRID$_7$ is $\mathsf{EXP}_{\mathcal{A}}^{\mathsf{ind\text{-}ocpa}}(\mathsf{POPE}, \lambda, 1)$.

HYBRID$_0$ and HYBRID$_1$ are computationally indistinguishable due to IND-CPA security of the underlying encryption scheme Π_v; Symmetrically, HYBRID$_6$ and HYBRID$_7$ are also computationally indistinguishable.

For hybrids HYBRID$_1$ and HYBRID$_2$, observe that the server never creates a new ciphertext; it only uses the ciphertexts from the client in its query. Therefore, the comparison oracle can be perfectly simulated by associating $\bar{\ell}_i$ with ℓ_i^0. Therefore, HYBRID$_1$ and HYBRID$_2$ are identically distributed. Symmetrically, HYBRID$_5$ and HYBRID$_6$ are also identically distributed.

HYBRID$_2$ and HYBRID$_3$ are computationally indistinguishable due to IND-DCPA security of the underlying encryption scheme Π_ℓ. In particular, if there is an adversary $\mathcal{A}$ distinguishing the two hybrids, we can construct an adversary $\mathcal{B}$ breaking IND-DCPA security of the underlying encryption algorithm enc as follows:

> The adversary $\mathcal{B}$ sets up HYBRID$_2$/HYBRID$_3$. In particular, it creates the key for the value encryption, and sets up the server state. When $\mathcal{A}$ submits an insertion/search/deletion operation using label ℓ_i^0, $\mathcal{B}$ does the following:
>
> – If the label ℓ_i^0 is not new, copy the old encryption $\bar{\ell}_f$, where $f = \mathsf{first}(\ell_1^0, \ldots, \ell_i^0)$.
>
> – If the label ℓ_i^0 is new, submit $(\ell_i^0, \mathsf{first}(\ell_1^0, \ldots, \ell_i^0))$ to IND-DCPA experiment and receives a ciphertext encrypting one of the two messages. Now, use this ciphertext as ℓ_i.
>
> $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs.

Note that $\mathcal{B}$ is admissible adversary for IND-DCPA experiment. Moreover, the comparison oracle behaves identically in both hybrids, so it holds that $\mathcal{A}$'s advantage of distinguishing HYBRID$_2$ and HYBRID$_3$ is exactly the same as $\mathcal{B}$'s advantage of breaking IND-DCPA security of the label encryption scheme. Therefore, due to IND-DCPA security of the label encryption scheme, HYBRID$_2$ and HYBRID$_3$ are computationally indistinguishable. Symmetrically, HYBRID$_4$ and HYBRID$_5$ are computationally indistinguishable.

For HYBRID$_3$ and HYBRID$_4$, observe that the answers from the comparison oracle in both hybrids are identically distributed. This is because the adversary is admissible; i.e. the ordering of labels in both hybrids is the same, even though the actual labels may be different. Moreover, for all i, we have $\mathsf{first}(\ell_1^0, \ldots, \ell_i^0) = \mathsf{first}(\ell_1^1, \ldots, \ell_i^1)$. Therefore, the hybrids are identically distributed. ■

If a randomized label encryption scheme is used, the scheme needs to be IND-CPA.

Theorem 3. *If Π_ℓ and Π_v are both IND-CPA secure private-key encryption schemes, our POPE is IND-OCPA secure.*

6 FH-POPE: Hiding Frequency with POPE

Our POPE construction achieves a notion of IND-OCPA security (from the selective model of OPE) fitted to our *adaptive* POPE setting. However, a recent attack [NKW15] has demonstrated that this level of security is sometimes insufficient (at least in the OPE setting), by showing how the revealed order can be used to statistically recover a significant amount of plaintext data stored in a medical database by specifically targeting the OPE encoded fields.

To address the above issue, [Ker15] recently proposed a stronger security notion for OPE, called *indistinguishability under frequency-analyzing ordered chosen plaintext attack* (IND-FA-OCPA). This definition is modeled around the notion of a *randomized order* of some sequence X of (possibly non-distinct) elements x_i. Intuitively, a "randomized order" Y (some permutation of $[n]$ for n-element sequences) is one of the possible total order extensions of a given partial order on a sequence.

For example, the randomized order of the sequences $X_1 = (1, 2, 3, 4)$ and $X_2 = (2, 3, 4, 5)$ could only be $Y_1 = (1, 2, 3, 4)$ (meaning "first in sorted order was inserted first," "second in sorted order . . . ," and so on) because X_1, X_2 began totally ordered, but the sequence $X_3 = (1, 2, 2, 3)$ has two possible extensions to total orders, namely $Y_1 = (1, 2, 3, 4)$ and $Y_2 = (1, 3, 2, 4)$. On the other hand, note that for any particular randomized order, e.g. $Y_1 = (1, 2, 3, 4)$, there are *many* sequences that could map onto it (depending only on the domain of the sequence and the constraints imposed by "known" partial order information on the sequence). For example, the sequences $(1, 1, 1, 1)$, $(10, 20, 30, 40)$, $(10, 20, 20, 40)$ and $(2, 2, 17, 17)$ all share Y_1 as a possible "randomized order," but the sequence $(4, 3, 2, 1)$ does not.

Given this idea of a "randomized order," the definition of [Ker15] (IND-FA-OCPA) asks that for any two sequences of plaintexts X_1, X_2, if X_1, X_2 share *some* randomized order Y, then the frequency-hiding (FH-OPE) encodings of X_1, X_2 should be indistinguishable. For example, sequences $X_1 = (1, 1, 1, 1)$ and $X_2 = (10, 20, 30, 40)$ should be indistinguishable under FH-OPE encoding, because they share order $Y = (1, 2, 3, 4)$. For the same reason, $X_1 = (10, 20, 20, 30)$ and $X_2 = (10, 20, 30, 40)$ should be indistinguishable under FH-OPE encoding.

To turn an OPE scheme (for sequences of only *distinct* elements sharing an order) into an FH-OPE scheme, consider adding a small, random fractional component to the OPE-ordered field during encoding, e.g. $X_1 = (1, 1, 2, 2)$ becomes e.g. $X_1' = (1.12, 1.36, 2.41, 2.30)$ (which *randomly* maps X_1 to the total ordering $Y = (1, 2, 4, 3)$), and then X_1' is encoded under the OPE scheme. In [Ker15], this type of scheme is shown IND-FA-OCPA secure in the *programmable random oracle model* by programming in appropriate randomness to "break ties in the correct direction" on repeated plaintext values (in order to land on a common randomized order Y) to ensure indistinguishability of X_1 and X_2's encodings.

6.1 Considerations in Defining Frequency-Hiding POPE

Differences between FH-OPE and FH-POPE. The motivating example for frequency-hiding security is a database that stores a large number n of encodings for which the underlying label space $\mathcal{L} = \{\ell_1, ..., \ell_T\}$ is small, i.e. $T \ll n$. [Ker15] suggested a setting where each label is either ℓ_1 = "female" (F) or ℓ_2 = "male" (M), with the sequence (F, F, M, M) ideally encoded as, say, $(1, 2, 3, 4)$. Examining only $(1, 2, 3, 4)$ does not reveal if the underlying sequence was originally (F, F, F, F), (F, F, F, M), (F, F, M, M), (F, M, M, M), or (M, M, M, M) — yet, you can still search for all female entries by range query for $[1, 2]$, or all male entries by range query for $[3, 4]$, so functionality is maintained as well.

However, defining frequency-hiding for FH-POPE in an identical way to [Ker15] for FH-OPE raises a number of challenges. We highlight a few of these differences here.

First, one issue is that the IND-FA-OCPA security game in the OPE setting considers *no semantic information* on the FH-OPE encoding $(1, 2, 3, 4)$ that would be gained by seeing many queries for, say, $[1, 2]$, $[3, 4]$, and $[1, 4]$ (but not any for, e.g. $[1, 1]$). In the POPE setting, this is analogous to defining a security game that only includes Insert operations but no Search (or Delete) queries! Because our POPE scheme leaks no order information prior to the first Search query, it turns out that (formally) IND-OCPA for POPE implies IND-FA-OCPA (in the sense of FH-OPE) *with no extra modification*.

Of course, this conclusion is not fully satisfying; we haven't yet said anything about frequency-hiding of our POPE scheme! This raises a second concern: Is it even possible to achieve frequency-hiding in the *adaptive query* setting of the IND-OCPA definition for POPE? Consider, in the complexity-theoretic model, the following adversarial choice of sequences (possibly, with repeated elements) in the style of IND-OCPA for POPE. The first sequence X_1 is $(1, 1, 1...)$, while the second sequence $X_2 = (x_1, x_2, ...)$ is statefully generated with a random coin flip for every i-th element $x_i \in X_2$. If the i-th coin flip is heads, then $x_i = i$; if tails, $x_i = -i$.

Note that X_2 is a completely distinct sequence, so its (unique) randomized order will only depend on x_i's sign as determined by the i-th coin flip. On the other hand, the sequence X_1 can take on exponentially-many total orders depending only the *randomness used to encode*, independent of the exact elements of X_1. Therefore, for *fixed* sequences X_1, X_2 of length n as above, the randomized order of X_1 induced by a *real* encoding will differ from the unique order of X_2 with probability about $(1 - 2^{-n}) \approx 1$, unless the randomness used to encode X_1 is explicitly correlated with the *adversarially chosen* sequence X_2 itself in the proof.

This issue is partially avoided in [Ker15], since the two sequences X_1, X_2 must be fully specified at once, before the FH-OPE system is run. (In the analogous POPE notion, the adversary adaptively chooses queries and sees the outcome of each before choosing again, which we can view as the coin flip game when the i-th coin is not fixed until the i-th insertion.) Yet even with the advantage of a *selective* choice of sequences X_1, X_2, the security proof of [Ker15] still needs that the *randomness* for X_1's encoding is correlated with the *sequence* X_2. This requires *programming* the random oracle.

FH-POPE is easy with the random oracle. Briefly, we sketch an informal justification for why using a programmable random oracle in the manner of [Ker15], particularly in the context of POPE or FH-POPE, appears to be too strong of an assumption. The idea is that, given the ability to program a random oracle, we can do more than break ties (and hide local frequency differences) when deciding a randomized order. In the context of POPE, we can easily achieve something much stronger, resembling a "non-committing" notion of POPE security. In particular, consider the following encryption scheme:

- $\mathsf{Enc}_k(\ell_i)$: Choose u and r_i at random. Compute $\ell'_i \leftarrow \ell_i \| r_i$, where the randomness r_i is used to break the tie. We call ℓ'_i an *augmented label*. The output ciphertext is $\big(f_k(u), H(u) \oplus \ell'_i\big)$, where f_k is a pseudorandom permutation and H is the random oracle.

The above encryption is non-committing as in [Nie02]; that is, given any ciphertext c, the simulator may claim that the ciphertext c encrypts whatever augmented label ℓ'_i it would like, by programming the random oracle H.

With the above label encryption scheme, consider the following game, played between an adversary and simulator, loosely modeled after our IND-OCPA definition for POPE (i.e. Definition 1, but strengthened):

First, some FH-POPE scheme is initialized (e.g. by initializing our underlying POPE scheme using the above label encryption). Then in as many iterations as it would like, a stateful adversary $\mathcal{A}$ requests its choice of operations be performed. A crucial difference is that whenever $\mathcal{A}$ requests an Insert, it does not specify which label should be encrypted at all. (In contrast, frequency-hiding for OPE requires specifying two sequences X_0, X_1 that demand one of two labels are encrypted. Our simulator here is at a *significant disadvantage*.) Nonetheless, the simulator $\mathcal{S}$ responds by choosing *not necessarily distinct* junk strings x_i for each i-th insertion query, then encrypting x_i and inserting it according to our (distinct-element) POPE.

For Search (resp. Delete) queries, the adversary not only specifies plaintext end-point labels ℓ_1, ℓ_2 for a range query, but may also name its choice of constraint on specific ciphertext movements within the FH-POPE data structure. The simulator then implements some Search execution following any constraint specified by the adversary, and making arbitrary (but consistent) decisions about the other partial orders that must be induced between ciphertexts, independent of their underlying plaintexts x_i, in order to process each given query.

Eventually, $\mathcal{A}$ stops making queries, and demands that $\mathcal{S}$ *open* all ciphertexts in the transcript of accesses by providing the private symmetric keys K used by $\mathcal{S}$ to encrypt ciphertexts in the course of the game. In particular, the adversary will check if $\mathcal{S}$ was able to "correctly guess" appropriate underlying plaintexts $\ell_1, ..., \ell_n$ whose order information is consistent with the movement of ciphertexts throughout the game.

We claim that intuitively, since $\mathcal{S}$ was given no information about the future ciphertext movement when it had to commit to some ciphertext (and thus, to some choice of initial plaintext x_i), the simulator should only be able to suceed with negligible probability in this situation. However, by instantiating the POPE encryption with a non-committing encryption scheme (using the full power of programming the random oracle), the simulator $\mathcal{S}$ can first find (any) valid labeling ℓ_i of the ciphertexts' plaintexts that is consistent with the game's transcript, then *open* the ciphertexts to these labels $\ell_i \neq x_i$ in the view of the adversary $\mathcal{A}$, guaranteeing success w.h.p. We take this as evidence that definitions of frequency-hiding in the random oracle model (at least in the case of POPE) may not approximate the real world well.

6.2 Security of Frequency-Hiding POPE

We define frequency-hiding security IND-FA-POCPA (indistinguishability under frequency-analyzing partial-order chosen plaintext attack) for POPE *without using the random oracle*. What our definition captures is the following: Encryption of non-distinct labels implicitly chooses a randomized *partial* order, and the ciphertexts reveal nothing more than this order.

The IND-FA-POCPA definition considers an experiment with two adversary non-communicating algorithms $\mathcal{A}_0$ and $\mathcal{A}_1$ which proceeds as follows:

Experiment $\mathsf{EXP}^{\text{ind-fa-pocpa}}_{\mathcal{A}_0,\mathcal{A}_1}(\mathsf{POPE}, \lambda, b)$:

1. $\mathcal{A}_0$ adaptively chooses operations where the labels may be *non-distinct*. That is, each time $\mathcal{A}_0$ chooses an operation, the environment performs the operation and sends the view of the operation to $\mathcal{A}_0$, based on which $\mathcal{A}_0$ chooses the next operation.
2. The environment determines the partial order that the above operations leak, and gives this order to $\mathcal{A}_1$.
3. $\mathcal{A}_1$ adaptively chooses operations, where the type of each operation should be the same as the counterpart in the above, and the labels should be *distinct* and *subject to the given partial order*. The operations are performed using the *same randomness* for the operations performed in step 1).
4. The environment outputs the view of operations for $\mathcal{A}_b$.

We require that the following ensembles should be computationally indistinguishable:

$$\{\mathsf{EXP}^{\text{ind-fa-pocpa}}_{\mathcal{A}_0,\mathcal{A}_1}(\mathsf{POPE}, \lambda, 0)\}_\lambda$$
$$\approx_c \{\mathsf{EXP}^{\text{ind-fa-pocpa}}_{\mathcal{A}_0,\mathcal{A}_1}(\mathsf{POPE}, \lambda, 1)\}_\lambda.$$

We note a few aspects of the above experiment.

- The partial order that $\mathcal{A}_0$'s operations leak is defined as follows:
  - (Base case) If ℓ_i and ℓ_j were directly compared through the comparison oracle, and the oracle said ℓ_i is less than ℓ_j, we have $\ell_i \prec \ell_j$.

  - (Transitive closure) If there is ℓ_k such that $\ell_i \prec \ell_k$ and $\ell_k \prec \ell_j$, we have $\ell_i \prec \ell_j$.
  - Otherwise, ℓ_i and ℓ_j are incomparable.

- The same random coins are used for $\mathcal{A}_0$'s operation sequence and $\mathcal{A}_1$'s. Here, the randomness corresponds to what the buffer tree uses when it choose labels to promote to the parent node. This randomness actually affects which labels to be compared through the comparison oracle, and using the same randomness enforces the same-indexed labels to be compared in both operation sequences.
- We require $\mathcal{A}_1$ use distinct labels. As discussed above, in order to achieve indistinguishability when $\mathcal{A}_1$ uses nondistinct labels, we need to use stronger assumptions such as the programmable random oracle.

6.3 Construction of Frequency-Hiding POPE

We give a construction for a frequency-hiding POPE scheme by augmenting our original POPE construction. The main idea is that each label encryptions add a random fraction to the input label to break the tie.

Let $(\Pi'_\ell, \Pi_v, \mathsf{InitState}, \mathsf{Cmp}', \mathsf{Insert}, \mathsf{Search}, \mathsf{Delete})$ be an IND-OCPA secure POPE, for the symmetric encryption scheme $\Pi'_\ell = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$. Our augmented frequency-hiding POPE construction is as follows:

$$(\Pi_\ell, \Pi_v, \mathsf{InitState}, \mathsf{Cmp}, \mathsf{Insert}, \mathsf{Search}, \mathsf{Delete}),$$

where $\Pi_\ell = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. That is, we only change the label encryption scheme and the comparison oracle.

Label encryption. The label encryption is slightly changed so that it randomly breaks the tie. In particular, the algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ for encrypting labels are specified below.

- $\mathsf{KeyGen}(1^\lambda)$: Compute $k \leftarrow \mathsf{gen}(1^\lambda)$ and return k.
- $\mathsf{Enc}_k(\ell)$: Choose $r \leftarrow \{0,1\}^\lambda$, compute $\bar{\ell} \leftarrow \mathsf{enc}(\ell')$ with $\ell' = \ell \| r$, and return $\bar{\ell}$.
- $\mathsf{Dec}_k(\bar{\ell})$: Compute $\ell' \leftarrow \mathsf{dec}_k(\bar{\ell})$ and return ℓ by removing λ rightmost bits from ℓ'.

Comparison oracle. The comparison oracle compares the encrypted labels by considering *not only the label but also the tie-breaking random value*.

- $\mathsf{Cmp}(k, (\bar{\ell}_1, \ldots \bar{\ell}_q))$: This algorithm works essentially the same as that of the original POPE scheme, except that the comparison of the elements are based on $\mathsf{dec}_k(\bar{\ell}_i)$s instead of $\mathsf{Dec}_k(\bar{\ell}_i)$s.

6.4 Analysis of Frequency-Hiding POPE

Security. Security is easy to see:

Theorem 4. *The above POPE is IND-FA-POCPA secure.*

Proof. Observe that the augmented labels (i.e., the label and the tie-breaking random value) for $\mathcal{A}_0$ has the same ordering as those for $\mathcal{A}_1$. Moreover, each sequence has distinct augmented labels. Therefore due to IND-OCPA security of the original POPE construction, $\mathsf{EXP}^{\mathsf{ind\text{-}fa\text{-}pocpa}}_{\mathcal{A}_0,\mathcal{A}_1}(\mathsf{POPE}, \lambda, 0)$ is computationally indistinguishable from $\mathsf{EXP}^{\mathsf{ind\text{-}fa\text{-}pocpa}}_{\mathcal{A}_0,\mathcal{A}_1}(\mathsf{POPE}, \lambda, 1)$ augmented POPE construction. ■

Partial Order Leakage. A feature of our IND-FA-POCPA definition, as well as our POPE construction, is that the server only learns a *partial* order of the ciphertexts, rather than a total order as is revealed by existing OPE constructions.

Recall that a *strict partial order* $\prec$ on a set of elements S is isomorphic to a directed acyclic graph, closed under transitive closure, whose nodes are elements of S and whose edges encode the binary relation. A total order (such as the *randomized order* of [Ker15]) on n items always has $\binom{n}{2}$ edges. In any partial order, two elements $x, y \in S$ are said to be *incomparable* iff neither $x \prec y$ nor $y \prec x$. In a total order, no pair of elements is incomparable.

Therefore, a scheme providing IND-FA-POCPA is *more secure* if the partial order leaked in step (2), is truly a partial order and not a total order. The strength of such a scheme can be measured by the *number of pairs of incomparable elements* in the partial order leaked by the experiment.

Theorem 5. *Whenever adversary $\mathcal{A}_0$ chooses n insertions and m range queries, then the partial order revealed by the POPE construction above leaves at least $\Omega(n^2/(mL\log_L n))$ pairs of incomparable elements.*

Proof. We model the server's view of the ciphertext ordering as some k ciphertexts whose order is completely known, and where the remaining $n-k$ ciphertexts are partitioned into one of $k+1$ buckets according to the k ordered ciphertexts. Essentially, this is a worst-case scenario where all internal node buffers in the POPE tree are empty, the total size of all internal node sorted lists is k, and the remaining $n-k$ ciphertexts reside in leaf node buffers.

From theorem 1, the total rounds of communication with the comparison oracle after n insertions and m range queries is $O(m\log_L n)$. From the construction, each round of communication with the comparison oracle can add at most L new ciphertexts to those whose sorted order is completely known.

Therefore, in the worst case, the server has $k = O(mL\log_L n)$ ciphertexts in its sorted order, and at least $\lfloor (n-k)/(k+1) \rfloor$ ciphertexts in each unsorted bucket. Each bucket contains $\Omega((n/k)^2)$ incomparable items, for a total of $\Omega(n^2/k)$ incomparable pairs.

Observe that this is the worst case; having all $n-k$ unsorted ciphertexts in the same bucket, for instance, would result in a larger number of incomparable pairs. ■

In our construction, we require that $mL \leq n$ and $\log_L n \in O(1)$, so that $n^2/(mL\log_L n)$ will be at least $\Omega(n)$ incomparable pairs. By contrast, all previous OPE constructions impose a total ordering on the ciphertexts (at best), leaving 0 incomparable pairs of elements.

Performance. For the original POPE construction, an IND-DCPA secure encryption scheme can simply be instantiated with a pseudorandom permutation (PRP) by treating each value as one block message.

However, since the frequency-hiding POPE construction needs to encrypt the label and the tie-breaking random value, the plaintext becomes two blocks, and a simple PRP is not enough for instantiating IND-DCPA secure encryption in this case. An easy solution to this issue is just using CTR or CBC mode that ensures IND-CPA security. However, now the length of the ciphertext is three blocks. This will incur 3x blow-up in terms of communication complexity.

A better solution in this case to utilize the already-existing tie-breaking random value and simulate a mode of operation. In particular, let f be a PRP. To encrypt:

- $\mathsf{enc}_k(m\|r)$:
  return $(f_k(r), f_k(r+1) \oplus m)$.

Here, the encryption simulates the CTR mode using the tie-breaking random value, but one can use other secure mode of operations. To decrypt:

- $\mathsf{dec}_k(c_1, c_2)$:
  compute $r \leftarrow f_k^{-1}(c_1), m \leftarrow f_k(r+1) \oplus c_2$
  return $m \| r$.

Note the probability that the inputs to f_k have a collision is negligible, since each input is a random value r or $r+1$. Therefore, IND-DCPA security follows from security of PRP. With this construction, we will have only 2x blow-up in terms of communication complexity.

Enforcing consistent output. Suppose the following labels have been inserted: $(1, 2, 2, 3)$.

Suppose in addition that the client would search labels that are at least 2 and at most 4. When our frequency-hiding POPE is used, due to the random tie-breaking property, the client may get (3) or $(2, 3)$ or $(2, 2, 3)$ depending how the tie would be broken. This feature may be inconvenient to the client.

With two additional bits, this problem can be fixed easily. We introduce three different two-bit tags, i.e., $\tau_l = 00, \tau_m = 01, \tau_r = 10$ for left, middle, right positions. When a label is inserted, the middle tag τ_m is attached at the end of the label. For a search/delete query, the left tag τ_l is used for the left label, and the right tag τ_r for the right label.

Continuing the prior example, the server now holds labels:

$$1\|01,\ 2\|01,\ 2\|01,\ 3\|01.$$

The client will use labels $(2\|00, 4\|10)$ for search, and therefore it will get the desired result $(2\|01, 2\|01, 3\|01)$.

7 Evaluation

We have made a proof-of-concept implementation of our POPE scheme in order to test the practical utility of our new approach. The code is written in Python3 and our tests were performed using a single core on a machine with an Intel Xeon E5-2440 2.4 GHz CPU and 72GB available RAM. Our implementation follows the details presented in Section 4, with a single shared key for the client and comparison Oracle. The symmetric cipher used is 128-bit AES, as provided by the PyCrypto library. The full source code of our implementation is available upon request.

While we performed experiments on a wide range of database sizes and number of range queries, our "typical" starting point is one million insertions and one thousand range queries. This is the same scale as recent work in the databases community for supporting range queries on outsourced data [LLWB14], and would therefore seem to be a good comparison point for practical purposes.

7.1 Experimental setup

Our tests measured communication (in terms of rounds and total ciphertexts transferred) between the server and the comparison oracle, and total server-plus-oracle computation time. We did not measure the communication between client and server, since this is inherent in the operation being performed and would be the same for any alternative implementation.

For a fair comparison to prior work, we also implemented the mOPE scheme of [PLZ13] in Python3 along with our implementation of POPE. We followed the description in their work, using a B-tree with at most 4 items per node to store the encryptions. To get a fair comparison, we used the same framework as our POPE experiments, with a comparison oracle that receives sorting and partitioning requests from the server. In the case of mOPE, each round of communication consisted of sending a single B-tree node's worth of ciphertexts, along with one additional ciphertext to be encoded, and receiving the index of the result within

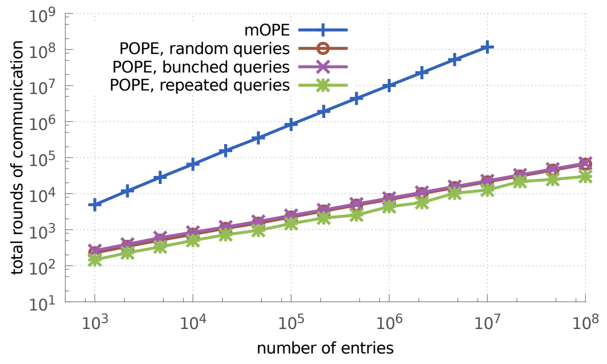

Figure 5: Total rounds of communication for POPE and mOPE, plotted in log/log scale according to total number of insertions n. Lower is better. The number of range queries in all cases was $\sqrt{n}$.

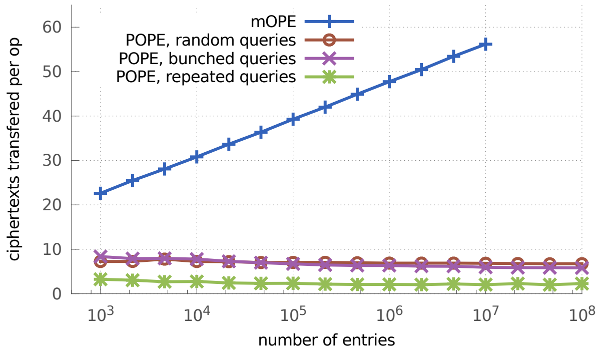

Figure 6: Amortized communication costs for POPE and mOPE, according to total number of insertions n. Lower is better. The number of range queries in all cases was $\sqrt{n}$.

that sorted order. We acknowledge that our implementation is likely less tuned for efficiency than that of the original authors, but it gives a fair comparison to our own proof-of-concept implementation of POPE. It is also important to note that the communication costs we are measuring depend only on the algorithm and not on the efficiency of the implementation.

In our main experiments, we performed a number of insertions and range queries on random bigrams of English words, encrypted using AES-128. The actual size of each range being searched was, on average, 100 database entries. While the distribution of searches does not affect the running time of mOPE, for POPE we varied among three distributions of the random range queries: uniformly distributed queries, either uniformly distributed among the searches or all "bunched" at the end after all insertions; or a single, repeated query, performed at random intervals among the insertions.

According to our theoretical analysis, the "bunched" distribution should be the worst-case scenario, although in practice we did not see much difference in performance between bunched or random queries. Though as expected, we observed improved performance for the repeated query case.

7.2 Overview of results

A summary of the raw data from our preliminary experiments may be found in Appendix A.

In our experiments, we varied the total database size between one thousand and 100 million entries, each time performing roughly $m = n^{1/2}$ range queries and with $L = n^{1/4}$ local storage on the comparison oracle. That is, $\epsilon = 0.25$ in these experiments. The size of each range being queried was randomly selected from a geometric distribution with mean 100; that is, each range query returned on average 100 results.

We note that, for 1 million entries and using our proof-of-concept Python implementation without parallelization, we achieved over 55,000 operations per second with POPE vs. less than 2,000 operations per second for mOPE.

We were able to run experiments with POPE up to 100 million entries, limited only by the storage space available on our workstation. We observed no significant change in per-operation performance after one million entries, indicating our construction should scale well to even larger datasets.

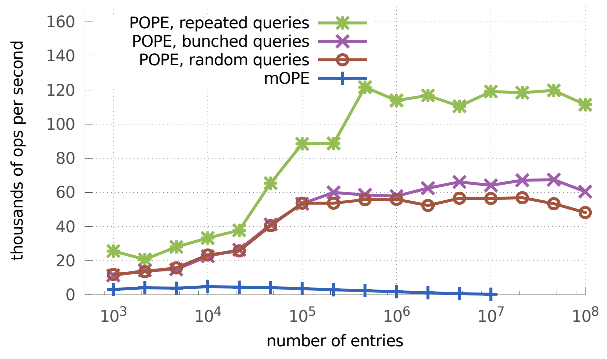

Figure 7: Operations performed per second for POPE and mOPE. Higher is better. The number of range queries in all cases was $\sqrt{n}$.

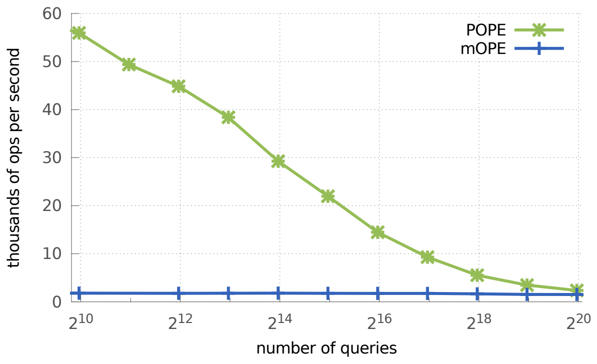

Figure 8: Degradation in POPE performance with increasing number of queries, measured in operations per second. Higher is better. In all experiments, the number of insertions n was fixed at 1 million, and the client-side storage at $L = 32$. For these choices, $2^{10} \approx \sqrt{n}$ queries is as shown in prior figures, and our $O(1)$-cost analysis holds up to $m \approx 2^{15}$.

7.3 Experimental communication costs

Figures 5 and 6 show the communication costs, the total number of rounds of communication, and the average number of ciphertexts transferred per operation. The number of insertions n is shown in the plots, and for each experiment we performed $m = \sqrt{n}$ searches allowing $L = n^{1/4}$ entries stored in temporary memory on the client.

From these figures, it is clear that the round complexity for POPE, which is constant *per range query*, is several orders of magnitude less than that of mOPE. Furthermore, when averaged over all operations, the number of ciphertexts transferred per operation for POPE is roughly 7 in the worst case, whereas for mOPE this increases logarithmically with the database size.

7.4 Experimental computation costs

The number of operations performed per second, for our main experiments with n insertions, $m = \sqrt{n}$ range queries, and $L = n^{1/4}$ client-side storage, are presented in Figure 7. For POPE, the performance increases until roughly 1 million entries, after which the per-operation performance holds steadily between 50,000 operations per second with random, distinct queries, and 110,000 operations per second with a single, repeated query.

Our POPE construction is well-suited especially to problems with many more insertions than range queries; indeed we require that $mL \leq n$ for our performance guarantees to be valid. Figure 8 shows the effects of varying number of range queries on POPE performance. While our theoretical guarantees hold only when $mL < n$, we observed competitive performance to mOPE even when performing $m = n$ range queries, although the POPE performance clearly degrades with the number of queries performed.

Acknowledgments. Daniel Roche's work is supported in part by Office of Naval Research (ONR) award N0001415WX01532. Daniel Apon's work is supported in part by NSF awards #1111599, #1223623, and #1514261. Seung Geol Choi's work is supported in part by ONR awards N0001415WX01532 and N0001415WX01232. Arkady Yerukhimovich's work is sponsored by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, recommendations and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

References

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 131–148. Springer, March 2014.

[AKSX04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 563–574, 2004.

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[AV87] Alok Aggarwal and JeffreyScott Vitter. The i/o complexity of sorting and related problems. In Thomas Ottmann, editor, *Automata, Languages and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 467–478. Springer Berlin Heidelberg, 1987.

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 535–552. Springer, August 2007.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, pages 224–241, 2009.

[BCO11] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 578–595, 2011.

[BCOP04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 506–522, 2004.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BHJP14] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Comput. Surv.*, 47(2):18:1–18:51, 2014.

[BKN02] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In Vijayalakshmi Atluri, editor, *ACM CCS 02*, pages 1–11. ACM Press, November 2002.

[BLR$^+$15] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 563–594, 2015.

[BW07] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 535–554, 2007.

[CDG$^+$06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

[CGKO06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 79–88, 2006.

[CJJ$^+$13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373, 2013.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220, 2007.

[DvDF$^+$16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. Theory of Cryptography Conference, TCC '16, 2016. cf. Cryptology ePrint Archive, Report 2015/005.

[FJK$^+$15] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive, Report 2015/927, 2015. `http://eprint.iacr.org/`.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 449–458, 2015.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 174–187, 1986.

[Goh03] Eu-Jin Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Ker15] Florian Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 656–667, 2015.

[KGM$^+$14] Jeremy Kepner, Vijay Gadepally, Peter Michaleas, Nabil Schear, Mayank Varia, Arkady Yerukhimovich, and Robert K. Cunningham. Computing on masked data: a high performance method for improving big data veracity. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, pages 1–6, 2014.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 965–976, 2012.

[KS14] Florian Kerschbaum and Axel Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 275–286, 2014.

[LLWB14] Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *Proc. VLDB Endow.*, 7(14):1953–1964, October 2014.

[MMB15] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication oram with small blocksize. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 862–873, New York, NY, USA, 2015. ACM.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, August 2002.

[NKW15] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015.

[PKV$^+$14] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 359–374, 2014.

[PLZ13] Raluca A. Popa, Frank H. Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 463–477, 2013.

[PRZB11] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 85–100, 2011.

[PZB15] Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan. Guidelines for using the cryptdb system securely. Cryptology ePrint Archive, Report 2015/979, 2015. `http://eprint.iacr.org/`.

[SBC$^+$07] Elaine Shi, John Bethencourt, Hubert T.-H. Chan, Dawn Xiaodong Song, and Adrian Perrig. Multi-dimensional range query over encrypted data. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 350–364, 2007.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, December 2011.

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 44–55, 2000.

[Thea] The Apache Software Foundation. Accumulo. https://accumulo.apache.org/. Accessed: 2015-09-24.

[Theb] The Apache Software Foundation. Cassandra. https://cassandra.apache.org/. Accessed: 2015-09-24.

[Thec] The Apache Software Foundation. Hbase. http://hbase.apache.org/. Accessed: 2015-09-24.

[WCS15] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.

[WGA06] D. Westhoff, J. Girao, and M. Acharya. Concealed data aggregation for reverse multicast traffic in sensor networks: Encryption, key distribution, and routing adaptation. *Mobile Computing, IEEE Transactions on*, 5(10):1417–1431, Oct 2006.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

A Experimental Data

Table 9 presents raw data from our preliminary experiments. We note that our experiments with mOPE stop at databases of 10 million items (whereas we have data for POPE up to 100 million items), because the mOPE experiments became prohibitively slow (over a day) on our workstation past this size, performing less than one operation per second. POPE experiments for 100 million items took about 30 minutes each, with about 40,000 operations per second. The size of our experiments is limited primarily by the size of statistically-sound, random data that we are able to synthesize and test on in a reasonable timeframe, which is primarily constrained by our workstation's RAM rather than POPE's performance on even larger datasets.

	insertions	range queries	local storage	comm. rounds	comm. size	time (seconds)
mOPE [PLZ13]	1 000	32	6	4 943	23 334	0.327
	2 154	46	7	11 951	56 049	0.530
	4 642	68	8	28 277	132 321	1.209
	10 000	100	10	65 805	311 042	2.117
	21 544	147	12	156 079	730 241	4.851
	46 416	215	15	352 882	1 695 090	11.205
	100 000	316	18	834 320	3 940 783	27.666
	215 443	464	22	1 917 495	9 069 763	73.039
	464 159	681	26	4 406 712	20 875 512	191.690
	1 000 000	1 000	32	10 058 650	47 772 258	554.060
	2 154 435	1 468	38	22 816 305	108 756 600	1 788.217
	4 641 589	2 154	46	52 415 869	248 152 448	7 139.188
	10 000 000	3 162	56	117 191 315	561 980 070	30 847.342
	insertions	range queries	local storage	comm. rounds	comm. size	time (seconds)
POPE (this paper)	1 000	32	6	229	7 484	0.086
Random queries,	2 154	46	7	345	16 001	0.163
uniformly distributed.	4 642	68	8	514	36 686	0.298
	10 000	100	10	749	73 229	0.433
	21 544	147	12	1 094	156 706	0.843
	46 416	215	15	1 519	328 039	1.151
	100 000	316	18	2 243	707 469	1.867
	215 443	464	22	3 279	1 523 650	4.020
	464 159	681	26	4 794	3 240 471	8.337
	1 000 000	1 000	32	6 856	6 898 429	17.888
	2 154 435	1 468	38	10 073	14 865 016	41.170
	4 641 589	2 154	46	14 693	32 039 838	82.054
	10 000 000	3 162	56	21 358	68 623 791	177.157
	21 544 347	4 642	68	30 975	146 590 749	378.576
	46 415 888	6 813	83	45 013	312 980 862	868.758
	100 000 000	10 000	100	65 771	676 386 914	2 072.918
	insertions	range queries	local storage	comm. rounds	comm. size	time (seconds)
POPE (this paper)	1 000	32	6	264	8 633	0.091
Random queries,	2 154	46	7	396	17 398	0.154
bunched at the end.	4 642	68	8	602	37 554	0.316
	10 000	100	10	849	78 762	0.447
	21 544	147	12	1 190	158 063	0.821
	46 416	215	15	1 697	325 877	1.131
	100 000	316	18	2 495	674 486	1.880
	215 443	464	22	3 548	1 395 883	3.600
	464 159	681	26	5 173	2 953 508	7.943
	1 000 000	1 000	32	7 512	6 363 063	17.311
	2 154 435	1 468	38	10 867	13 372 696	34.492
	4 641 589	2 154	46	15 752	28 724 419	70.218
	10 000 000	3 162	56	22 769	59 631 557	155.939
	21 544 347	4 642	68	33 170	126 484 416	320.787
	46 415 888	6 813	83	48 205	271 964 129	688.054
	100 000 000	10 000	100	70 082	582 794 073	1 653.199
	insertions	range queries	local storage	comm. rounds	comm. size	time (seconds)
POPE (this paper)	1 000	32	6	146	3 358	0.040
A single random query,	2 154	46	7	229	6 688	0.106
repeated at uniform intervals.	4 642	68	8	334	12 746	0.168
	10 000	100	10	502	27 774	0.303
	21 544	147	12	726	52 790	0.573
	46 416	215	15	958	108 623	0.713
	100 000	316	18	1 482	237 194	1.134
	215 443	464	22	2 122	468 798	2.433
	464 159	681	26	2 564	969 315	3.819
	1 000 000	1 000	32	4 352	2 111 713	8.787
	2 154 435	1 468	38	5 690	4 432 591	18.454
	4 641 589	2 154	46	10 311	10 237 319	42.027
	10 000 000	3 162	56	12 618	20 569 205	83.989
	21 544 347	4 642	68	21 691	48 965 999	181.853
	46 415 888	6 813	83	24 866	93 983 082	387.506
	100 000 000	10 000	100	30 002	228 018 130	897.771

Figure 9: Experimental communication costs. The actual data inserted and searches performed were exactly the same across each row. Communication costs are measured only between the server and the comparison oracle, and communication size counts the number of ciphertexts transferred.