Practical Forward-Secure Range and Sort Queries with Update-Oblivious Linked Lists

ABSTRACT

We revisit the problem of privacy-preserving range search and sort queries on encrypted data in the face of an untrusted data store. Our new protocol RASP has several advantages over existing work. First, RASP strengthens privacy by ensuring *forward security*: 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 $[a,b]$. We are able to accomplish this using only traditional hash and block cipher operations, abstaining from expensive asymmetric cryptography and bilinear pairings. Consequently, RASP is highly practical, even for large database sizes. Additionally, we require only cloud *storage* and not a computational cloud like related works, which can reduce monetary costs significantly. At the heart of RASP, we develop 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 on Amazon Dynamo, demonstrating its efficiency and real world applicability.

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 completely trusted, due to, for example, hacker and insider attacks [12, 25]. While encryption of outsourced data protects against many privacy threats in cloud scenarios, it renders subsequent operations on data (i.e., data analysis) extremely difficult. 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 can negate any cloud cost advantages.

In this paper, we address the problem of performing privacy-preserving range search and sort queries on encrypted outsourced data with a particular focus on practicality. We envision a 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 values ("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 observing multiple range and sort operations. For example, the cloud could learn access patterns and correlate them. Consequently, also the analysis operations ("queries") need to be privacy protected. The crucial challenge here is practicality, i.e., high efficiency in terms of *bandwidth* and *memory* requirements as well as user/surveyor/cloud *computations*.

Related Work Besides FHE, another approach that could apply here is performing range or sort queries on top of an Oblivious RAM [11]. However, with n the total number of outsourced records in the cloud, ORAM worst-case communication complexity is poly-logarithmic in n [20, 24]. Thus, for large n (such as $n=2^{30}$ records or more), this overhead becomes unacceptably expensive for sorting and range search. Along the same lines, searchable encryption techniques [7, 10, 21] would either require computational and communication complexities linear in n or non-trivial extensions to perform updates to stored data in a privacy-preserving fashion. Other techniques such as Order Preserving Encryption (OPE) [5] are highly efficient, but provide weak privacy. Finally, recent work on range search [6, 16, 19, 26] offers insufficient "selective" privacy and, being based on bilinear pairings, becomes impractical for, e.g., embedded devices or smartphones and large n as targeted in this paper. None of the above related works support multiple users. Note that we discuss insufficiency of related work in great detail at the end of the paper in Section 5.

Our contributions. 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. As with 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. While RASP targets practicality and offers weaker privacy properties than, e.g., ORAM, it provides stronger, forward-secure privacy compared to related work on range search [6, 16, 19, 26]. Moreover, RASP only relies on efficient computations such as hashing and symmetric encryption, in contrast to expen-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

sive pairing-based related work. The technical highlights of this paper are:

- LL, a dynamic data structure that provably hides any information about a newly added data record until this data record is explicitly read. (Section 3)
- RASP, a protocol employing LL for range search and sort queries on encrypted data in the cloud with support for multiple users. Compared to related work on range search, RASP offers stronger, forward-secure privacy. We formally prove that the cloud cannot learn any information about records added until the surveyor queries them. Details about queries are hidden, and only the overlap between queries is leaked. RASP is efficient and scales well. The users' and surveyor's, computational and communication complexities are constant in the total number of records. In contrast to related work, RASP seamlessly integrates into cheap storage-only, no computation cloud services such as Amazon S3 or Dynamo and allows for multiple different users that do not trust each other. (Section 4)
- An implementation and evaluation of LL and RASP in Amazon's DynamoDB cloud. The source code is available for download [3].

2. PROBLEM STATEMENT

General scenario: 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 general setup above. 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, and date. At times, 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. Once in a while, 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.

Trust Model: We only assume that the surveyor is trusted by the users. The cloud server is untrusted, and any user can collude with the server and they should not be able to reveal any data besides their own. Additionally, users should not be able to reveal additional data by colluding with each other.

2.1 Range and Sort Queries

We now formalize privacy-preserving range and m-sort schemes. We start by introducing the functionality that each scheme should support. The main idea is that users respectively encrypt and upload their records to the store, while the surveyor performs range search and sort operations.

DEFINITION 1 (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 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 a 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* $\mathsf{Decrypt}(\mathsf{Encrypt}(I, M, Seed_i), SK, i) = M$*, 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 $\mathcal{T}^R$.*
- $\mathsf{RangeQuery}(\mathcal{T}^R, \{C_1, \ldots, C_n\})$*: 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 $I_i \in [a, b]\}$ is computed.*
- $\mathsf{PrepareSortQuery}(m, SK)$*: with secret key SK and length $m, 1 \leq m \leq n$, outputs a sort query token $\mathcal{T}^S$.*
- $\mathsf{SortQuery}(\mathcal{T}^S, \{C_1, \ldots, C_n\})$*: using sort query token $\mathcal{T}^S$ and ciphertexts C_i, outputs a sequence $S^S = < C'_1, C'_2, \ldots, C'_m >$ with $C'_i \in \{C_1, \ldots, C_n\}$ 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

Overview: 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 its poly-logarithmic worst-case communication complexity [20, 24] quickly becomes expensive with large n such as $n = 2^{30}$. Additionally, as ORAM does not allow multiple users, a straightforward extension to range search and sort is vulnerable to collusions.

RASP targets privacy that is slightly weaker than ORAM's privacy, but still stronger than privacy provided by related work on range search. We call this privacy *forward-secure*. 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 when the

surveyor executes a range search or sort query will $\mathcal{A}$ 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 formalize our privacy goal. Targeting a standard 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.

In summary, forward-secure privacy allows $\mathcal{A}$ 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. As we will discuss later in Section 2.3 this forward-secure privacy is actually stronger than the privacy offered by related work on range search [6, 16, 19, 26].

We focus on key-value "(k, v)" based cloud stores/databases such as Amazon Dynamo DB or S3 in this paper, so we assume that each record is uniquely addressable by an *address* k in the store. We refer to the keys in the key-value paradigm as addresses to avoid the confusion with cryptographic keys.

2.2.1 Formal Definitions

Following Curtmola et al. [10], we formalize privacy by quantifying the information leakage of a scheme Π.

DEFINITION 2 (OPERATION). *For range and m-sort scheme Π, an operation* op *is defined as either* $(\mathsf{Encrypt}, I, M, Seed_i)$ *or* $(\mathsf{RangeQuery}, a, b, SK)$ *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.*
- $\mathsf{Execute} : (\mathsf{op}, K) \rightarrow (\mathcal{K} = (k_1, ..., k_t), \mathcal{C} = (c_1, ..., c_t))$ *which executes* op *using the key K 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). *The q-query history H 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 $\beta(H)$ induced by a q-query history H is defined as the sequence $\beta(H) = (\mathsf{Type}(o_1), ..., \mathsf{Type}(o_q))$.*

The operation pattern is a very mild leakage, only telling the adversary whether we are doing range search or sort queries.

DEFINITION 5 (QUERY PATTERN). *Let π be a random permutation of integers $\{1, \ldots, D\}$. The query pattern of a q-query history is the q-length sequence $\sigma(H)$ defined as follows: first, consider the case that* $\mathsf{Type}(o_i) = \mathsf{RangeQuery}$ *or* $\mathsf{Type}(o_i) = \mathsf{SortQuery}$. *Let* $\mathsf{Execute}(o_i, K) = ((k_1, \ldots, k_t), (c_1, \ldots, c_t))$ *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 $\perp$.*

That is, $\sigma(H)$ reveals which *previous* Encrypt operations are being queried as part of the *current* range search or sort query operation i (as in related work) 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. Note that this leakage is also relatively weak: all existing schemes other than Oblivious RAM leak the intersection of the results of two queries, and the category overlap can be determined from this with a high degree of accuracy if the adversary has some knowledge of the plaintext distribution.

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 (FORWARD-SECURE PRIVACY). *Let Π be a 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 $\mathbf{Real}^{\Pi}_{\mathcal{A}}(s)$ and $\mathbf{Sim}^{\Pi}_{\mathcal{A}}(s)$:*

$\mathbf{Real}^{\Pi}_{\mathcal{A}}(s)$
$K = (SK, \{Seed_i\}_{1 \le i \le |\mathcal{U}|}) \leftarrow \mathsf{KeyGen}(1^s)$
for $1 \le i \le q$
  $(st_{\mathcal{A}}, o_i) \leftarrow \mathcal{A}(st_{\mathcal{A}}, (\mathcal{K}_1, \mathcal{C}_1), ..., (\mathcal{K}_{i-1}, \mathcal{C}_{i-1}), 1^s)$
  $(\mathcal{K}_i, \mathcal{C}_i) \leftarrow \mathsf{Execute}(o_i, K)$
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 $1 \le i \le q$
  $(st_{\mathcal{A}}, o_i) \leftarrow \mathcal{A}(st_{\mathcal{A}}, (\mathcal{K}_1, \mathcal{C}_1), ..., (\mathcal{K}_{i-1}, \mathcal{C}_{i-1}), 1^s)$
  $(\mathcal{K}_i, \mathcal{C}_i, st_{\mathcal{S}}) \leftarrow \mathcal{S}(st_{\mathcal{S}}, \tau(o_1, ..., o_i), 1^s)$
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 forward-secure privacy-preserving, iff *for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$, such that for all PPT distinguishers $\mathcal{D}$,*

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

2.2.2 Discussion

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 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.

2.3 Forward-Secure privacy vs. privacy of related work

We stress that forward-secure privacy, Definition 8, is *stronger* than privacy models of related work on range search [6, 16, 19, 26]. These schemes offer only *selective security*. That is, a query for a specific range $[a,b]$ implies that $\mathcal{A}$ from then on will be able to automatically determine whether new records added in future will also be within $[a,b]$ or not – a major disadvantage. 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. Moreover, the work by Shi et al. [19] is *Match Revealing* (selective-MR). This means that, if a record matches, its category is leaked to $\mathcal{A}$, too. In contrast, the work by Boneh and Waters [6] and Lu [16] is *Match Concealing* (selective-MC). Definition 8 never leaks the category to $\mathcal{A}$, and therefore is Match Concealing. Additionally, Lu [16] requires that the adversary must not know the distribution of records' categories. As the ciphertexts are stored in a B-tree, visible to the store, knowledge of the category distribution is enough to reveal all ciphertexts, cf. Lu [16], §9. This is a rather significant drawback, as there are many useful situations where the adversary may either partially or fully know the plaintext distribution.

Definition 8 is also *stronger* than Order Preserving Encryption (IND-OPE). In IND-OPE, $\mathcal{A}$ immediately learns the order of records. After a range query, $\mathcal{A}$ can determine for any record added in the future whether it is in the (encrypted) range or not. Also, $\mathcal{A}$ learns for *any* record whether it is smaller or larger than a range's (encrypted) endpoint. In contrast, Definition 8 only leaks membership to an encrypted range of encrypted records for those records queried at the time of the query, but not for updates (therefore called "update-oblivious"). Targeting practicality, Definition 8 is weaker than the one of ORAM. In contrast to ORAM, Definition 8 does not protect access patterns. That is, $\mathcal{A}$ can observe that subsequent queries access the same records. However, as noted before, in addition to its inefficiency ORAM is vulnerable to collusion attacks as it does not support multiple users.

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 at the least 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 cloud store. Each "value" v added is uniquely addressable by an address k in the store. More formally, an add-enumerate data type supports 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 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). *An enumeration pattern induced by a q-query history H is the $q \times q$ 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 $\{1,...,D\}$. 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. We stress that this is also revealed as part of the enumeration pattern, and so is not additional leakage, but we include this separate notation for clarity and ease of exposition in our proof.

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.

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

$C:=\mathsf{Get}(h_\kappa(\text{“delta”}))$; // $C=\mathsf{Enc}_\kappa(\Delta)$
$\Delta=(\delta_1,...,\delta_D):=\mathsf{Dec}_\kappa(C)$;
if $\delta_I=1$ **then**
  $\gamma_I^{\text{Writer}}:=\gamma_I^{\text{Writer}}+1$;
  $\psi_I:=h_\kappa(I,\gamma_I^{\text{Writer}})$;
  $\delta_I:=0$;
  $\mathsf{Put}(h_\kappa(\text{“delta”}),\mathsf{Enc}_\kappa(\Delta)=\mathsf{Enc}_\kappa(\delta_1,...,\delta_D))$;
end
$k:=\psi_I$;
$\psi_I \xleftarrow{\$} \{0,1\}^s$;
new Record e;
$e.\text{value}:=\mathsf{Enc}_\kappa(v)$; $e.\psi:=\mathsf{Enc}_\kappa(\psi_I)$;
$\mathsf{Put}(k,e)$;
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 $\mathcal{S}=\{c_i|c_i\in I\}$

$\mathcal{S}:=\varnothing$;
$C:=\mathsf{Get}(h_\kappa(\text{“delta”}))$; // $C=\mathsf{Enc}_\kappa(\Delta)$
$\Delta=(\delta_1,...,\delta_D):=\mathsf{Dec}_\kappa(C)$;
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**
  $\text{start}:=h_\kappa(I,\gamma_I^{\text{Reader}})$;
  $\mathcal{S}:=\mathcal{S}\cup\mathsf{Retrieve}(\text{start},\kappa)$;
  $\delta_I:=1$;
  $\mathsf{Put}(h_\kappa(\text{“delta”}),\mathsf{Enc}_\kappa(\Delta)=\mathsf{Enc}_\kappa(\delta_1,...,\delta_D))$;
  $\gamma_I^{\text{Reader}}:=\gamma_I^{\text{Reader}}+1$;
return $(\mathcal{S})$;

Algorithm 3: LL-Retrieve(ψ,κ)

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

$\mathcal{S}:=\varnothing$;
Record $e:=\mathsf{Get}(\psi)$;
while $e\neq\perp$ **do**
  $\mathcal{S}:=\mathcal{S}\cup e.\text{value}$;
  $\psi:=D_\kappa(e.\psi)$;
  $e:=\mathsf{Get}(\psi)$;
return $\mathcal{S}$;

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 v, 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 simply append v to I, but creates a new *chain* for I, adds v 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. The **security rationale** for starting a new chain for I after an enumeration of I is that $\mathcal{A}$ cannot determine category I for a newly added value by linking to a previous enumerate of I.

Details: Reader and Writer share a secret key κ that, for simplicity, has been exchanged in advance. 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.

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 $\mathsf{Enc}_\kappa(\Delta)$, where Enc 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 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 $\delta_I=0$, 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 Enc. 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

For our proof, we use the notion of IND$-CPA encryption from Rogaway [18]. Informally, this definition means that an encryption scheme $(\mathsf{Enc},\mathsf{Dec})$ is indistinguishable from an oracle which produces random strings of the same length as a ciphertext. 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* Enc *is an IND$-CPA encryption, then* LL *is update-oblivious.*

PROOF. 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 Enc 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. □

Resiliency to Collusion Attacks: Since every user has their own key, independent of the other users, it is simple to see that users cannot collude with each other or with the cloud server to learn anything beyond their own data.

Extensions to Mitigate Consistency Attacks: Contrary to previous work on range search [6, 16, 19, 26], we allow *different* entities to access data: the Reader and the Writer. As the two entities synchronize using Δ, a fully malicious adversary could mount attacks by desynchronization, such as sending outdated versions of Δ.

So far, our definitions of update-oblivious and of data type LL above have implicitly required *read-after-write* consistency. Yet, although the Reader has read bucket I and set $\delta_I := 1$, $\mathcal{A}$ could send an old version of $\mathsf{Enc}(\Delta)$ 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.

Thus, we now show how we can easily extend our system to cope even with adversaries $\mathcal{A}$ mounting consistency attacks. Inspired by Li et al. [15], 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, the Reader increases its counter. For each Add, the Writer increases its counter. Both parties keep local copies of counters, compare to Δ's counters upon receipt, and therewith verify the freshness of Δ. Even in the face of fully malicious $\mathcal{A}$ mounting consistency attacks on the (augmented) Δ, this approach achieves *Fork Consistency*, the strongest consistency possible in the absence of a third trusted party [15]. In short, after such an attack, Surveyor and User will be in different "worlds": no change performed to the data will ever be seen by the Surveyor. The surveyor remains at the state of the old, not-updated data set, however with full privacy guarantees. For more details on this technique, we refer to Li et al. [15]. Note that in scenarios similar to related work with only *one* entity to read and write to the store, LL would not synchronize Δ, making consistency attacks impossible.

Generalization: The update-oblivious property can be extended to other data types. Let a *monotonically-expanding* data type S be any data type supporting 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 trees) can be viewed as a collection of edges. The Add adds edges to the collection, while Enumerate lists edges (or properties of edges). Our bucket data type LL can be used to implement 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 algorithms PrepareRangeQuery and RangeQuery intertwined. Similarly, for a sort query, he executes algorithms PrepareSortQuery and SortQuery intertwined. In the algorithms below, Enc and Dec are IND$-CPA 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 input padding.

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.

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 encrypted data to the key-value store.

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

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

```
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 := Dec_{Seed_i}(C);
return M;
```

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 $S^R = \{C_i\}$

```
S^R := ∅;
for i := 1 to |U| do
    for j := a to b do
        Seed_i := h_SK(i);
        S^R := S^R ∪ LL-Enumerate(π_{Seed_i}(j), Seed_i);
return S^R;
```

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 $\{a,\ldots,b\}$. To retrieve all data of user U_i in bucket $\pi(j)_{\text{Seed}_i}$, the surveyor uses LL-Enumerate($\pi_{\text{Seed}_i}(j)$, 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($\mathcal{T}^S$, $\{C_1, \ldots, C_n\}$) Similar to range search queries, we consolidate PrepareSortQuery and SortQuery together in Algorithm 8. Again, the surveyor starts

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

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

```
for i := 1 to |U| do
    Seed_i := h_SK(i);
S^S := ∅; i := 1; pos := 1;
while m > 0 and pos ≤ D do
    S' := LL-Enumerate-UpTo(pos, Seed_i, m);
    m := m − |S'|;
    S^S := S^S ∪ S';
    if i = |U| then
        i := 1;
        pos := pos + 1;
    else
        i := i + 1;
return S^S;
```

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 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 forward-secure privacy-preserving.*

PROOF. We describe a PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$, the outputs of $\mathbf{Real}_{\mathcal{A}}^{\text{RASP}}(s)$ and $\mathbf{Sim}_{\mathcal{A},\mathcal{S}}^{\text{RASP}}(s)$ are indistinguishable. We construct a simulator $\mathcal{S}$ that uses data type LL. Given a partial trace of a history H, $\tau(o_1,\ldots,o_i)$, $\mathcal{S}$ 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 (initially set to 1). If o_i is the operation $\mathcal{S}$ is simulating, and

1.) $\beta(H)[i] =$ 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 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. □

4.3 Evaluation

We have implemented RASP in Java, and the source code is available for download [3]. As RASP does not require any computation on the store, our implementation uses the standard Amazon Dynamo DB cloud as the underlying key-value store. 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, we configured a database supporting 3000 Get and 1000 Puts per second and *read-after-write consistency*. Such a database would cost $\approx$\$680 per month [2]. As encryption Enc, we use 128 Bit AES in CBC-mode and HMAC with SHA-1 as 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 (Algorithm 5, including upload), Range Queries, and Sort Queries on a Dynamo data store with a fixed number of $n = 2^{25}$ records. We have set the number of users $|\mathcal{U}|$ to 100 and varied $D = \{64, 256, 1024\}$, and m from .01% to .4% of n for range queries. For m-sort, we varied m from 32 to 512. Data is distributed into buckets according to a Gaussian distribution to better represent a real world scenario. In practice, the most interesting pieces of data are usually in the tail of the distribution, and most data is distributed normally. However, this choice does not significantly effect the running time of our scheme as it depends only on m and D. The uneven values of m for range search are the result of querying ranges separated by one bucket when $D = 64$ (the smallest unit of change that would be evenly divisible for all values of D). Since the data is Gaussian, increasing the range by one bucket increases m by an uneven amount. 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%.

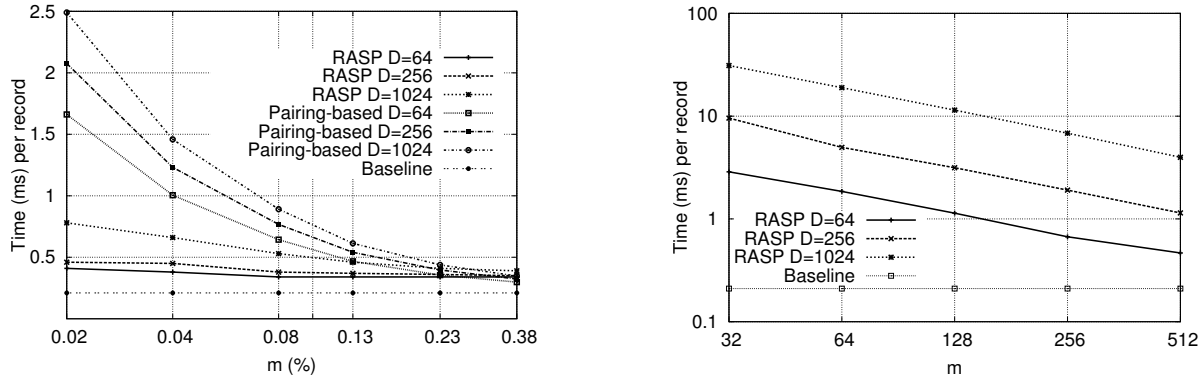
Figure 1 sums up our evaluation and presents timings in ms *per record*. All timings are dominated by network latency. In Dynamo, a single Get takes 31 ms, and a Put takes 39 ms. In contrast, (Java Bouncy Castle) AES encryption (≈ 14 μs) and HMAC evaluation (≈ 6 μs) are comparably fast on our setup. In comparison, a typical Type-A 512 bit pairing [8] required by related work [6, 16, 19, 26] is 3 orders of magnitude more expensive and takes ≈ 8 ms, an exponentiation takes ≈ 11 ms. We stress that, in contrast to other work [16], our evaluation *does not* rely on (expensive) cryptographic hardware acceleration running on the user, surveyor, or cloud store. We established a baseline (≈ 0.21 ms) by testing the amount of time it takes to retrieve a record from Dynamo without any encryption or secure data structure. This was done using Dynamo's built-in range search capability and represents a lower bound for any range search algorithm. A single Encrypt of a single user in RASP takes ≈ 64 ms in our configurations.

Times for single Get, Put, and Encrypt are significantly higher compared to Range and Sort queries for the surveyor *per record*, although the same operations (encryption, hash, network access) are required. This is due to the fact that during, e.g., a range query, RASP allows the surveyor to enumerate multiple LL-buckets in *parallel* with Dynamo. Dynamo allows to issue multiple Get-requests from multiple threads in parallel, reducing the total response time significantly. We estimate the single record upload time in related work, e.g., Lu [16] to ≈ 589 ms for $D = 256$ ($2 + 6 \cdot \log D$ exponentiations + 1 Put) which is notably higher than RASP.

Timings for range queries, Fig. 1a, and sort queries, Fig. 1b, 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 downloaded and processed. Sort is more expensive than range search, because access to buckets cannot be parallelized: to find the first m records, buckets need to be accessed sequentially. However, sort has an additional security property that is not present with range search. If the end of a sort query lands in a bucket which contains entries for more than one user, not all of the users' data will be revealed (only enough to satisfy the sort query). In exchange for this additional privacy, sort queries are significantly more expensive. If this additional privacy is not needed, one can accomplish the same thing by issuing sequential range queries for each bucket until enough results are returned to satisfy the m-sort. This will achieve similar performance per record to the much faster range sort.

To put RASP's range search into perspective, we determined the performance for Lu [16] by using JPBC [8] to find a cost for one pairing on our hardware and then multiplying by the number of pairings needed per query in their scheme. We then divided by the number of records returned to get an amortized query computation cost per record and finally added the baseline communication cost as outlined above. Note that our comparison is for RASP with 100 users and related work with just a single user. When related work is modified for multiple users, the straightforward way which allows for the same cross-user privacy guarantees as RASP (making a separate index for each user) imposes over a 20 times slowdown. We compare against our scheme with 100 users to show that we can scale to a multi-user setting with minimal performance degradation.

In comparison with Lu [16], range search with RASP is several times faster when D or m are small, but also comes very close to the optimal baseline when m is larger, see Fig. 1a. Additionally, RASP is faster at adding records by an order of magnitude. As the costs for exponentiation and pairing based related work is non-negligible even on our powerful hardware, we conclude that its use in scenarios with embedded devices, smartphones, and large n is limited. On the server side, RASP does not require any computation to be performed at the store and so can be run on a cloud storage service without any costly computational resources.



(a) Range Queries

(b) Sort Queries

Figure 1: RASP evaluation results

5. RELATED WORK AND SUMMARY

ORAM Privacy-preserving range search and sort queries could be realized based on, for example, ORAM [11] protocols. The idea would be to simply encrypt all records, store them in an ORAM (e.g., in sorted order), and let the surveyor perform the range search on the ORAM. This results in strong 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($\log n$) [20, 24]. We do not include "ObliviStore" [22] in our comparison, because, fundamentally, it is only an implementation of the $\sqrt{n}$ complexity ORAM by Stefanov et al. [23]. While also targeting practicality, it requires either trusted hardware on the data store or a private cloud for the user and is therefore difficult to compare to our setup. Currently, Wang et al. [27] are using ORAM to develop general oblivious data structures. While they are able to reduce complexity for, e.g., a search tree down to $O(\log n)$ with $O(\log n)$ client memory, this is still more than RASP, cf. Table 1.

OPE Alternatively, using OPE [5, 17], ciphertexts retain the order of their underlying plaintexts, making range search and sorting straightforward. However, OPE gives weak privacy, as the relationship between ciphertexts is immediately visible to the cloud.

Searchable Encryption Schemes for general search on encrypted data, see seminal work by Song et al. [21] or Boneh et al. [7], or see Curtmola et al. [10] for an overview, can be extended in a straightforward way to perform range search. For example, each data record could be encrypted with the category it belongs to. Such constructions would have the (prohibitive) drawback of being linear in n. Any more efficient approach, e.g., enumerating over categories as RASP, would need to solve the (non-trivial) problem of being forward-secure. One might apply RASP to allow for such privacy, resulting in a protocol comparable to RASP. Similarly, it is non-trivial to extend searchable encryption schemes to allow for sort in a privacy-preserving, yet practical way and, consequently, deserves its own research.

Range Search Some related work focuses especially on privacy-preserving range search. For example, Hacigümüs et al. [13] and Hore et al. [14] encrypt records and put ciphertexts 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. Other works [6, 16, 19, 26] 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 [6, 16, 26] are selective match concealing (selective-MC), [19] is selective match revealing (selective-MR), i.e., the cloud will learn the category of a record that matches a range query. Moreover, these schemes make use of computationally expensive bilinear pairings. Similarly, recent work by Wand et al. [26] uses asymmetric cryptography to overcome a certain privacy leakage in multi-dimensional range search. However, based on R-trees, its range search worst-case complexity is in $O(n)$.

Comparison.

As RASP offers not only range search, but also sort capabilities together with stronger (forward-secure) privacy than related range search schemes or OPE, it is difficult to compare its performance. Still, to put things into perspective, we sum up RASP's asymptotic performance and contrast it to related work in tables 1 and 2.

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 require a significant redesign. A straightforward extension adds a factor of $|\mathcal{U}|$ in tables 1 and 2, which renders such approaches overly costly.

Computation and Communication 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 symmetric key ciphertexts are exchanged, and s' indicates asymmetric key ciphertexts. In each table, we compare to an *ideal* solution, representing a lower bound for each complexity, respectively.

ORAM provides a regular RAM interface, so any operation can be done by simply using the same "ideal" algorithm that would be used on unencrypted data. The overall cost of this operation will then be the same as the ideal, but with a poly-logarithmic overhead specific to the ORAM implementation. For example, In the case of an Insert, an ideal solution would be using an interval tree [9]. OPE and an *Ideal* solution all require (a factor of) $\log D$ communication complexity, because the m records in the D categories/buckets need to be addressed.

Note: It is important to point out that, while related work on range search [6, 16, 19] or OPE [17] has better asymptotic complexities than RASP ($\log D$ vs. D), RASP is linear in D only due

Table 1: Worst-case complexities for range search and privacy comparison. n: total number of records, m: number of records queried D: size of records' 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. Note: RASP is linear in D only due to bit array Δ which is very small in practice.

		Range				Privacy
		Computation per Query			Communication per Query	
		Insert User	Surveyor	Cloud		
Single User	OPE [17]	$O(\log n + \log D)^{\ddagger}$	$O(\log n + \log D + m)^{\ddagger}$	—	$O((\log n + \log D + m) \cdot s)$	IND-OCPA
	ORAM [20]	$O(\log D \cdot \log^3 n)^{\ddagger}$	$O((\log D + m) \cdot \log^3 n)^{\ddagger}$	—	$O((m + \log D) \cdot \log^3 n \cdot s)$	IND-CPA and Pattern
	[24]	$O(\log D \cdot \log^2 n)^{\ddagger}$	$O((\log D + m) \cdot \log^2 n)^{\ddagger}$	—	$O((m + \log D) \cdot \log^2 n \cdot s)$	
	Range Search [6]	$O(D)^{\ddagger,\diamond}$	$O(m)^{\ddagger,\diamond}$	$O(n)^{\diamond}$	$O(m \cdot s)$	selective-MC
	[16]	$O(\log D)^{\ddagger,\diamond}$	$O(\log D + m)^{\ddagger,\diamond}$	$O(\log D \cdot \log n + m)^{\diamond}$	$O(s' \cdot \log D + s \cdot m)$	selective-MC
	[19]	$O(\log D)^{\ddagger,\diamond}$	$O(\log D + m)^{\ddagger,\diamond}$	$O(n)^{\diamond}$	$O(m \cdot (\log D + s))$	selective-MR
Multi User	*Ideal*	$O(\log D)$	$O(\log D + m)$	—	$O((m + \log D) \cdot s)$	IND-CPA and Pattern
	This paper	$O(D)^{\ddagger}$	$O(\|\mathcal{U}\| \cdot D + m)^{\ddagger}$	—	$O((\|\mathcal{U}\| \cdot D + m) \cdot s)$	Forward-Secure

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

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

		m-Sort		
		Computation per Query		Communication per Query
		Surveyor	Cloud	
Single User	OPE [17]	$O(m)^{\ddagger}$	$O(\log n + m)$	$O(m \cdot s)$
	ORAM [20]	$O(m \cdot \log^3 n)^{\ddagger}$	—	$O(m \cdot \log^3 n \cdot s)$
	[24]	$O(m \cdot \log^2 n)^{\ddagger}$	—	$O(m \cdot \log^2 n \cdot s)$
Multi User	*Ideal*	$O(m)$	—	$O(m \cdot s)$
	This paper	$O(\|\mathcal{U}\| \cdot D + m)^{\ddagger}$	—	$O((\|\mathcal{U}\| \cdot D + m) \cdot s)$

‡ Involves Symmetric Cryptography

to synchronization array Δ. In practice, bit array Δ is very small, especially compared to a single patient record. For example, with $D = 8192$ categories, $|\Delta| = 1$ KByte resulting in only 64 AES operations. In all practical scenarios as targeted in this paper ($n \gg D$) the linear number of AES operations will outperform $\log D$ exponentiations and pairing operations.

Refering to our evaluation in Section 4.3, we indicate that RASP's constants are very low, using only symmetric cryptography, i.e., hash functions and block ciphers. Also, RASP does not require any expensive $O(n)$ 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 [17] would just parse the OPE tree and send the m records. Again, ORAM-based sorting mechanisms with $O(m)$ accesses to the ORAM (assuming records are already sorted, e.g., in an interval tree) 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, 16, 19, 26] do not support m-sort queries in a straightforward way, so we cannot include them in Table 2. While extending range search schemes to support sorting in an efficient, yet secure (e.g., forward-secure) way might be possible, it is far from straightforward.

Privacy RASP's forward-secure privacy notion is stronger than related work's selective match concealing or selective match revealing [19] or IND-OCPA as discussed in Section 2.3. Yet, RASP offers weaker privacy than ideal IND-CPA and indistinguishable query patterns.

Storage Finally, we briefly summarize storage requirements. Being tree based, OPE by Popa et al. [17] requires an additional $O(\log n)$ storage overhead per ciphertext. Similarly, recent ORAMs are tree based and, with n nodes in the tree, require an overhead factor of either $O(\log n)$ [20] or $O(s')$ [24] per ciphertext. Here, s' is an additional security parameter. While the work by Stefanov et al. [24] has superior computational worst-case complexity than Shi et al. [20], $O(\log^2 n)$ compared to $O(\log^3 n)$, a drawback is its large memory requirement of $O(s' \cdot \log^2 n \cdot s)$. In contrast, RASP features only $O(s)$ ciphertext overhead and $O(D \cdot \log n + s)$ (for D counters and SK) memory requirements.

6. CONCLUSION

RASP addresses privacy-preserving range search and sort on outsourced, encrypted data. RASP offers stronger privacy than related work as well as support for multiple, non-trusted users. RASP builds on top of LL, a new dynamic data structure (of independent interest) for privacy-preserving add/enumerate operations. Both, RASP and LL seamlessly integrate into cheap, real world storage-only cloud services such as S3 or DynamoDB. Abstaining from pairings and exponentiations, our protocols target practicality. Our performance evaluations show that, even without hardware acceleration support, RASP offers substantially better performance than recent $\log D$ range search techniques for some interesting settings, in addition to requiring only storage capabilities and not cloud computation (a significant cost savings).

References

[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 for submission. RASP Source Code, 2014. https://www.dropbox.com/s/5am2r9g9u4coe2d/rasp.zip.

[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 CRYPTO*, pages 578–595, Santa Barbara, USA, 2011.

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

[7] D. Boneh, G. DiCrescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt*, pages 506–522, Barcelona, Spain, 2004.

[8] A. De Caro and V. Iovino. jPBC: Java pairing based cryptography. In *Proceedings of Symposium on Computers and Communications*, pages 850–855, Kerkyra, Greece, 2011.

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

[10] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. IACR ePrint Archive, 2006. http://eprint.iacr.org/2006/210.

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

[12] Google. A new approach to China, 2010. http://googleblog.blogspot.com/.

[13] 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.

[14] 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.

[15] 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.

[16] Y. Lu. Privacy-preserving Logarithmic-time Search on Encrypted Data in Cloud. In *Proceedings of Network and Distributed System Security Symposium*, San Diego, USA, 2012.

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

[18] P. Rogaway. Nonce-Based Symmetric Encryption. In *Proceedings of FSE*, pages 348–359, Delhi, India, 2004. ISBN 3-540-22171-9.

[19] 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.

[20] E. Shi, T.-H.H. Hubert Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of ASIACRYPT*, pages 197–214, Seoul, S. Korea, 2011.

[21] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of Symposium on Security and Privacy*, pages 44–55, Berkeley, USA, 2000.

[22] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *Proceedings of Symposium on Security and Privacy*, pages 253–267, Berkeley, USA, 2013.

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

[24] 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 ePrint Archive, http://eprint.iacr.org/2013/280.

[25] Techcrunch. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010. http://techcrunch.com/.

[26] B. Wand, Y. Hou, M. Li, H. Wang, and H. Li. Maple: Scalable Multi-Dimensional Range Search over Encrypted Cloud Data with Tree-based Index, 2014. To appear in ASIACCS'14, preprint available at http://digital.cs.usu.edu/~mingli/papers/Wang_asiaccs14.pdf.

[27] X. Wang, K. Nayak, C. Liu, E. Shi, E. Stefanov, and Y Huang. Oblivious Data Structures, 2014. IACR ePrint Archive, https://eprint.iacr.org/2014/185.