Practical Privacy-Preserving Range and Sort Queries with Update-Oblivious Linked Lists

Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir

College of Computer and Information Science
Northeastern University, Boston MA-02115, USA
{blass,travism,noubir}@ccs.neu.edu

Abstract. We present RASP, a new protocol for privacy-preserving range search and sort queries on encrypted data in the face of an untrusted data store. The contribution of RASP over related work is twofold: first, RASP improves privacy guarantees by ensuring that after a query for range $[a,b]$ any new record added to the data store is indistinguishable from random, even if the new record falls within range $[a,b]$. Second, RASP is highly practical, abstaining from expensive asymmetric cryptography and bilinear pairings. Instead, RASP only relies on hash and block cipher operations. The main idea of RASP is to build upon a new *update-oblivious* bucket-based data structure. We allow for data to be added to buckets without leaking into which bucket it has been added. As long as a bucket is not explicitly queried, the data store does not learn anything about bucket contents. Furthermore, no information is leaked about data additions following a query. Besides formally proving RASP's privacy, we also present a practical evaluation of RASP using Amazon Dynamo.

1 Introduction

Outsourcing data to cloud stores has become a popular strategy for businesses, as cloud properties like scalability and flexibility allow for significant costs savings. However, cloud infrastructures cannot always be trusted, due to, for example, hacker and insider attacks [10, 18]. While encryption of outsourced data protects against many privacy threats in cloud scenarios, it renders subsequent operations on data (i.e., data analysis) impractical and inefficient. Although fully homomorphic encryption (FHE) offers an elegant solution to perform operations and data analysis on encrypted data, today's techniques are still impractical and their use will negate any cloud cost advantages.

In this paper, we address the specific problem of performing practical range search and sort queries on encrypted outsourced data in a privacy-preserving fashion. We envision a generic scenario, where a set of *users* upload a large number of encrypted data records to an untrusted cloud store. From time to time, a *surveyor* wants to perform data analysis operations. Specifically, the surveyor queries for all the records in a certain range of *categories*. Alternatively, the surveyor may query for the *top* m sorted records, i.e., the m smallest records following some order.

Although the individual data records are encrypted, an untrusted cloud could still infer information about them by performing multiple range and sort operations, including the record access patterns, and correlating them. Consequently, also the operations ("queries") need to be privacy protected. The crucial challenge here is practicality,

i.e., high efficiency in terms of bandwidth, user/surveyor/cloud computations and memory requirements. Besides FHE, another technique that could apply here is Oblivious RAM [9]. With n the total number of outsourced records in the cloud, recent research has lowered ORAM worst-case communication complexity down to poly-logarithmic in n [16, 17]. Still, for large n (i.e. $n = 2^{30}$ records or more), this overhead becomes overly expensive. Finally, techniques such as Order Preserving Encryption (OPE) [5] are highly efficient, but provide weaker privacy guarantees.

We present RASP ("Range And Sort Privacy"), an original scheme for privacy-preserving range and sort queries on encrypted data. At the core of RASP, we introduce a new privacy-preserving bucket data structure LL similar to bucket sort. Each individual bucket in LL can grow dynamically in size, and we will use the buckets to represent the categories that (encrypted) records can belong to. Similar to standard bucket sort, we assume that the number D of possible different buckets for records remains small compared to n ($D \ll n$). We call our data type LL, which is of independent interest, *update-oblivious*, as it hides into which bucket a new record is added. RASP uses LL for range search, where it hides bucket contents until the surveyor explicitly queries for them. RASP also naturally extends to support m-sort queries. Targeting efficiency, RASP achieves slightly weaker privacy properties than, e.g., ORAM, but stronger privacy than related work on range search [6, 15]. Moreover, RASP only relies on hash computations and symmetric encryption, outperforming related work based on asymmetric cryptography and expensive bilinear pairings.

The major contributions of this paper are:

- RASP, a protocol that allows range search and sort queries on encrypted data in the cloud. We formally prove that the cloud cannot learn any information about added records until the surveyor queries them. Also, details about the queries are hidden, and only the overlap between queries is leaked.
- LL, a dynamic data structure that provably hides any information about a newly added data record until this data record is explicitly read.
- RASP is efficient and scales well. The user's and surveyor's, computational and communication complexities are constant $\mathcal{O}(1)$ in the total number of records n. In contrast to related work, RASP seamlessly integrates multiple different users that do not trust each other.
- We implement and evaluate RASP in Amazon's DynamoDB cloud.

2 Problem Statement

Background: To motivate our work, we use an example scenario throughout this paper. Assume a set of users $\mathcal{U}$ that continuously upload data records to a cloud store. Each record comprises: (1.) a category I of some domain $\mathbb{D}$ with an order relation, e.g., $\mathbb{D} = \{1, \ldots, D\} \subset \mathbb{N}$ and "$\leq$", and (2.) some payload data M. For the sake of range search and sort queries in this paper, M is not particularly interesting, and we focus only on indices I. After some time, users have uploaded a total of n records to the cloud, where n can become very large, while D is comparatively small, e.g., $n = 2^{30}$ and $D = 1024$. Periodically, a surveyor queries the uploaded records for those records whose indices match a certain range in $\mathbb{D}$. The set of records that match this range has

size m. Alternatively, the surveyor wants to retrieve the first m records according to their sorted indices.

Possible Applications: One can imagine various real world application scenarios that fall within the above general setup. For example, imagine a set of banks ("users") that upload financial transactions, i.e., the amount of each transaction together with details such as sender, receiver, date etc. Eventually, to detect fraudulent behavior and money laundering, the police queries for all transactions within some suspicious range or the m highest transactions of a certain time period. Alternatively, imagine a set of physicians that upload patient records, comprising the patient's personal information and, say, the patient's blood pressure. At some point, for further analysis, a health insurance wants to retrieve details about all patients with blood pressure in a critical range or the top m patients with high blood pressure. In both application scenarios, the stored data is sensitive, and the underlying cloud store should not learn details about either stored data or queries performed. This implies encrypting uploaded data by the users and "oblivious queries" by the surveyor.

2.1 Range and Sort Queries

We now formalize privacy-preserving m-Range and m-Sort schemes. We start by introducing the functionality that each scheme should support.

Definition 1 **(m-Range Search and m-Sort Scheme Π).** *Let I,$0 \leq I \leq D-1$, denote a category within domain $\mathbb{D}$ and M a plaintext ("payload"). A m-Range and m-Sort search scheme Π comprises the following algorithms.*

- $\mathsf{KeyGen}(s)$*: This algorithm uses security parameter s to generate secret key SK and the set of user keys $\{Seed_i\}_{1 \leq i \leq |\mathcal{U}|}$.*
- $\mathsf{Encrypt}(I, M, Seed_i)$*: encrypts M at category I using user key $Seed_i$. The algorithm's output is ciphertext C.*
- $\mathsf{Decrypt}(C, SK, i)$*: decrypts ciphertext C, such that* $\mathsf{Decrypt}(\mathsf{Encrypt}(I, M, Seed_i), SK) = M$*, where SK and $Seed_i$ were generated from* $\mathsf{KeyGen}(s)$.
- $\mathsf{PrepareRangeQuery}(a, b, SK)$*: uses secret key SK and a pair a,$b \in \mathbb{N}$ with $a \leq b$ to generate a range query token $\mathcal{T}^R$.*
- $\mathsf{RangeQuery}(\mathcal{T}^R, \{C_1, ..., C_n\})$*: using range search token $\mathcal{T}^R = \mathsf{PrepareRangeQuery}(a, b, SK)$ and a set of ciphertexts C_i, a response $S^R = \{C_i | C_i = \mathsf{Encrypt}(I_i, M_i, Seed_j) \forall j, 1 \leq j \leq |\mathcal{U}|$ and $I \in [a,b]\}$ is computed.*
- $\mathsf{PrepareSortQuery}(m, SK)$*: with secret key SK and length m,$1 \leq m \leq n$, outputs a sort query token $\mathcal{T}^S$.*
- $\mathsf{SortQuery}(\mathcal{T}^S, \{C_1, ..., C_n\})$*: using sort query token $\mathcal{T}^S$ and ciphertexts C_i, outputs a sequence $S^S = < C'_1, C'_2, \ldots, C'_m >$ as response. Here, ciphertext $C'_i = \mathsf{Encrypt}(I_i, M_i, Seed_i)$ denotes the ciphertext on the i^{th} position according to the order of the underlying indices I. More formally: (1.) for C'_1: there is no $C_{j,1 \leq j \leq n}$ such that $I_j < I_1$, (2.) for any pair C'_i,C'_{i+1}: either $I_i = I_{i+1}$, or $I_i < I_{i+1}$ and there are a total i ciphertexts $C_{j,1 \leq j \leq i}$ with $I_j < I_{i+1}$, (3.) for any pair C'_i,C'_j : $C'_i \neq C'_j$.*

2.2 Privacy

We will now present RASP's notion of privacy. Informally, our goal is to leak as little information as possible about the outsourced data records and the queries to the cloud. While the IND-CPA encryption of records already provides a viable first step, the challenge is to restrict leakage of query access patterns. For example, the cloud should not learn any additional information about records that are *not* part of a query result – besides that these records are obviously not in the queried range or among the top m records. Typically, ORAM based solutions would offer strong protection. However, focusing on efficiency, we dismiss ORAM, because even recent research resulting in poly-logarithmic worst-case communication complexity [16, 17] quickly become expensive with large n such as $n = 2^{30}$.

RASP focuses on a slightly weaker notion of privacy than ORAM: intuitively, the cloud (now called adversary $\mathcal{A}$) should not learn any details about a *new* record R that is added to the store, i.e., $\mathcal{A}$ should not learn anything about R's category I (and payload M). Only as soon as the surveyor executes a range search or sort query, $\mathcal{A}$ will learn whether R matches this query or not. Our goal is that any two records R,R' that do *not* match a query will remain computationally indistinguishable for $\mathcal{A}$. We now formalize our privacy goal. Targeting a simulation-based privacy definition, the idea is that, given a well specified privacy-leakage, a polynomial-time simulator can generate a transcript of RASP which is computationally indistinguishable from the output of the actual protocol. If this is true, then $\mathcal{A}$ cannot learn any information beyond the defined leakage.

We allow $\mathcal{A}$ to *only* learn: (1.) the operation pattern, i.e., which operation (range or sort) is performed, (2.) the data access pattern, i.e., which records are accessed during an operation, and (3.) the enumeration pattern, i.e., which records are returned.

We focus on key-value "(k,v)" based systems such as Amazon Dynamo DB or S3 as underlying store/database platform in this paper, so we assume that each record is uniquely addressable by an *address* k in the store.

Definition 2 (Operation). *For m-Range and m-Sort scheme Π, operation* op *is defined as either* $(\mathsf{Encrypt},I,M,Seed_i)$, $(\mathsf{RangeQuery},a,b,SK)$ *or* $(\mathsf{SortQuery},m,SK)$. *For ease of exposition, we introduce the following functions on operations:*

- $\mathsf{Type} : \mathsf{op} \rightarrow \{\mathsf{Encrypt},\mathsf{RangeQuery},\mathsf{SortQuery}\}$ *which extracts the operation type from an operation.*
- $\mathsf{Execute} : \mathsf{op} \rightarrow (\mathcal{K} = (k_1,...,k_t),\mathcal{C} = (c_1,...,c_t))$ *which executes* op *and returns the result. The set $\mathcal{K}$ contains the sequence of addresses accessed on the cloud, and $\mathcal{C}$ contains the data, i.e., records at those addresses after the operation.*
- $\mathsf{Categories} : (c_1,...,c_t) \rightarrow \{I_1,...,I_t\} \subset \mathbb{D}^t$ *which extracts the categories I_i out of a sequence of records c_i.*

Definition 3 (History). *A q-query history is the sequence of operations $H = (o_1,...,o_q)$, where $o_i = (\mathsf{Encrypt},I,M)$, $o_i = (\mathsf{RangeQuery},a,b)$ or $o_i = (\mathsf{SortQuery},m)$.*

Definition 4 (Operation Pattern). *The operation pattern induced by a q-query history H is the sequence $\beta(H) = (\mathsf{Type}(o_1),...,\mathsf{Type}(o_q))$.*

Definition 5 (Category Pattern). *Let π be a random permutation of integers $\{1,...,D\}$. The category pattern of a q-query history is the following q-length sequence $\sigma(H)$: first, consider the case that* $\mathsf{Type}(o_i) = \mathsf{RangeQuery}$ *or* $\mathsf{Type}(o_i) = \mathsf{SortQuery}$. *Let* $\mathsf{Execute}(o_i) = ((k_1,...,k_t),(c_1,...,c_t))$ *and* $\mathsf{Categories}(c_1,...,c_t) = \{I_1,...,I_t\}$. *Then, $\sigma(H)[i]$ is the* unordered *set of tuples $\{(\pi(I_1),\chi(I_1)),...(\pi(I_t),\chi(I_t))\}$, where $\chi(I) = (e_1,...,e_m)$ returns the indices such that $\forall j < i : o_{e_j} = (\mathsf{Encrypt}, I_j, M_j)$. If $\mathsf{Type}(o_i) = \mathsf{Encrypt}$, then $\sigma(H)[i]$ is the empty set.*

That is, $\sigma(H)$ reveals which *previous* Encrypt operations are being queried as part of the *current* range search or sort query operation i and the pattern of categories that are accessed for that operation. For any two range queries, $\sigma(H)$ will tell which categories they have in common. Due to random permutation π, $\sigma(H)$ does not give any information about the ordering of the categories beyond what can be inferred by the overlap.

Definition 6 (Sort Leakage). *The sort leakage from a q-query history H is the q-length sequence $\gamma(H)$ defined as follows: if o_i is a sort query, i.e.,* $\mathsf{Type}(o_i) = \mathsf{SortQuery}$ *with* $\mathsf{Execute}(o_i) = ((k_1,...,k_t),(c_1,...,c_t))$ *and* $\mathsf{Categories}(c_1,...,c_t) = \{I_1,...,I_t\}$, *then $\gamma(H)[i]$ is the tuple $(m,(\pi(1),...,\pi(t)))$. If* $\mathsf{Type}(o_i) = \mathsf{Encrypt}$ *or* $\mathsf{Type}(o_i) = \mathsf{RangeSearch}$, *then $\gamma(H)[i]$ is $\perp$.*

This means that the sort leaks the first j categories, including their order, as well as the number of records returned by the query.

Definition 7 (Trace). *The trace induced by a q-query history H is the tuple $\tau(H) = (\sigma(H),\beta(H),\gamma(H))$.*

Definition 8 (Privacy-Preserving). *Let Π be an m-Range Search and $m-$Sort Scheme implementing* $\mathsf{Execute}$ *for operations* $\mathsf{Encrypt}, \mathsf{RangeQuery}, \mathsf{SortQuery}$ *and therewith generating trace τ. Let $s \in \mathbb{N}$ be the security parameter, $\mathcal{A}$ be an adversary, and $\mathcal{S}$ be a simulator. Consider the following two experiments:*

$\mathbf{Real}^{\Pi}_{\mathcal{A}}(s)$
$K \leftarrow \mathsf{KeyGen}(1^s)$
for $1 \le i \le q$
  $(st_{\mathcal{A}}, o_i) \leftarrow \mathcal{A}(st_{\mathcal{A}}, (\mathcal{K}_1,\mathcal{C}_1), ..., (\mathcal{K}_{i-1},\mathcal{C}_{i-1}))$
  $(\mathcal{K}_i,\mathcal{C}_i) \leftarrow \mathsf{Execute}(o_i)$
let $\mathcal{K}^* = (\mathcal{K}_1,...,\mathcal{K}_q)$
let $\mathcal{C}^* = (\mathcal{C}_1,...,\mathcal{C}_q)$
output $\mathbf{v} = (\mathcal{K}^*,\mathcal{C}^*)$ *and* $st_{\mathcal{A}}$

$\mathbf{Sim}^{\Pi}_{\mathcal{A},\mathcal{S}}(s)$
for $1 \le i \le q$
  $(st_{\mathcal{A}}, o_i) \leftarrow \mathcal{A}(st_{\mathcal{A}}, (\mathcal{K}_1,\mathcal{C}_1), ..., (\mathcal{K}_{i-1},\mathcal{C}_{i-1}))$
  $(\mathcal{K}_i,\mathcal{C}_i,st_{\mathcal{S}}) \leftarrow \mathcal{S}(st_{\mathcal{S}}, \tau(o_1,...,o_i))$
let $\mathcal{K}^* = (\mathcal{K}_1,...,\mathcal{K}_q)$
let $\mathcal{C}^* = (\mathcal{C}_1,...,\mathcal{C}_q)$
output $\mathbf{v} = (\mathcal{K}^*,\mathcal{C}^*)$ *and* $st_{\mathcal{A}}$

Scheme Π is privacy-preserving, iff *for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$, such that for all PPT distinguishers $\mathcal{D}$,*

$$|Pr[\mathcal{D}(\mathbf{v},st_{\mathcal{A}}) = 1 : (\mathbf{v},st_{\mathcal{A}}) \leftarrow \mathbf{Real}^{\Pi}_{\mathcal{A}}(s)] - Pr[\mathcal{D}(\mathbf{v},st_{\mathcal{A}}) = 1 : (\mathbf{v},st_{\mathcal{A}}) \leftarrow \mathbf{Sim}^{\Pi}_{\mathcal{A},\mathcal{S}}(s)]| \le \mathsf{negl}(s).$$

Our definition captures an adaptive adversary which generates the history one operation at a time, seeing the results of the previous operations. This allows for an adversary which changes his strategy depending on what the simulator has output after $i < q$ operations. Consequently, the simulator calculates one step of the simulation at a time based on a partial trace generated from an adaptive history.

Definition 8 is generic in that it allows us to bound the information leaked by any protocol which uses a cloud data store. If there exists a simulator, given only the trace, which can produce a sequence of "accesses" that is indistinguishable from a real execution of the protocol, then no information other than the trace can be leaked.

Discussion: Note that our privacy definition is *stronger* than "selective security" offered by related work on range search [6, 15]. There, a query for a specific range $[a,b]$ implies that $\mathcal{A}$ from then on will be able to automatically determine whether records added in future will also be within $[a,b]$ or not. In contrast, our privacy definition guarantees that even a record $R \in [a,b]$ added *after* the range query for $[a,b]$ is indistinguishable from random until $[a,b]$ is queried again. We stress that our privacy notion is also stronger than IND-OCPA [14], leaking only record-category relations that are explicitly queried.

3 Update-Oblivious Linked Lists

For its range and sort queries, RASP relies on a new kind of data type that we call *update-oblivious add-enumerate* data type. Its purpose is to allow a *Writer* to add values to *buckets* (the categories). Also, a *Reader* can enumerate all values of a bucket. The sole privacy goal is to hide from an adversary storing all data which bucket a new value is added to by the Writer until this specific bucket is enumerated by the Reader. We will now start by describing the operations and privacy properties that update-oblivious add-enumerate data types support. Then, we will introduce an original data type LL, a sequence of linked lists that supports update-oblivious operations, and we will prove its privacy properties.

An add-enumerate data type comprises a sequence of D buckets, dynamic data structures indexed by $I, 1 \leq I \leq D$. This data type supports adding values to the individual buckets and enumerating individual buckets, i.e., enumerating all values that have been previously added to one of the buckets. Again, we assume a key-value based underlying store. Each "value" v added is uniquely addressable by an address k in the store. More formally, an add-enumerate data type supports the following two algorithms:

- $\mathsf{Add}(I,v)$: On input bucket $I, 1 \leq I \leq D$ and a value $v \in \{0,1\}^*$, this algorithm adds v to I and outputs an address $k \in \{0,1\}^*$. We call the pair (k,v) *valid*.
- $\mathsf{Enumerate}(I)$: This algorithm returns the set $\{(k,v) | v \in I \wedge (k,v) \text{ is valid}\}$.

3.1 Update-Oblivious Privacy

Again, we use a simulation-based privacy definition for update-oblivious data types. Similarly to Definition 8, $\mathcal{A}$ can learn (1.) the operation pattern (which operation is performed on the data type), (2.) the data access pattern (which data in the individual data structures is accessed during an operation), and (3.) the enumeration pattern (which values are returned as part of an $\mathsf{Enumerate}$). However, the enumeration pattern will not reveal the indices of values enumerated never before. Being clear from the context, **we**

reuse the notions of histories and operations defined previously in the context of add-enumerate data types, with a new definition of Trace to quantify information leakage.

Definition 9 (Operation). *An operation* op *is either* (Add, I, v) *or* $(\mathsf{Enumerate}, I, \perp)$.

Definition 10 (Enumeration Pattern). *The enumeration pattern induced by a q-query history H is $q \times q$ binary matrix $\sigma(H)$ where for $1 \leq i,j \leq q$ the entry in the i^{th} row and j^{th} column is 1,* iff $i \leq j$ $\mathsf{Type}(o_j) = \mathsf{Enumerate}$ *and $I_i = I_j$. Otherwise, this entry is 0.*

If $\mathsf{Type}(o_i) = \mathsf{Add}$, then the i^{th} row of this matrix contains ones in the columns corresponding to an enumerate that happens *after* this add. Therewith, the enumeration pattern reveals which category an add corresponds to only *after* an enumerate occurs for that same category. If the result of an add is never queried, i.e., no enumerate occurs after for that category, then the entire row in the matrix will contain only zeroes, and the category of the add is not leaked.

Definition 11 (Access Pattern). *Let π be a random permutation of the integers $\{1,\ldots, D\}$. The access pattern of a q-query history is q-length sequence $\gamma(H)$, such that, if o_i is an enumerate on category j, then $\gamma(H)[i] = \pi(j)$. Otherwise, $\gamma(H)[i] = 0$.*

That is, access pattern γ will reveal when two enumerates are on the same category. This is also revealed as part of the enumeration pattern, but we include this separate notation for clarity.

Definition 12 (Trace). *The trace induced by a q-query history H is the tuple $\tau(H) = (\sigma(H), \beta(H), \gamma(H))$.*

Definition 13 (Adaptive update-oblivious). *We define adaptive update-oblivious ("update-oblivious") privacy using the same generic simulation-based experiments as above (Definition 8), but include the new Definition 12 of trace for this data structure.*

3.2 The Data Type LL

We present a new add-enumerate data type LL which implements the sequence of D buckets as *linked lists* on top of any key-value based store.

Overview: The main rationale of LL is that Reader and Writer synchronize their access to the same bucket/linked list I using an array of flags. If the Writer wants to update linked list I by adding a new value, he verifies whether the Reader has enumerated I after the last add by checking the flag for this list. If the Reader has enumerated I, then the Writer does not append a new value to I, but creates a new *chain* for I, adds the new value to this chain, and updates the flag. The Writer will continue adding values to this new chain, until the Reader enumerates I again. Then, the Writer will create another new chain etc. On the other hand, the Reader checks a flag to understand whether the Writer has created a new chain for I, thereby knowing how many chains of I contain values.

Details: Reader and Writer share a secret key κ. LL comprises a total of D linked lists which are indexed by $I, 1 \leq I \leq D$. In the underlying key-value store, the head of linked list I, the start of the first chain of I, can be accessed using address $h_\kappa(I,1)$, where h is a pseudo-random function, and "," is an unambiguous pairing of inputs.

Algorithm 1: LL-Add(I,v,κ)

Input : Pair (I,v), secret key κ, local sequences of next list pointers $\Psi=(\psi_1,...,\psi_D)$ and counters $\Gamma^{\text{Writer}}=(\gamma_1^{\text{Writer}},...,\gamma_D^{\text{Writer}})$,security parameter s

Output: Address k, ciphertext e of new record

$E_\kappa(\Delta):=\mathsf{Get}(h_\kappa(\text{“delta”}))$ and decrypt to $\Delta=(\delta_1,...,\delta_D)$;
if $\delta_I=1$ **then**
  $\gamma_I^{\text{Writer}}:=\gamma_I^{\text{Writer}}+1$;
  $\psi_I:=h_\kappa(I,\gamma_I^{\text{Writer}})$;
  $\delta_I:=0$;
  $\mathsf{Put}(h_\kappa(\text{“delta”}), E_\kappa(\Delta)=E_\kappa(\delta_1,...,\delta_D))$;
end
$k:=\psi_I$;
$\psi_I \xleftarrow{\$} \{0,1\}^s$;
new Record e;
$e.\text{value}:=E_\kappa(v)$; $e.\psi:=E_\kappa(\psi_I)$;
$\mathsf{Put}(k,e)$;
return (k,e);

Algorithm 2: LL-Enumerate(I,κ)

Input : Category I, secret key κ, local counters $\Gamma^{\text{Reader}}=(\gamma_1^{\text{Reader}},...,\gamma_D^{\text{Reader}})$

Output: Set of ciphertexts $\mathcal{S}=\{c_i|c_i\in I\}$

$\mathcal{S}:=\varnothing$;
$E_\kappa(\Delta):=\mathsf{Get}(h_\kappa(\text{“delta”}))$ and Decrypt to $\Delta=(\delta_1,...,\delta_D)$;
for $i:=1$ **to** $\gamma_I^{Reader}-1$ **do**
  $\text{start}:=h_\kappa(I,i)$;
  $\mathcal{S}:=\mathcal{S}\cup\mathsf{Retrieve}(\text{start},\kappa)$;
if $\delta_I=0$ **then**
  $\text{start}:=h_\kappa(I,\gamma_I^{\text{Reader}})$;
  $\mathcal{S}:=\mathcal{S}\cup\mathsf{Retrieve}(\text{start},\kappa)$;
  $\delta_I:=1$;
  $\mathsf{Put}(h_\kappa(\text{“delta”}), E_\kappa(\Delta)=E_\kappa(\delta_1,...,\delta_D))$;
  $\gamma_I^{\text{Reader}}:=\gamma_I^{\text{Reader}}+1$;
return $(\mathcal{S})$;

Writer and reader synchronize using an encrypted array of flags $\Delta=(\delta_1,...,\delta_D)$, $\delta_i\in\{0,1\}$. They can save and retrieve $E_\kappa(\Delta)$, where E denotes encryption, in the underlying key-value store using address $h_\kappa(\text{“delta”})$. Initially, all flags δ_i are set to 0.

For each linked list I, the Writer stores a local counter γ_i^{Writer}. All counters are initialized to 1. The purpose of these counters is to keep track of the number of chains that have been created per linked list. Each time the Writer starts a new chain for a linked list I, he increases γ_I^{Writer} by one. Along the same lines, the Reader also keeps a local sequence of counters γ_i^{Reader}, initialized to 1. Each time the Reader sees that the flag for a specific linked list I has been changed, i.e., the Writer has created a new chain for I, the Reader will increase γ_I^{Reader} by one. Moreover, the Writer locally stores for each linked list I a *next pointer* ψ_I. This next pointer represents the address in the underlying key-value store for the next value v to be added to linked list I. Initially, each ψ_I is set to $h_\kappa(I,1)$, i.e., the start of the first chain of list I.

Add*:* In case the Writer wants to add a new value v to linked list I, he executes Algorithm 1. First, he downloads and decrypts the δ_i. Note that we use the standard key-value semantic Get and Put to access data in the underlying store. If $\delta_I=1$, then the Reader has accessed list I since the last add, and the Writer creates a new chain for I. The Writer increases counter γ_I^{Writer}, sets next pointer ψ_I to the start of the new chain $h_\kappa(I,\gamma_I^{\text{Writer}})$, resets flag δ_I, and uploads a new encryption of all flags Δ. In any case, the Writer uploads an encrypted version of v using the current address that ψ_I points at. Together with the encryption of v, the Writer also uploads a randomly chosen encrypted

Algorithm 3: LL-Retrieve(ψ,κ)

Input : Chain start pointer ψ, secret key κ
Output: Set of ciphertexts $\mathcal{S}$

```
S := ∅;
Record e := Get(ψ);
while e ≠ ⊥ do
    S := S ∪ e.value;
    ψ := D_κ(e.ψ);
    e := Get(ψ);
return S;
```

new next pointer ψ_I. For convenience, we call the combination of an encrypted value v and encrypted next pointer ψ_I a *record*.

Enumerate*:* In case the Reader wants to retrieve all (encrypted) values of linked list I, he executes Algorithm 2. First, the Reader downloads and decrypts the δ_i. If $\delta_I = 1$, then the Writer has not updated I since the last enumerate. Consequently, the Reader will retrieve all values of all the current $(\gamma_I^{\text{Reader}} - 1)$ chains of list I. Otherwise, if $\delta_I = 1$, then the Writer has started a new chain for I since the last enumerate. So, the Reader will retrieve all records of the previous $(\gamma_I^{\text{Reader}} - 1)$ chains of I, then retrieve all records of chain γ_I^{Reader}, set flag δ_I, encrypt and upload all flags Δ, and finally increase counter γ_I^{Reader}. Note that the Reader retrieves all values of a chain by using Algorithm 3. There, D is the decryption algorithm for encryptions E. Using Δ for synchronization between Writer and Reader, the Writer will avoid putting a new value into the underlying store using an address that the Reader has previously already queried for as part of an enumerate. In this case, the Writer will start a new chain and notify the Reader that a new chain is available.

3.3 Privacy Analysis

In accordance with Curtmola et al. [8], we introduce PCPA-secure encryption as an encryption producing ciphertexts that are indistinguishable from random. This can be implemented, e.g., by a PRP (like AES) in CBC- or Counter-mode.

Theorem 1. *If h is a pseudo-random function, and E is a PCPA-secure encryption, then* LL *is update-oblivious.*
Due to space constraints, we move the proof to Appendix A.

Discussion: Our definition of update-oblivious and data type LL capture only semi-honest ("honest-but-curious") adversaries. A fully malicious adversary $\mathcal{A}$, however, might violate an implicit consistency requirement that we have: *read-after-write consistency*. Although the Reader has read bucket I and set $\delta_I := 1$, $\mathcal{A}$ could send an old version of $E(\Delta)$ with $\delta_I = 0$ to the Writer during Add. Consequently, the Writer would not create a new chain, but add a new record at an address already read by the Reader – violating update-obliviousness. Yet, we can modify LL in a straightforward manner to cope with consistency attacks. We augment Δ by two additional global counters, one for the Reader, one for the Writer. Both counters will be encrypted as part of Δ. For

each Enumerate operation, the Reader increases its counter. For each Add, the Writer increases its counter. Both parties keep local copies of counters and can therewith verify the freshness of Δ before each operation. Along the lines of Li et al. [13], we then achieve *Fork Consistency*. Due to space constraints, we will not discuss details.

Extensions: The update-oblivious property can be extended to other data types. Let a *monotonically-expanding* data type be any data type S, that supports two general operations Add(S,E), and Enumerate(S,param) such that $i < j \Rightarrow \mathsf{Enumerate}(\mathsf{S_i},\mathsf{param}) \subseteq \mathsf{Enumerate}(\mathsf{S_j},\mathsf{param})$. We postulate that any monotonic data type can be made update-oblivious. Hash Tables, Trees, Graphs are examples of data types that can be restricted to be monotonically-expanding, if deletions are not allowed. Such expanding types make sense in applications where data is continuously added to an application data store. For example, an update-oblivious Hash Table that stores key-value pairs can be constructed using our bucket data type LL. The user hashes the key into a bucket id I, then invokes LL-Add(I,v,κ). Graphs (and their special case trees) can be viewed as a collection of edges. The Add adds edges to the collection, while Enumerate lists the edges (or properties of the edges). We conjecture that our bucket data type LL can be used to enable the same update-obliviousness for expanding graphs, trees, and other dynamic data types.

4 RASP

Overview: The main rationale behind RASP is to arrange uploaded data with category I using an update-oblivious add-enumerate data type such as LL that offers buckets. In RASP, each individual bucket represents a category I within domain $\mathbb{D}$ of uploaded data M. With $n \gg D$, we achieve low query complexity similar to bucket sort. Uploading new data into a bucket is realized by using Add in LL. Similarly, range search queries and actual bucket sorts can be realized using Enumerate. Our goal with this approach is to reduce the complexity for range search and sort queries over related work. RASP's query complexity depends only on D and $\mathcal{U}$ which we assume to be small, but the complexity is independent of n as in related work. To support multiple users $\mathcal{U}$, we use an LL data type *per user*. Users share pairwise different keys with the surveyor.

4.1 RASP Details

We now present RASP's details, following the notation introduced in Section 2.1. The system is initialized with KeyGen, producing key material for users and the surveyor. Each time, a user U_i wants to upload data to the cloud, he first uses Encrypt and uploads the resulting ciphertext into the cloud's key-value store. For a range query, the surveyor executes PrepareRangeQuery and RangeQuery intertwined. For a sort query, he uses PrepareSortQuery and SortQuery intertwined. In the algorithms below, E and D are PCPA encryption and decryption (see Section 3.3) such as AES-CBC with random IVs, and h is a pseudo-random function such as HMAC [4] using proper padding of inputs.

KeyGen(k) As shown in Algorithm 4, the system is initialized by generating a secret key SK for the surveyor as well as individual user keys Seed_i. A key Seed_i is then sent to user $\mathcal{U}_i$, and the surveyor receives SK. Note that knowledge of SK is sufficient for the surveyor to compute users keys Seed_i himself.

Algorithm 4: KeyGen(s) – generate keys for surveyor and users

Input: Security parameter s
Output: Surveyor's secret key SK, set of user keys $\{\text{Seed}_i\}_{1\leq i\leq|\mathcal{U}|}$

```
SK ←$ {0,1}^s;
for i := 1 to |U| do
    Seed_i := h_SK(i);
return SK, {Seed_i}_{1≤i≤|U|};
```

Algorithm 5: Encrypt(I,M,Seed$_i$) – user U_i encrypts and uploads to cloud.

Input: Category I, data M, user U_i's key Seed$_i$
Output: Ciphertext C that is uploaded to cloud

```
κ := Seed_i;
(k,C) := LL-Add(I,M,κ);
return C;
```

Algorithm 6: Decrypt(C, SK, i) – surveyor decrypts ciphertext

Input: Ciphertext C, surveyor secret key SK, user ID i
Output: Data M

```
Seed_i := h_SK(i);
M := D_{Seed_i}(C);
return M;
```

Encrypt(I, M, Seed_i) Algorithm 5 is executed by user U_i that wants to add data M to bucket I in the key-value store. U_i simply runs LL-Add to add data M to category/bucket I in LL. Note that LL-Add uploads the encrypted data to the key-value store as of Algorithm 1.

Decrypt(C, SK, i) Algorithm 6 is run by the surveyor. The surveyor computes Seed$_i$ using his secret key SK and decrypts the ciphertext.

PrepareRangeQuery(a,b,SK) and RangeQuery$(\mathcal{T}^R, \{C_1,...,C_n\})$ For ease of understanding, we present PrepareRangeQuery and RangeQuery together in Algorithm 7. As you will see, token $\mathcal{T}^R$ of PrepareRangeQuery is the sequence of addresses k_i required to download ciphertexts C_i from the key-value store. The surveyor iterates over all possible users to generate their keys Seed$_i$ and retrieve all data for buckets $j \in [a,b]$. The surveyor permutes his access to buckets j by using permutations π_{Seed_i} which are random permutations over integers $\{a,...,b\}$. To retrieve all data of user U_i in bucket $\pi(j)_{\text{Seed}_i}$, the surveyor uses LL-Enumerate$(\pi_{\text{Seed}_i}(j),\text{Seed}_i)$. The rationale behind using π_{Seed_i} to enumerate buckets is not to always access, first, bucket a, then bucket $a+1$, then $a+2$ etc. until bucket b, but to access buckets in a random order. Note that LL-Enumerate internally uses addresses to access data in the key-value store, representing $\mathcal{T}^R$.

PrepareSortQuery(m,SK) and SortQuery$(\mathcal{T}^S, \{C_1,...,C_n\})$ Similar to range search queries, we consolidate PrepareSortQuery and SortQuery together in Algorithm 8. Again, the surveyor starts by computing all possible user keys Seed$_i$. Now, the surveyor iterates over all possible buckets and therein over all possible users, starting with the lowest bucket. To retrieve data from individual buckets, the surveyor uses Algorithm LL-Enumerate-UpTo(i,Seed_j,m). This is a slight variation of the standard LL-Enumerate(i,Seed_j), cf. Algorithm 2. We do not give details for LL-Enumerate-UpTo, because the only difference is the additional parameter m. This parameter specifies that

Algorithm 7: PrepareRangeQuery(a, b, SK) and RangeQuery($\mathcal{T}^R$, $\{C_1, \ldots, C_n\}$) – surveyor prepares and executes range query from a to b with cloud

Input: Surveyor: Indices a and b, surveyor's secret key SK, Cloud: pairs (k_i, C_i)
Output: Set of ciphertext $S^R = \{C_i\}$

```
1  S^R := ∅;
2  for i := 1 to |U| do
3      for j := a to b do
4          Seed_i := h_SK(i);
5          S^R := S^R ∪ LL-Enumerate(π_Seed_i(j), Seed_i);
6  return S^R;
```

Algorithm 8: PrepareSortQuery(m, SK) and SortQuery($\mathcal{T}^S$,$\{C_1,\ldots,C_n\}$)

Input: Surveyor: Position in sorted data P, window length m, secret key SK, Cloud: pairs (k_i, C_i)
Output: Set of ciphertext $S^S = \{C_i\}$

```
1  for i := 1 to |U| do
2      Seed_i := h_SK(i);
3  S^S := ∅; i := 1; pos := 1;
4  while m > 0 and pos ≤ D do
5      S' := LL-Enumerate-UpTo(pos, Seed_i, m);
6      m := m − |S'|;
7      S^S := S^S ∪ S';
8      if i = |U| then
9          i := 1;
10         pos := pos + 1;
11     else
12         i := i + 1;
13 return S^S;
```

algorithm LL-Enumerate will stop retrieving ciphertexts after finding m ciphertexts (if available) while iterating over the chains of category i, regardless whether there might be more ciphertexts in the chains. Following the definition of m-sort in Section 2.1, the search token $\mathcal{T}^S$ in RASP is the sequence of addresses for the key-value store.

Note During range search and sort queries, when the surveyor performs multiple Enumerate sequentially for the same user, it is not necessary to download and upload Δ multiple times, but only once. The same Δ can be used for all categories/buckets of a single user, so this saves a factor of D in computation and communication complexity without affecting privacy. We apply this optimization in our evaluation in Section 4.3.

4.2 Privacy Analysis

Theorem 2. *If* LL *is update-oblivious, then RASP is privacy-preserving.*
Due to space constraints, we move the proof to Appendix A.

4.3 Evaluation

We have implemented RASP in Java, and the source code is available for download [3]. Our implementation uses Amazon's Dynamo DB cloud as key-value cloud storage. Dynamo DB charges based on the required Get/Put operations per second which is essentially an estimation of the load expected on the database. For our tests, with configured a database supporting 1000 Get and 1000 Puts per second and read-after-write consistency. Such a database would cost $\approx \$580$ per month [2]. As encryption E, we use 128 Bit AES in CBC-mode, and is HMAC is based on SHA-1 as underlying hash function h. As we are only interested in the additional overhead of RASP compared to a non-privacy preserving protocol, we did not encrypt and upload a real payload d (e.g.,

patient data) as part of records, but only a random string of length 160 Bit. In the real world, this could be an address for a larger file in the cloud. For the user and surveyor part of RASP, we have used a laptop with 2.4 GHz i7 CPU and 8 GByte RAM.

As RASP's query performance does not depend on n, we have measured timings for Encrypt (and upload), Range Queries, and Sort Queries on a Dynamo data store with a fixed number of $n = 2^{20}$ records. We have varied the number of users $|\mathcal{U}| = \{100,1000\}$, $D = \{256,8192\}$, and $m = \{1\% \text{ of } n, 5\% \text{ of } n\}$ for range queries and $m = \{50,500\}$ for *top* m-sort. As m only depends on n, the distribution of records into categories does not matter, and users add data uniformly randomly into categories. To model interleaving client and surveyor operations (and force creating new chains), we distributed queries exponentially with $\frac{n}{100}$ average arrival rate. However, we measured timings only after adding all n records, representing worst case queries. We have run each sample point 20 times, and relative standard deviation was low at $<5\%$.

Table 4 in Appendix A sums up our evaluation and presents timings in ms *per record*. All timings are dominated by network latency. In Dynamo, a single Get takes 31 ms, and a Put takes 39 ms. In contrast, AES encryption (≈ 14 ns) and HMAC evaluation (≈ 6 ns) are comparably fast on our hardware. Note that the time for a single Encrypt of a single user is significantly higher per record compared to Range and Sort queries for the surveyor (per record), although they same operations (encryption, hash, network access) are required. This is due to the fact that during, e.g., a range query, the surveyor can enumerate multiple LL-buckets of multiple users in *parallel* with Dynamo. Dynamo allows to issue multiple Get-requests from multiple threads in parallel, reducing the total response time significantly. Timings increase slightly with D, as Δ becomes larger and needs to be downloaded/uploaded and decrypted/encrypted by the surveyor. Similarly, with increasing $|\mathcal{U}|$, more Δ need to be taken care of. Sort is more expensive than Range search, because access to buckets cannot be parallelized as with range search: to find the first m records, buckets need to be accessed sequentially.

As you can see from Table 4, RASP is highly efficient. Even large range queries retrieving $\approx 50{,}000$ records, i.e., 5% of n over $8{,}192$ categories and $1{,}000$ users take only ≈ 1 min time. Similarly, for the same configuration, retrieving the *top*-50 records takes only ≈ 30 seconds. Again, we stress that these number do not depend on n.

5 Related Work and Summary

Traditionally, privacy-preserving range search and sort queries can be realized based on, for example, ORAM [9] protocols. The idea would be to simply encrypt all records, store them in an ORAM, and let the surveyor perform the range search on the ORAM. This results in "ideal" privacy, because plaintexts can be encrypted using an IND-CPA cipher, and ORAM does not leak any information about accesses and therewith queries. The drawback of ORAM is that its worst-case communication complexity remains high for this application despite recent results that reduce it to poly($\log n$) [16, 17]. Alternatively, using OPE [5, 14], ciphertexts retain the order of their underlying plaintexts, making range search and sorting straightforward. However, OPE gives weak privacy guarantees, as the relationship between ciphertexts is visible to the cloud.

Some related work focuses especially on privacy-preserving range search. For example, Hacigümüs et al. [11] and Hore et al. [12] encrypt records and put ciphertexts

Table 1. Worst-case complexity for m-Range search. n: total number of records, m: number of records queried D: size of domain, $\mathcal{U}$: set of users, s, s': security parameters. Typically, $s \gg \log n, s \gg \log D, s > s', n \gg D, n \geq m$. Related work requires additional factor $|\mathcal{U}|$ for multi-user.

		m-Range			
		Computation per Insert User	Computation per Query Surveyor	Computation per Query Cloud	Communication per Query
Single User	OPE [14]	$\mathcal{O}(\log n)^{\ddagger}$	$\mathcal{O}(\log n+m)^{\ddagger}$	—	$\mathcal{O}((\log n+m)\cdot s)$
	ORAM [16]	$\mathcal{O}(\log^3 n)^{\ddagger}$	$\mathcal{O}(n\cdot\log^3 n)^{\ddagger}$	—	$\mathcal{O}(n\cdot\log^3 n\cdot s)$
	ORAM [17]	$\mathcal{O}(\log^2 n)^{\ddagger}$	$\mathcal{O}(n\cdot\log^2 n)^{\ddagger}$	—	$\mathcal{O}(n\cdot\log^2 n\cdot s)$
	Range Search [6]	$\mathcal{O}(D)^{\diamond}$	$\mathcal{O}(m)^{\ddagger,\diamond}$	$\mathcal{O}(n)^{\diamond}$	$\mathcal{O}(m\cdot s)$
	Range Search [15]	$\mathcal{O}(\log D)^{\diamond}$	$\mathcal{O}(\log D+m)^{\diamond}$	$\mathcal{O}(n)^{\diamond}$	$\mathcal{O}(m\cdot(\log D+s))$
Multi User	*Ideal*	$\mathcal{O}(1)$	$\mathcal{O}(m)$	—	$\mathcal{O}((m+\log D)\cdot s))$
	This paper	$\mathcal{O}(D)^{\ddagger}$	$\mathcal{O}(\lvert\mathcal{U}\rvert\cdot D+m)^{\ddagger}$	—	$\mathcal{O}((\lvert\mathcal{U}\rvert\cdot D+m)\cdot s)$

‡ Involves Symmetric cryptography $^{\diamond}$Involves Exponentiations/Pairings

in a set of permuted categories. While this hides into which category a record is added, the cloud automatically learns the relationship between ciphertexts and can determine which ciphertexts are in the same category. Boneh and Waters [6] and Shi et al. [15] overcome this drawback and hide membership to a category until this particular category is queried – still, the cloud will be able to determine for any ciphertext added after the query whether it is belonging to the previously queried category or not. While Boneh and Waters [6] is "match concealing" (MC), the work by Shi et al. [15] is "match revealing" (MR), i.e., the cloud will learn the category of a record that matches a range query. Moreover, both schemes make use of computationally expensive bilinear pairings.

We sum up RASP's asymptotic performance and compare it to related work in tables 1 to 3. We stress that related work has not been designed for use in multi-user scenarios. While related work could be extended to multiple users, e.g., using different keys for each user in OPE or separate ORAMs for each user, this increases complexities significantly or would requires significant redesign. A straightforward extension adds a factor of $|\mathcal{U}|$ in tables 1 to 3, which quickly renders such approaches overly costly.

Tables 1 and 2 show the computational complexities to add a new record to the store for the user (for both range search and sort), for the surveyor to perform the query, and for the cloud during a query. The computational complexities comprise record encryption and decryption (for m records) operations. The communication complexities denote the communication between surveyor and cloud during a range or sort query. Security factor s in the communication complexities indicates that ciphertexts are exchanged. In each table, we compare to an *ideal* solution, representing a lower bound for each complexity, respectively.

ORAM does not provide any ordering, range queries would require scanning through all n records and quickly become expensive for large n, cf. Table 1. Consequently, a system based on ORAM would probably insert using a balanced tree or Radix sort. A balanced tree requires $\mathcal{O}(n\cdot\log n)$ accesses to the ORAM and Radix sort $\mathcal{O}(n\cdot\log D)$ accesses, each of them with complexity in poly($\log n$). OPE and an *Ideal* solution all

Table 2. Worst-case complexity for m-Sort.

		m-Sort		
		Computation per Query		Communication per Query
		Surveyor	Cloud	
Single User	OPE [14]	$\mathcal{O}(m)^{\ddagger}$	$\mathcal{O}(\log n+m)$	$\mathcal{O}(m\cdot s)$
	ORAM [16]	$\mathcal{O}(n\cdot\log D\cdot\log^3 n)^{\dagger,\ddagger}$	—	$\mathcal{O}(n\cdot\log D\cdot\log^3 n\cdot s)$
	ORAM [17]	$\mathcal{O}(n\cdot\log D\cdot\log^2 n)^{\dagger,\ddagger}$	—	$\mathcal{O}(n\cdot\log D\cdot\log^2 n\cdot s)$
Multi User	*Ideal*	$\mathcal{O}(m)$	—	$\mathcal{O}(m\cdot s)$
	This paper	$\mathcal{O}(\lvert\mathcal{U}\rvert\cdot D+m)^{\ddagger}$	—	$\mathcal{O}((\lvert\mathcal{U}\rvert\cdot D+m)\cdot s)$

† Using Radix Sort ‡Involves Symmetric cryptography

Table 3. Storage worst-case complexity and privacy guarantees.

		Storage		Privacy
		Per Ciphertext	User Memory	
Single User	OPE [14]	$\mathcal{O}(\log n+s)$	$\mathcal{O}(s)$	IND-OCPA
	ORAM [16]	$\mathcal{O}(\log n\cdot s)$	$\mathcal{O}(s)$	IND-CPA and Pattern
	ORAM [17]	$\mathcal{O}(s'\cdot s)$	$\mathcal{O}(s'\cdot\log^2 n\cdot s)$	
	Range Search [6]	$\mathcal{O}(D)$	$\mathcal{O}(s)$	MC
	Range Search [15]	$\mathcal{O}(\log D\cdot s)$	$\mathcal{O}(\log D\cdot s)$	MR
Multi User	*Ideal*	$\mathcal{O}(s)$	$\mathcal{O}(s)$	IND-CPA and Pattern
	This paper	$\mathcal{O}(s)$	$\mathcal{O}(D\cdot\log n+s)$	>IND-OCPA, >MC/MR, <IND-CPA and Pattern

require a factor of $\log D$ in communication complexity, because the m records in the D categories/buckets need to be addressed. While related works on range search [6, 15] or OPE [14] have better asymptotic complexities than RASP, an overhead of $\log D$ instead of D, we show in our evaluation in Section 4.3 that RASP's constants are very low, especially compared to related work. RASP only uses symmetric cryptography, i.e., hash functions and block ciphers compared to expensive bilinear pairings and identity based encryption of related work [6, 15]. Also, RASP does not require any expensive $\mathcal{O}(n)$ computation on the cloud side that the surveyor would have to pay for [1], but only a cheap key-value based storage cloud such as Amazon Dynamo [2].

For m-sort, a scheme based on OPE would use, e.g., an m-MinHeap [7] with $\mathcal{O}(n\cdot\log m)$ complexity for the cloud. Again, ORAM-based sorting mechanisms with $\mathcal{O}(n\cdot\log D)$ accesses to the ORAM become quickly too expensive. In contrast to related work, RASP is close to an *Ideal* solution, besides the additional factor of $|\mathcal{U}|\cdot D$ which is small in practice, cf. Section 4.3. Note that recent range search schemes [6, 15] do not support m-sort queries in a straightforward way, so we cannot include them in Table 2.

Table 3 summarizes storage requirements. Being tree based, OPE by Popa et al. [14] requires an additional $\mathcal{O}(\log n)$ storage overhead per ciphertext. Similarly, recent ORAMs are tree based and, with n nodes in the tree, require an overhead factor of either $\mathcal{O}(\log n)$ [16] or $\mathcal{O}(s')$ [17] per ciphertext. Here, s' is an additional security parameter. While the work by Stefanov et al. [17] has superior computational worst-case complexity than Stefanov et al. [16], $\mathcal{O}(\log^2 n)$ compared to $\mathcal{O}(\log^3 n)$, a drawback is

its large memory requirement of $\mathcal{O}(s' \cdot \log^2 n \cdot s)$. In contrast, RASP features only $\mathcal{O}(s)$ ciphertext overhead and $\mathcal{O}(D \cdot \log n + s)$ (for D counters and SK) memory requirements. Finally, RASP offers weaker privacy than ideal IND-CPA and indistinguishable query patterns. However, RASP's privacy notion is stronger than related work's "match concealing", "match revealing" [15] or IND-OCPA.

Bibliography

[1] Amazon. Elastic EC2 Pricing, 2013. `http://aws.amazon.com/ec2/pricing/`.

[2] Amazon. DynamoDB Pricing, 2013. `http://aws.amazon.com/dynamodb/pricing/`.

[3] Anonymized. RASP Source Code, 2013.

[4] M. Bellare. New Proofs for NMAC and HMAC: Security without Collision-Resistance. In *Proceedings of CRYPTO*, pages 602–619, Santa Barbara, USA, 2006. ISBN 3-540-37432-9.

[5] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proceedings of Annual International Cryptology Conference*, pages 578–595, Santa Barbara, USA, 2011. ISBN 978-3-642-22791-2.

[6] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Proceedings of Theory of Cryptography Conference*, pages 535–554, Amsterdam, Netherlands, 2007. ISBN 3-540-70935-5.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009. ISBN 978-0262033848.

[8] R. Curtmola, J.A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 79–88, Alexandria, USA, 2006.

[9] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. ISSN 0004-5411.

[10] Google. A new approach to China, 2010. `http://googleblog.blogspot.com/2010/01/new-approach-to-china.html`.

[11] H. Hacigümüs, B.R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of SIGMOD Conference*, pages 216–227, Madison, USA, 2002. ISBN 1-58113-497-5.

[12] B. Hore, S. Mehrotra, and G. Tsudik. A Privacy-Preserving Index for Range Queries. In *Proceedings of VLDB*, pages 720–731, Toronto, Canada, 2004. ISBN 0-12-088469-0.

[13] J. Li, Maxwell N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of Operating System Design and Implementation*, pages 121–136, San Francisco, USA, 2004.

[14] R.A. Popa, F.H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding, 2013. IACR Cryptology ePrint Archive, `http://eprint.iacr.org/2013/129`.

[15] E. Shi, J. Bethencourt, H.T.-H. Chan, D.X. Song, and A. Perrig. Multi-Dimensional Range Query over Encrypted Data. In *Proceedings of Symposium on Security and Privacy*, pages 350–364, Oakland, USA, 2007.

[16] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proceedings of Network and Distributed System Security Symposium*, San Diego, USA, 2012.

[17] E. Stefanov, M. van Dijk, E. Shi, C.W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path-ORAM: An Extremely Simple Oblivious RAM Protocol, 2013. IACR Cryptology ePrint Archive, `http://eprint.iacr.org/2013/280`.

[18] Techcrunch. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010. `http://techcrunch.com/2010/09/14/google-engineer-spying-fired/`.

A Appendix

Proof (of **Theorem 1***).* We describe a PPT simulator $\mathcal{LS}$ such that for all PPT adversaries $\mathcal{A}$, the outputs of $\mathbf{Real}^{\mathsf{LL}}_{\mathcal{A}}(s)$ and $\mathbf{Sim}^{\mathsf{LL}}_{\mathcal{A},\mathcal{LS}}(s)$ are indistinguishable. Consider the simulator $\mathcal{LS}$ that, given a partial trace of a history H, $\tau(o_1,...,o_i)$, outputs $\mathbf{v}=(\mathcal{K}_i,\mathcal{C}_i)$ as follows. $\mathcal{LS}$ keeps as state, a vector B of length D which contains the most recent contents of each bucket (from the simulator's perspective). B is initialized to all empty sequences, and $\mathcal{LS}$ will update B for each Enumerate which reveals additional bucket records. Additionally, $\mathcal{LS}$ manages list $\mathbf{k}$ which holds the addresses of add operations and an associative array $\mathbf{c}$ which maps addresses to values. $\mathbf{c}$ represents the simulator's view of the store's memory. If $\mathbf{c}$ is evaluated on an address which is empty, it returns $\perp$. If i is the operation $\mathcal{LS}$ is simulating and

1.) $\beta(o_1,...,o_i)[i] = \mathsf{Add}$: $\mathcal{LS}$ sets $\mathbf{k}[i]$ and $\mathbf{c}[\mathbf{k}[i]]$ to uniformly random strings and outputs $\mathcal{K}_i=\{\mathbf{k}[i]\}$ and $\mathcal{C}_i=\{\mathbf{c}[\mathbf{k}[j]]\}$.

2.) $\beta(o_1,...,o_i)[i] = \mathsf{Enumerate}$: $\mathcal{LS}$ creates the sequence $\mathcal{K}'$ such that it contains, in order, every record $\mathbf{k}[x]$ where $\sigma(H)[x,i]=1$. $\mathcal{LS}$ then sets $B_{\gamma(H)[i]}$ to $B_{\gamma(H)[i]}$ concatenated with $\mathcal{K}'$ and a uniformly random string. This can be viewed as the simulator returning all the records from the previous enumerate on the same bucket, plus any additional records that have been added to the bucket since then and finally adding a random empty address on the end (representing the end of a list). $\mathcal{LS}$ then returns $\mathcal{K}_i=B_{\gamma(H)[i]}$ and $\mathcal{C}_i$ equal to $\mathbf{c}$ evaluated on every address in $\mathcal{K}_i$.

Since the outputs of h and E are indistinguishable from random, simulator $\mathcal{LS}$ can put random strings in $\mathcal{K}_i$ and $\mathcal{C}_i$ during adds. Because of the enumeration pattern leakage, $\mathcal{LS}$ can also guarantee the correct pattern in $\mathcal{K}_i$ during enumerates by a simple check of $\beta(H)$. Future enumerates will also return, as a prefix, previous enumerates which guarantees consistency. $\mathcal{LS}$ simulates the end of a linked list by appending a random address and a $\perp$ value to each enumerate. □

Proof (of **Theorem 2***).* We describe a PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$, the outputs of $\mathbf{Real}^{\mathrm{RASP}}_{\mathcal{A}}(s)$ and $\mathbf{Sim}^{\mathrm{RASP}}_{\mathcal{A},\mathcal{S}}(s)$ are indistinguishable. We construct a simulator $\mathcal{S}$ that uses data type LL. Given a partial trace of a history H, $\tau(o_1,...,o_i)$, $\mathcal{S}$ outputs $\mathbf{v}=(\mathcal{K}_i,\mathcal{C}_i)$ as follows:

Using the trace information, $\mathcal{S}$ will translate the encrypt, range search, and sort operations in H into LL's Add and Enumerate operations which can be passed to $\mathcal{LS}$. $\mathcal{S}$ keeps, as state, sequences $\mathbf{b}$, and $\mathbf{g}$, and a matrix $\mathbf{s}$, representing the operation pattern, access pattern, and enumeration pattern respectively, which will be created and passed to $\mathcal{LS}$ as its trace. Generally speaking, encrypt operations will be translated into Adds in a one-to-one manner, and both range searches and sort queries will be translated into one or more Enumerate operations. Therefore, $\mathcal{S}$ will also have a counter x which keeps track of what location in the simulated trace it is at (initialized to 1). If o_i is the operation $\mathcal{S}$ is simulating, and

1.) $\beta(H)[i]=\mathsf{Encrypt}$: $\mathcal{S}$ sets $\mathbf{b}[x]$ to Add, increments x, and returns $\mathcal{LS}(\mathbf{b},\mathbf{s},\mathbf{g})$.

2.) $\beta(H)[i]=\mathsf{RangeSearch}$: Parse $\sigma(H)[i]$ as $\{(b_1,\mathbf{t}_1),...,(b_n,\mathbf{t}_n)\}$, where bs are permuted bucket IDs and $\mathbf{t}$s are indices of related Encrypt operations. $\sigma(H)[i]$ is an unordered set, so $\mathcal{S}$ orders it numerically by b_ℓ from lowest to highest. $\mathcal{S}$ sets $\mathcal{K}_i := \varnothing$ and $\mathcal{C}_i = \varnothing$. $\forall b_\ell$, $\mathcal{S}$ sets $\mathbf{b}[x]$ to Enumerate, sets $\mathbf{g}[x]$ to b_ℓ, $\forall u \in \mathbf{t}_\ell$ sets $\mathbf{s}[x,u]$ to 1,

appends $\mathcal{LS}(\mathbf{b},\mathbf{s},\mathbf{g})$ to $\mathcal{K}_i$ and $\mathcal{C}_i$, and increments x. This creates a series of enumerate operations in the trace for $\mathcal{LS}$ which is linked to all the correct add operations through $\sigma(H)$. $\mathcal{S}$ returns $\mathcal{K}_i$ and $\mathcal{C}_i$.

3.) $\beta(H)[i] = \mathsf{SortQuery}$: Parse $\gamma(H)[i]$ as $(m,(b_1,...,b_n))$. $\mathcal{S}$ proceeds as for $\mathsf{RangeSearch}$, but instead of ordering the enumerates by the random permutation π, it orders them according to the leakage from $\gamma(H)$ (i.e., in the order of the underlying categories).

$\mathcal{S}$, according to the algorithms for range search and sorting, emulates a number of add and enumerate queries which are to be done by the update-oblivious linked list data structure. Since we can translate every encrypt, range search or sort operation directly into one or more add or enumerate operations, $\mathcal{S}$ returns exactly the output of simulator $\mathcal{LS}$. Therefore, if the linked list data structure is secure and can be simulated by simulator $\mathcal{LS}$, the output of $\mathcal{S}$ is indistinguishable from the output of the real protocol. □

Table 4. RASP evaluation results, timings [ms] are *per record*.

	$D=256$		$D=8192$	
	$\lvert\mathcal{U}\rvert=100$	$\lvert\mathcal{U}\rvert=1000$	$\lvert\mathcal{U}\rvert=100$	$\lvert\mathcal{U}\rvert=1000$
Encrypt	64		65	
Range Query 1% ; 5%	0.68 ; 0.73	0.92 ; 1.18	0.79 ; 0.93	1.06 ; 1.37
Sort Query *top*-50 ; *top*-500	29.58 ; 29.36	50.62 ; 49.59	421 ; 443	546 ; 564