Adaptive Indexing over Encrypted Numeric Data

Panagiotis Karras[◊] Artyom Nikitin[◊] Muhammad Saad[◊]
Rudrika Bhatt[§] Denis Antyukhov[◊] Stratos Idreos[¶]
[◊]Skoltech [§]Rutgers University [¶]Harvard University

ABSTRACT

Today, outsourcing query processing tasks to remote cloud servers becomes a viable option; such outsourcing calls for encrypting data stored at the server so as to render it secure against eavesdropping adversaries and/or an honest-but-curious server itself. At the same time, to be efficiently managed, outsourced data should be indexed, and even adaptively so, as a side-effect of query processing. Computationally heavy encryption schemes render such outsourcing unattractive; an alternative, Order-Preserving Encryption Scheme (OPES), intentionally preserves and reveals the order in the data, hence is unattractive from the security viewpoint. In this paper, we propose and analyze a scheme for lightweight and indexable encryption, based on linear-algebra operations. Our scheme provides higher security than OPES and allows for range and point queries to be efficiently evaluated over encrypted numeric data, with decryption performed at the client side. We implement a prototype that performs incremental, query-triggered adaptive indexing over encrypted numeric data based on this scheme, without leaking order information in advance, and without prohibitive overhead, as our extensive experimental study demonstrates.

1. INTRODUCTION

We have entered the database-as-a-service era; data management capabilities need not be present at the data owner's locale, but can be provided *as a service* by a cloud-based service provider, to which the data is *outsourced*. This model provides users (i.e., the data owner and its trusted clients) power to create, store, modify, and retrieve data from anywhere in the world; the discussion about the applicability of this model has entered domains such as high-frequency trading, whereby a firm can use cloud providers' servers to test trading strategies, run time series analysis, assess risks, and even execute trades, while collecting financial data daily, or on a finer time scale [30, 2]. At the same time, such a model raises security and confidentiality concerns; to follow the same application area, a firm may deploy trading, analytics, and risk management modules to the cloud so as to filter out the most relevant financial data for in-house analysis. Thereby, sensitive data and query results may be leaked to malicious adversaries and/or an honest-but-curious service provider itself. Such concerns have motivated research on data encryption and query answering over encrypted data, starting with SQL query processing over encrypted data [19], and expanding to the design of tailor-made schemes that allow the processing of specialized queries over encrypted data, such as kNN [43] and skyline queries [10]. CryptDB has provided a coherent collection of efficient SQL-aware encryption schemes that allows for the execution of SQL queries over encrypted data [36].

In all cases, the aim is to allow data to be managed by a server with minimal intervention by data owner and clients. The server should be able to process queries over the encrypted data, and deliver encrypted query results to the client, while the client should only have to decrypt the data and obtain the actual results thereby. Nevertheless, in order for a system to manage data and process queries efficiently, it should, first of all, be capable to *index* the data to the extent deemed necessary. In addition, modern applications, handling continuously arriving data, call for an inherent capacity to perform incremental indexing *on demand*, while processing user queries, and as a side-effect of such queries, requiring neither a priori idle time nor a priori workload knowledge; in other words, the system should not only be capable for indexing; it should be capable for *adaptive indexing* and *self-organization* [25, 20, 28].

These requirements for data management in the database-as-a-service era bring forth a contradiction. The requirement that data be kept confidential necessitates that it be *encrypted*; the requirement that data be efficiently managed necessitates that it be indexed. *Encryption* and *indexing* are in tension with each other; the former requires that the service provider knows nothing about the values of the data, while the latter necessitates that the same service provider knows, ideally, the exact values of the data under its purview. Given such conflicting aims, previous research has confronted these two challenges in isolation from each other and made no effort to face them in unison. Solutions for adaptive indexing and self-organization have not considered security and confidentiality, while secure database systems do not cater for adaptivity in dynamic environments. Unfortunately, by shying away from such an objective, such approaches offer fragmentary solutions to the problem of secure and efficient outsourced data management; in particular, extant encryption schemes suffer from one or more of the following drawbacks: (i) they are too computationally expensive; or (ii) leak too much information on the encrypted data, and/or (iii) require more data than the actual query results to be retrieved and then filtered in a post-processing step.

In this paper, we propose a model for breaking this isolation and achieving a seemingly contradictory target: allow for database systems that are both secure and adaptive. We examine the challenge of *adaptively indexing an encrypted database*. To bring about such a result, we investigate the extent to which encryption and indexing

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882932

are in tension with each other: we study the bare-bones requirements for indexing, and the ways in which such requirements may be reconciled with encryption. We propose a lightweight and efficient encryption scheme that reveals less information than previous suggestions and allows for indexing encrypted numeric data adaptively. This indexing capacity does not aim to develop a complete index upfront, but to one built in an incremental, progressive manner, so that only those data which are queried get indexed. Such an *adaptive indexing* enables lightweight adaptation of the system to the workload at hand and also mitigates the tension between encryption and indexing. Our scheme relies on simple linear-algebra operations for encryption and decryption, while it can efficiently process range and point queries over ciphertexts without disclosing the order among attribute values, as an Order-Preserving Encryption Scheme [3], and without the computational and storage burdens of schemes like fully homomorphic encryption [14].

2. BACKGROUND AND STATE OF THE ART

We review prior work on data systems that support queries over encrypted data, as well as on adaptive indexing.

2.1 Processing Encrypted Data

Research on processing encrypted data can be classified in two broad classes. The former class of solutions propose processing the encrypted data **directly**. Hacıgümüş et al. [19] proposed a bucketization scheme that allows for approximate filtering of query results at the server, followed by the final processing at the client, after decryption. Hore et al. [22] expand on this scheme with an index that supports obfuscated range queries, extended to the multidimensional case [21]. However, such schemes reveal the data distribution and involve the client in query processing, while the only indexing capacity they offer is that afforded by bucketization.

An attempt for fine-grained indexing would necessitate *sorting* values at the server. Such an alternative is offered by Agrawal et al.'s *Order-Preserving Encryption Scheme* (OPES) [3], built on an encoding that preserves the order of numerical data. However, as previous research has noted [44, 6, 7, 37], OPES reveals the data order, hence cannot overcome attacks based on statistical analysis on encrypted data. Arguably, OPES provides an *overkill* solution: it delivers encrypted values in sortable form, hence allows such values to be compared to each other. We argue that a more conservative alternative would enable *selective indexing* without a priori leaking information about the order of values.

In the meantime, advances in cryptography have led to the capacity to perform arbitrary computations over encrypted data, so as to obtain a correct encrypted result, by fully homomorphic encryption (FHE) [14]. Such computations rely on an expensive public key cryptosystem, bringing forth prohibitive overheads of up to nine orders of magnitude [15]. Besides, FHE does not enable *indexing* encrypted data. With FHE, even while the server performs all computations correctly, it does so *only in the encrypted view of the world*. It has no access to any truth in the real world that would allow for building an index. Mani et al. [33] discuss the potential of FHE to enable the vision of secure database as a service, and conclude that practicality remains a very important concern. To process a simple select query by the two-step process proposed in [33], the server computes the query results first and sends to the client the encrypted number n of result tuples; the client decrypts n, and asks for $n' \geq n$ rows from the server. The server returns the top-n' rows in the result; thus, the client needs to be actively involved in simple query processing tasks.

Wang and Lakshmanan [42] propose a scheme for securely evaluating queries over encrypted XML data. They advocate an extension of OPES, order-preserving encryption with splitting and scaling (OPESS), whereby each plaintext is "split" into more ciphertexts and the split data are "scaled", so that the attacker cannot determine the identity of ciphertexts based on data frequency knowledge. However, this scheme cannot support updates; thus, it is unsuitable for a dynamic and adaptive database system.

Shi et al. [38] propose an encryption scheme for answering multidimensional range queries. Yet the problem they address is that of allowing an auditor to decrypt *those and only those* records (e.g., financial audit logs, medical anamneses, etc.) whose attributes fall within a specified range; privacy is not protected when an entry is matched by the query. Unfortunately, this *match-revealing* scheme does not allow for protecting attribute values while enabling a service provider to identify records matching queries over encrypted data in a way that can be exploited for indexing.

Boneh and Waters [9] process complex queries over encrypted data based on Hidden Vector Encryption (HVE). The notion of security in [9] is stronger than the one in [38]: query-matching records are identified, but their attribute values remain hidden. This *match-concealing* security notion fits the requirements for indexing while protecting the privacy of attribute values. However, the scheme in [9] is not practicable for vast volumes of dynamic data; it necessitates $O(DT)$ public key size, encryption time, and ciphertext size, for D attributes and T discrete values per attribute.

Tu et al. developed MONOMI [40], a system that can execute analytical queries over encrypted data. Building on CryptDB [36], MONOMI uses several techniques so as to improve performance, along with a designer that chooses efficient server-side physical designs and a planner that selects efficient query execution plans involving server and client. However, insofar as MONOMI allows for order-based indexing, it does so by relying on OPES. More recently, Li et al. [31] proposed a scheme for processing range queries over encrypted data in which inequality checks are conducted via exhaustive equality checks; no mechanism for pure inequality checks over encrypted data is suggested.

Overall, these approaches do not provide a good tradeoff between confidentiality and efficiency; lately, some research efforts have provided tailor-made encryption schemes that allow the processing of specialized queries over encrypted data, such as kNN queries [43], range search queries [45], and skyline queries [10].

An alternative to processing encrypted data directly is to maintain an **encrypted index** on the server, and rely on the client for traversing this index and locate the data of interest after a few iterations of retrieval and decryption. Damiani et al. [12] build a B-tree over plaintext values, but encrypt every tuple and the B-tree itself at node level; the tree's content is not visible to the server. Ge and Zdonik [13] propose *Fast Comparison Encryption* (FCE). An index traversal with FCE necessitates comparisons between plaintext and ciphertext key values by *partial decryption* on the client side. Wang et al. [41] provide a secure query processing framework that protects data both *in storage* and *at access*, based on Salted IDA (Information Dispersal Algorithm). By this scheme, a client maintains a secret *information dispersal matrix* and the keys for decoding a data matrix D; the client encodes and disperses D onto n servers, while using a pseudo-random number generator with a secret seed to add a random *salt* to each data entry; when retrieving data, salts are reconstructed and deducted from the decoded data, so as to recover D. Thus, query processing is directed by the client, while servers can only access data following the client's instructions. Overall, approaches based on an encrypted index do not allow the server to build, maintain, and traverse an index relying on its own devices.

A third direction involves processing encrypted data in **secure hardware**, most recently exemplified by Cipherbase [5]. Yet, for

a range index Cipherbase reveals the full ordering information of index keys, even to a weak adversary; thus, range indexes in Cipherbase provide "similar confidentiality guarantees" [5], and have the same drawbacks as, order preserving encryption (OPE).

2.2 Adaptive Indexing

Apart from security requirements, data systems for modern applications need to be flexible and agile, easily adapting to rapidly changing requirements [11, 24]. *Adaptive indexing* is a recently introduced concept, by which index tuning need not be performed during system initialization [25, 26, 29, 20, 17, 18, 23, 34, 16, 35]; instead, it occurs during query processing: each query is interpreted not only as a request for a particular result set, but also as an advice on how to physically store the data, triggering small actions that refine the adaptive indexes. Those data portions that are queried become progressively and incrementally indexed. This capacity to index by continuously reacting to a changing query workload brings forth the property of *self-organization*.

In this section, we describe *database cracking* [23] in more detail. Database cracking introduced the notion of continuous, incremental, and on-demand adaptive indexing in the context of modern column-stores [1]. With cracking, the select operator of a database system performs index-building operations as a side-effect of processing a range (or equality) filtering action [25]; an index is built and refined collaterally to query execution: the more a data range is queried, the more its index is refined. The physical data store is changing with each incoming range query q, interpreting *q as a hint on how data should be stored*. That hint may explicitly use q's query bounds, or follow a more lax interpretation for the sake of robustness [20]. Without loss of generality, we discuss the strict interpretation. Assume a query requests $A<10$. A cracking DBMS responds to q by clustering all tuples of A with $A<10$ at the beginning of the respective column C, while pushing all other tuples to the column's end. A subsequent query requesting $A\geq v_1$, where $v_1\geq 10$, will only need to look into the last piece of C, while a query that requests $A<v_2$, where $v_2\leq 10$, searches its first part; each subsequent query cracks its respective piece further. Figure 1 shows how two queries crack a column by their selection predicates. Query Q1 cuts the column in three pieces, and Q2 enhances this partitioning by cutting the first and the last piece further. Each query collects its qualifying tuples in a contiguous area. Thereby, the original column A (including positions) is copied into a *cracker index* column, which is then continuously reorganized.

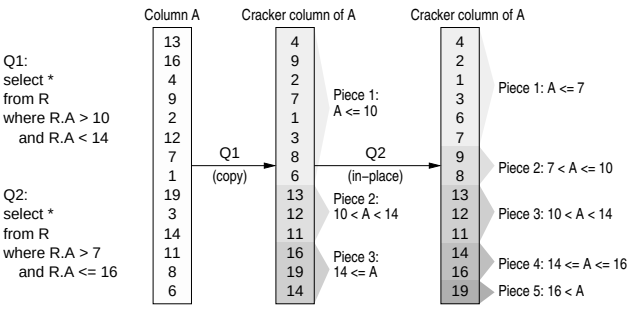


Figure 1: Cracking a column [25]

As queries are being processed, the adaptive index of a column is continuously split into more (and thus smaller) pieces. Therefore, we also need a data structure to localize a piece of interest. Past adaptive indexing literature has relied on an in-memory AVL-tree to keep track of all pieces created due to physical reorganization [25]. As a column becomes progressively indexed, queries requesting ranges that are exact matches of values known by the index are answered at the cost of searching only that tree-index; in all cases, even when there is no exact match, the index reduces the amount of data a query has to touch, as each range query reorganizes at most two pieces at the edge of the relevant range.

This on-the-fly physical reorganization is performed on individual pieces (or the whole column for the very first query) by in-place algorithms relying on sequential access patterns [25]. Algorithm 1 shows the most basic cracking algorithm that splits a piece into two, identifying tuples that need to be exchanged by *comparison* operations. Over the years, numerous algorithms have been proposed that split a piece of a column into three pieces [25], N pieces using radix clustering [29], partially sorting pieces in combination with partition/merge operations [29], fully sorting pieces when touched for the first time [**?**], and splitting pieces based on random pivots as opposed to query predicates [20, 34], yet the core idea remains the same. Figure 1 shows Query Q1, which partitions the whole column into three pieces, followed by Query Q2 that reorganizes Pieces 1 and 3 [25]. Other pieces of work accommodate updates [26], propagate the physical organization from one column to others on demand [27], and exploit multi-core CPUs [34, 35].

Algorithm 1 CrackInTwo(c,posL,posH,med,inc)
Physically reorganize the piece of column *c* between *posL* and *posH* such that all values lower than *med* are in a contiguous space. *inc* indicates whether *med* is inclusive or not, e.g., if *inc* = *false* then ϕ_1 is "<" and ϕ_2 is ">="

```
1: x1 = point at position posL
2: x2 = point at position posH
3: while position(x1) < position(x2 ) do
4:    if value(x1) φ1 med then
5:       x1 = point at next position
6:    else
7:       while value(x2) φ2 med &&
         position(x2) > position(x1) do
8:          x2 = point at previous position
9:       exchange(x1 ,x2 )
10:      x1 = point at next position
11:      x2 = point at previous position
```

In this work, without loss of generality, we consider the basic cracking design. We focus on the action of reorganizing a column into pieces with respect to a breakpoint as part of a query plan itself; thereby, a cracking select operator physically reorganizes the proper pieces of a column to bring all qualifying values in a contiguous area, which is leveraged to return the result, without additional result materialization [23]. Besides, with an AVL tree used for indexing, tree updates and rebalancing actions are also performed during query processing, in logarithmic time. This core operation is common and universally applicable to all adaptive indexing design. Our subsequent discussion shows how to apply this basic notion over encrypted data; we emphasize that our design extends the basic cracking design in a way that does not violate its assumptions about the underlying architecture: we assume that data is stored one column-at-a-time in *fixed-width* dense arrays as in modern column-stores [32, 39, 8], both on disk and in memory. Query processing may happen in bulk or vector-wise processing. These basic properties apply to all column-stores.

Crucially, adaptive indexing never necessitates fully sorting individual pieces [23, 29]; when a piece becomes small enough to fit in L1 cache, we scan the data at virtually no overhead. Thus, queries only cause reorganization for data pieces larger than a size threshold; that threshold can be bigger (e.g., L3 cache size) without a significant performance drop. In the context of our work, it follows that we *never* leak the *total data order* by a fully sorted index, as OPES does by default.

3. PROPOSED ENCRYPTION SCHEME

We aim to devise an encryption scheme that provides protection against attacks that aim to compromise data confidentiality, yet enables self-organizing indexing operations over encrypted numeric data, i.e. an *indexable encryption scheme*. However, unlike OPES [3], we do not wish our encryption to preserve the order of numerical data; we aim to enable efficient querying without leaking information about the order of tuples. An adversary can be either an external malicious entity or an honest-but-curious server. Insider attacks, such as those arising from malicious partners, are not considered. Client machines are assumed to be safe; confidential information such as secret keys on the client and client queries are not known to attackers. Attacker's computations are bounded by polynomial-size circuits. We first outline the requirements we would like our encryption scheme to satisfy.

1. It should provide the server with *some* capacity to **conduct inequality comparisons** between data values (plaintexts) employing their encrypted forms (ciphertexts).
2. This capacity to perform comparisons should **not directly reveal the order** among encrypted numeric data.
3. Comparison operations should not necessitate decryption at the server; attribute values should always **remain hidden** to the server.
4. The key sizes, encryption time, and ciphertext sizes should be **manageable** and not growing with the size of data, number of attributes, or number of discrete values.
5. The client should not be elaborately involved in processing simple queries; the server should deliver the encrypted results, and only those, to the client in a **single round** of communication.
6. The server should be capable to gracefully accommodate newly arriving data values and support **updates** in the encrypted data.

A prime challenge is to resolve the tension between requirements (1) and (2), i.e., enable inequality comparisons *without* revealing order. A way to achieve this result is to postulate that comparisons should *not* be possible *among* encrypted data residing at the server. If such data are not comparable to each other, then they cannot be straightforwardly brought to sorted order. We must then spell out under what circumstances a server should be able to perform an inequality check among ciphertexts. Such comparisons should be possible only *on demand*, between ciphertexts (e.g., a query bound and a tuple value) rendered comparable by their encryption; in effect, indexing actions would also be possible *only on client demand*. Such *on-demand indexing*, would be compatible with adaptive indexing: If we have to perform on-demand indexing for the sake of security, we also need to perform exactly such actions for the sake of adaptiveness, as the data has to be indexed in response to queries. Then, instead of being in tension with each other, the requirements for security and adaptivity reinforce each other.

We must then devise a method of creating *comparable ciphertexts*. Such a method could utilize two complementary encryption modes, A and B, *interfacing* with each other, so that a ciphertext encrypted in mode A can be compared to one encrypted in mode B. Then, encrypting attribute values in mode A and query bounds in mode B, we would offer the server a capacity to compare the former to the latter; as we saw, such comparisons form the backbone of adaptive indexing operations, and are performed with operators ϕ_1 and ϕ_2 in Algorithm 1. Assume a server needs to perform an inequality check between key value v in a column C and query bound value b, i.e., to check whether $v \geq b$. Equivalently, we need to check whether:

$$v - b \geq 0 \tag{1}$$

Using vector notation, we express Inequality (1) as

$$\begin{pmatrix} 1 \\ b \end{pmatrix} \cdot \begin{pmatrix} v \\ -1 \end{pmatrix} = v - b \geq 0 \tag{2}$$

Our objective is to devise an encryption scheme that allows the computation of such scalar vector products in obscured fashion. We employ three obscurement operations, to be be performed by the data vendor when uploading the data to the server and by a trusted client before issuing a query: (i) noise addition; (ii) scalar multiplication, and (iii) matrix multiplication. We elaborate on these in the following.

3.1 Noise Addition

Our noise addition operation builds two longer vectors $\mathbf{b}$, $\mathbf{v}$, by adding extra noisy elements to the length-2 vectors used in Inequality (2), such that the result of the vector product remains the same. We call $\mathbf{b}$ and $\mathbf{v}$ the encrypted *bound* and *value* vector, respectively. The specific length ℓ of vectors $\mathbf{b}$ and $\mathbf{v}$, and the positioning and distinction between the added *noisy contents* and original *value contents* therein constitute part of our *encryption key*, known to the data owner and trusted client, but not to the service provider. Without loss of generality, we present an example using vectors of length 5. We can rewrite Inequality (2) as follows:

$$\mathbf{b} \cdot \mathbf{v} = \begin{pmatrix} -4 \\ 1 \\ 8 \\ b \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ v \\ -2 \\ -1 \\ 7 \end{pmatrix} = v - b \geq 0 \tag{3}$$

In this example, the original contents of the vectors occupy positions 1 and 3 in them, whereas positions $\{0, 2, 4\}$ are reserved for noisy contents that cancel each other out in the scalar vector product, as they form two orthogonal length-3 embedded subvectors, $\mathbf{n}_b$ and $\mathbf{n}_v$, with inner product 0:

$$\mathbf{n}_b \cdot \mathbf{n}_v = \begin{pmatrix} -4 \\ 8 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ -2 \\ 7 \end{pmatrix} = 0 \tag{4}$$

We call these orthogonal subvectors *noisy subvectors* of $\mathbf{b}$ and $\mathbf{v}$. We emphasize that we need not use the same exact noisy subvectors for each encrypted key value v or each encrypted bound value b, yet we do have to use the same *positions* for noisy contents across all encrypted values in the same column. We could use any pair of vectors orthogonal to each other, such that their inner product produces 0. Nevertheless, a vector $\mathbf{b}$ produced for a given bound value b should be orthogonal to *all* vectors $\mathbf{v}$ produced for attribute values. Therefore, we cannot choose our orthogonal vector pairs in an entirely arbitrary fashion. However, it suffices to select a unit vector $\mathbf{u}$, and make sure that each bound value b is obscured by embedding into the column vector $\binom{1}{b}$ a randomly selected noise vector $\mathbf{n}_b$ collinear to $\mathbf{u}$, $\mathbf{n}_b = \lambda(b) \cdot \mathbf{u}$, with the values 1 and b occupying specific preselected positions, as shown above, to produce $\mathbf{b}$. Then, any data value v is obscured by similarly embedding a randomly selected vector $\mathbf{n}_v = \mathbf{u}^{\perp}(v)$, orthogonal to $\mathbf{u}$, into the column vector $\binom{v}{-1}$, to produce $\mathbf{v}$. We emphasize that the vectors $\mathbf{u}^{\perp}(v)$ do *not* need to be collinear to each other. Any vector

orthogonal to $\mathbf{u}$ will suffice. Summing up, given a unit vector $\mathbf{u}$, the noisy subvectors embedded in $\mathbf{b}$ should be collinear to $\mathbf{u}$, while those embedded in $\mathbf{v}$ should be orthogonal to $\mathbf{u}$. Then an inequality check can be performed by means of a scalar vector product, as $v - b$ is calculated as $\mathbf{b} \cdot \mathbf{v}$.

In the above example, the unit vector used is $\mathbf{u} = \begin{pmatrix} \frac{-1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{pmatrix}$. Notably, $\|\mathbf{u}\| = 1$. A noisy subvector of $\mathbf{b}$ is produced by multiplying $\mathbf{u}$ by a randomly selected factor $\lambda(b)$. In above case, $\lambda(b) = 4\sqrt{6}$. On the other hand, a noisy subvector of $\mathbf{v}$ is produced by randomly selecting *any* vector of length $\ell - 2$, $\mathbf{u}^{\perp}(v)$, orthogonal to $\mathbf{u}$.

3.2 Scalar Multiplication

So far we have devised an encryption scheme that enables the calculation of the difference $v - b$ by the server, via the calculation of a scalar product of two vectors. Thus, we attain the objective of performing inequality checks, hence processing range queries, over the encrypted numeric data, while our encryption scheme does *not* preserve the order of the data in the way that OPES does. However, given an encrypted bound vector $\mathbf{b}$, an adversary can calculate any scalar product of the form $\mathbf{b} \cdot \mathbf{v}$, hence the difference $v - b$, for any encrypted value vector $\mathbf{v}$; then, by comparing the obtained $v - b$ values, one obtains not only the order, but also the exact differences among the encrypted data values.

To prevent this kind of leakage, we should not enable the calculation of exact differences of the form $v - b$. After all, we are not interested in obtaining the correct norms of those differences. Getting merely their *sign* would suffice. The norm of $|v - b|$ may be altered, as long as the sign is correctly obtained. We achieve this effect by adding a *scalar multiplication* in our obscurement operation. Then, for each data value v, the data vendor chooses a random positive multiplier $\xi(v) > 0$, and recalculates the encrypted value vector $\mathbf{v}$ as $\mathbf{v}' = \xi(v)\mathbf{v}$. Henceforward, we use $\mathbf{v}$ to refer to this recalculated value vector. In effect, the scalar product we calculate becomes $\mathbf{b} \cdot \mathbf{v} = \xi(v)(v - b)$. The server still obtains the correct *sign* of the difference $v-b$, *without* the exact norm of this difference being leaked. In effect, an adversary cannot reconstruct the order of data values using information obtained from scalar products.

3.3 Matrix Multiplication

Our noise addition and scalar multiplication operations produce a vector $\mathbf{v}$ representing each key value v and a vector $\mathbf{b}$ representing a query bound value b, so that *the sign*, but not the norm, of $v - b$ can be obtained from the scalar product $\mathbf{b} \cdot \mathbf{v}$. Still, the security these operations provide cannot withstand a simple breach: an adversary who learns the position within a vector $\mathbf{v}$ where the values $\xi(v)v$ and $-\xi(v)$ reside, can effectively acquire all original key values. We now provide an additional encryption layer that obstructs this breach.

Assume $\mathbf{v}$ and $\mathbf{b}$ are vectors of length ℓ, and let M be any invertible $\ell \times \ell$ matrix, and M^{-1} its inverse. M constitutes part of our encryption key; it is known to the data vendor and its trusted clients, but not to the service provider. We can then encrypt any key value v and any breakpoint b as follows.

$$\mathbf{E}_v(v) = M^{-1}\mathbf{v} \tag{5}$$

$$\mathbf{E}_b(b) = M^T\mathbf{b} \tag{6}$$

where $\mathbf{E}_v(\cdot)$, $\mathbf{E}_b(\cdot)$ are our encryption functions, $\mathbf{b}$ and $\mathbf{v}$ are represented as column vectors, and M^T is the transpose of matrix M. Then:

$$\begin{aligned} \mathbf{E}_b(b) \cdot \mathbf{E}_v(v) &= \mathbf{E}_b(b)^T\mathbf{E}_v(v) \\ &= (M^T\mathbf{b})^T(M^{-1}\mathbf{v}) \\ &= (\mathbf{b}^T M)(M^{-1}\mathbf{v}) \\ &= \mathbf{b}^T\mathbf{v} = \mathbf{b} \cdot \mathbf{v} = \xi(v)(v - b) \end{aligned}$$

where row vector $\mathbf{a}^T$ is the transpose of column vector $\mathbf{a}$.

In effect, any inequality check can be effectively conducted using the encrypted vectors $\mathbf{E}_v(v)$ and $\mathbf{E}_b(b)$. $\mathbf{E}_v(v)$ is generated by the data vendor and passed on to the server when the data (or an update) is generated. $\mathbf{E}_b(b)$ is produced by the trusted client, or again by the data vendor on behalf of the client, when issuing a query. These two encryption (and reverse decryption) steps are the only workload data vendor and client have to bear. Other computations and query processing are conducted by the server as with a non-encrypted database. At the same time, given those encrypted vectors, an adversary cannot determine the values of v or b without knowing the invertible matrix M.

3.4 The Encryption Key

In a nutshell, our total encryption key consists of:

1. The unit vector $\mathbf{u}$.
2. The arbitrary orientation of each vector $\mathbf{u}^{\perp}(v)$, orthogonal to $\mathbf{u}$, which is individually selected to construct $\mathbf{v}$ for a data value v.
3. Each factor $\lambda(b)$ used to produce a noisy vector collinear to $\mathbf{u}$, in order to construct $\mathbf{b}$ for a query bound b.
4. Each random positive multiplier $\xi(v)$ used to obscure the norm of $v - b$.
5. The positions occupied by the contents of $\begin{pmatrix} \xi(v)v \\ -\xi(v) \end{pmatrix}$ and $\begin{pmatrix} 1 \\ b \end{pmatrix}$ in $\mathbf{v}$ and $\mathbf{b}$, respectively.
6. The invertible matrix M.

3.5 Resistance to Attacks

We provide a sketch of the capacity of this lightweight encryption scheme to withstand attacks, assuming an honest but curious adversary, aware of the internal workings of our scheme, who aims to learn our key from known ciphertexts and potentially known plaintexts. As our scheme consists of three layers, we separate our discussion in two parts.

Noise and Scalar Multiplication Layers Leaving the matrix multiplication part aside, assume an adversary, Alice, who directly observes noisy vectors before they are multiplied by M. In such a known ciphertext attack, Alice would only be challenged to sort out the noisy contents from the value contents of those vectors. She can use the information that the noisy elements of an encrypted value vector $\mathbf{v}$ and an encrypted breakpoint vector $\mathbf{b}$ are orthogonal to each other, and make a hypothesis about the positions of the noisy contents vs. value contents in the observed vectors, i.e. arbitrarily select 2 positions out of ℓ in those vectors where the assumed value contents reside. To test this hypothesis, she can examine whether the assumed noisy subvectors produce consistently inner product 0 across many different observed $\{\mathbf{b}, \mathbf{v}\}$ pairs. If that is the case, then she can safely conclude that her hypothesis is verified. The question is how much Alice would have to try in order to reach a correct hypothesis by exhaustive search. Her hypothesis amounts to selecting 2 out of ℓ vector elements. As there are

$\binom{\ell}{2} = \frac{\ell(\ell-1)}{2} = O(\ell^2)$ ways of making such a selection, Alice can arrive at the correct hypothesis in polynomial time. We conclude that the noise layer of our scheme is easy to break. This is why we added the matrix multiplication layer, which we study next.

Matrix Multiplication Layer Once the noisy vectors Alice observes get multiplied by M, she cannot carry out the above attack. Now the multiplication matrix M enters the picture, consisting of ℓ^2 unknowns. The unit vector $\mathbf{u}$ brings $\ell - 2$ more unknowns into the picture; each encrypted vector $\mathbf{v}$ brings $\ell - 3$ more unknowns, as only one out of its $\ell - 2$ noisy elements can be derived using the $\ell - 3$ others and orthogonality to $\mathbf{u}$, and the $\xi(v)$ factor, which has multiplied $\{v, -1\}$ to produce its noisy contents; each encrypted vector $\mathbf{b}$ brings a $\lambda(b)$ factor, which multiplies $\mathbf{u}$ to produce its noisy contents. An attack that could still bear fruit is a known plaintext attack, in case Alice could gain access to pairs of an original value (either v or b) and its encrypted form ($\mathbf{E}_v(v)$ or $\mathbf{E}_b(b)$, respectively). For each such pair, she can construct ℓ scalar linear equations. Eventually, it can be shown that it suffices for Alice to know $N \geq \frac{\ell^2+\ell-2}{\ell-1} + 1 = O(\ell)$ plaintext-ciphertext pairs of values b; she can then solve the resulting system of equations for each of $\frac{\ell(\ell-1)}{2}$ ways of positioning noisy contents in $\mathbf{v}$ and $\mathbf{b}$, and, again, identify the one that leads to a solution consistently resulting in orthogonal noisy vector products. We conclude that the security of our encryption scheme against known plaintext attacks strongly depends on the chosen ciphertext size ℓ, while our scheme provides the data owner with the flexibility to choose the value of ℓ at will.

4. INDEXING ENCRYPTED DATA

We now discuss how our server can perform adaptive indexing over data encrypted by our encryption scheme. Notably, the choice of ℓ in our scheme determines the dimensions of matrix M, and incurs a ℓ-fold increase in our storage requirements. We contend that this storage overhead is an affordable price to pay for the sake for performing secure adaptive indexing operations.

4.1 The Problem of Leakage by Structure

We have so far emphasized that we eschew an order-preserving encryption scheme, so that the *order* of values does not leak to adversaries. Nevertheless, adaptive indexing operations do progressively bring the underlying data column in *sorted order*. After all, database cracking can be validly described as an *incremental quicksort*, while another alternative for adaptive indexing [**?**], adaptive merging, can be seen as an *incremental external merge sort*. An adaptive indexing mechanism, left to its own devices, will *tend* to bring the data in sorted order. Given a workload $\mathcal{W}$ such that, for any pair of values in the column v_1, v_2, there exists at least one query bound $b \in \{v_1, v_2\}$, adaptive indexing with $\mathcal{W}$ will eventually bring the data in an exact sorted order.

In effect, even though the encryption does not intrinsically betray the order of values, that order can eventually be observed in the order records are brought to after enough cracking operations have been applied. An adversary who can identify a target tuple can infer its position in the sorted order by observing the structure of the constructed index tree. The more refined the tree becomes, the more information it can leak about the order of underlying tuples.

4.2 Deliberately Allowing Errors

To mitigate the problem of leaking order by structure, we allow for erroneous interpretations of outsourced data, while the server cannot distinguish which interpretation is valid. Thus, even an adversary (or a server) knowing the internal workings of our scheme will not be able to make confident inferences about the relative positions of tuples.

We define *two* possible ways of interpreting each encrypted data value, without leaking which one of the two is correct for each tuple t. The server merely takes actions on both, by maintaining both possible interpretations of each record whenever needed. Only one of those versions corresponds to the true record, yet only the data owner and its trusted clients can make this distinction. From the point of view of the server, each interpretation is equally likely to be valid. In other words, the sever is concurrently managing multiple *possible worlds*. We configure such *multiple realities* by making each $\mathbf{E}_v(v)$ vector longer by a value θ, arbitrarily adding θ as a prefix *or* as a suffix to it, to obtain $\bar{\mathbf{E}}_v(v)$; we select θ so that both the ℓ-prefix and ℓ-suffix of $\bar{\mathbf{E}}_v(v)$ work consistently as valid $\mathbf{E}_v(v)$ vectors interfacing with $\mathbf{E}_b(b)$ vectors: in both, after multiplying by matrix M, the contents obtained in the positions reserved for noisy contents in a $\mathbf{v}$ vector are orthogonal to $\mathbf{u}$.

To facilitate the following discussion, we introduce a number of matrices useful so as to formally represent the creation of vectors and noise addition via linear-algebra operations. These matrices, and the purposes they serve, are gathered together in Table 1.

Matrix	Purpose
E_{nm}	*expansion matrix*: extends vector with $n - m$ zeros
P_{nm}	*permutation matrix*: shuffles extended value/bound vectors
P^c_{nm}	*complementary permutation matrix*: shuffles noise vector
S	*shift matrix*: cyclically shifts vector elements down
S^T	*symmetric shift matrix*: cyclically shifts vector elements up

Table 1: Employed matrices

We calculate the value of θ as follows. We are starting out from $(v; -1)^T$, the noisy subvector $\mathbf{n}_v \propto \mathbf{u}^\perp$, and $(1; b)^T$, the noisy subvector $\mathbf{n}_b = \lambda(b) \cdot \mathbf{u}$, where $\mathbf{u}$ is the secret unit vector and $\mathbf{u}^\perp$ a vector orthogonal to $\mathbf{u}$. Then, for the purposes of our encryption scheme, we first need to formally describe the *expanding* of these vectors into vectors of size ℓ. To that end, we employ two $n \times m$ *expansion matrices* that we denote as E_{nm}; such matrices multiply vectors from left and thereby extend those vectors with $n - m$ zeros. The structure of an E_{nm} matrix can be represented as $\begin{pmatrix} I \\ 0 \end{pmatrix}$, where I is the identity matrix. Then it holds that:

$$E_{\ell 2} \begin{pmatrix} v \\ -1 \end{pmatrix} = \begin{pmatrix} v \\ -1 \\ 0 \\ ... \\ 0 \end{pmatrix}, E_{\ell(\ell-2)}\mathbf{n}_v = \begin{pmatrix} \mathbf{n}_v \\ ... \\ 0 \end{pmatrix}$$

As a next step, we need to *shuffle* the contents of those expanded vectors so as to bring their contents in the positions predicted by our scheme. To that end, we employ two $n \times n$ *permutation matrices* that we denote as P_{nm} and P^c_{nm}; these matrices shuffle the extended value/bound and noise vectors, respectively, according to our noise addition scheme; only their first m rows have nonzero contents, hence it holds that $P_{nm}(P_{nm})^T = \begin{pmatrix} I_m & 0 \\ 0 & 0 \end{pmatrix}$; P^c_{nm} is chosen so that $P_{nm}(P^c_{nm})^T = 0$, i.e. P_{nm} and P^c_{nm} do not have permutation intersections: the one shuffles a vector's elements into positions complementary to those of the other. Last, let $\xi(v)$ and $\lambda(b)$ be the "scaling" functions we employ in our encryption scheme, as defined in Section 3.4. Thus, scaled noisy value and bound vectors can be represented as:

$$\mathbf{v} = \xi(v) \left(P_{\ell 2} E_{\ell 2} \begin{pmatrix} v \\ -1 \end{pmatrix} + P^c_{\ell(\ell-2)} E_{\ell(\ell-2)} \mathbf{n}_v \right)$$

$$\mathbf{b} = P_{\ell 2} E_{\ell 2} \begin{pmatrix} 1 \\ b \end{pmatrix} + P^c_{\ell(\ell-2)} E_{\ell(\ell-2)} \lambda(b) \mathbf{u}$$

We denote the final encrypted noisy value vector as $\mathbf{E}_v = M^{-1}\mathbf{v}$, where M is the encryption matrix. Our error-allowance scheme creates one of the following two vectors: $\begin{pmatrix} \mathbf{E}_v \\ \theta \end{pmatrix}$ or $\begin{pmatrix} \theta \\ \mathbf{E}_v \end{pmatrix}$. Without loss of generality, we consider only the first variant. In this first variant, we need to represent the *fake encrypted noisy value vector* $\mathbf{E}^f_v$, i.e., the ℓ-suffix of $\bar{\mathbf{E}}_v(v)$, which can be considered as true by an adversary. To this end, we employ an $\ell \times \ell$ *shift matrix* that we denote as S; multiplication by this matrix cyclically shifts vector components down. For example, for $n = 3$: $S = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. Symmetrically, its transpose, S^T, shifts vector components up. Thereby:

$$\mathbf{E}^f_v = S^T \mathbf{E}_v + (\theta - \mathbf{E}_v \cdot \mathbf{e}_1)\mathbf{e}_\ell$$

where $\mathbf{e}_k$ is a unit vector with all components zero except the k^{th}; the above equation simply retains the $(\ell-1)$-prefix of $\mathbf{E}_v$, shifted one position upwards, while it places θ at the ℓ-th position.

We want this $\mathbf{E}^f_v$ to have all properties of a real encrypted value vector. Then, the "noisy" $l-2 \times 1$ subvector of the respective fake value vector $\mathbf{v}^f = M\mathbf{E}^f_v$ must be orthogonal to $\mathbf{n}_b$:

$$\mathbf{n}^f_v \cdot \mathbf{n}_b = E^T_{\ell(\ell-2)} (P^c_{\ell(\ell-2)})^T \mathbf{v}^f \cdot \mathbf{n}_b = 0 \Leftrightarrow$$
$$E^T_{\ell(\ell-2)} (P^c_{\ell(\ell-2)})^T M \left[S^T \mathbf{E}_v + (\theta - \mathbf{E}_v \cdot \mathbf{e}_1)\mathbf{e}_\ell \right] \cdot \mathbf{u} = 0 \Leftrightarrow$$
$$\left[\mathbf{E}^T_v S + (\theta - \mathbf{E}_v \cdot \mathbf{e}_1)\mathbf{e}^T_\ell \right] M^T P^c_{\ell(\ell-2)} E_{\ell(\ell-2)} \cdot \mathbf{u} = 0 \Leftrightarrow$$
$$\theta = \mathbf{E}_v \cdot \mathbf{e}_1 - \frac{\mathbf{E}^T_v SW \cdot \mathbf{u}}{\mathbf{e}^T_\ell W \cdot \mathbf{u}}$$

where $W = M^T P^c_{\ell(\ell-2)} E_{\ell(\ell-2)}$.

Figure 2 shows the form of an encrypted vector $\mathbf{E}_v$, while Figure 3 shows the encrypted vector $\bar{\mathbf{E}}_v(v)$, with ambiguity added. The size of the ambiguity-added vector is one more than original encrypted vector, hence each original value acquires two representations in the database: the one represented by the real encrypted vector, as the ℓ-prefix in the figure, and another represented by the fake vector, as the ℓ-suffix in the figure. Both are derived from the ambiguity-added vector as shown in Figure 3.

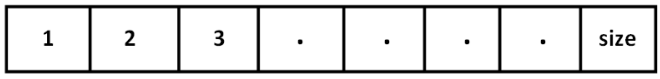


Figure 2: Encrypted Vector

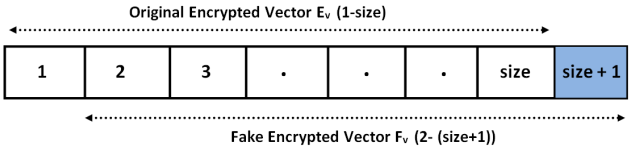


Figure 3: Encrypted Vector with Ambiguity

Having added this ambiguity about the identity of the real encrypted vector, we should render that identity identifiable by the trusted client. This can be achieved by selecting the (previously declared random) multiplier $\xi(v)$ in a particular way, e.g., postulating that it be an odd integer. The data owner and trusted client can then determine the real $\mathbf{E}_v(v)$ given $\bar{\mathbf{E}}_v(v)$ as follows: they decrypt both the ℓ-prefix and ℓ-suffix of $\bar{\mathbf{E}}_v(v)$, multiplying each by matrix M. The one that delivers an odd integer in the position of $\xi(v)$ is the real one. The fact that only one decryption attempt delivers an odd integer at that position is verified by the data owner during encryption.

However, what the client can determine so easily is inaccessible to the server. Under this arrangement, the server generates and manages each possible interpretation of $\bar{\mathbf{E}}_v(v)$ separately; this redundancy provides an obscurity that reduces the confidence of an adversary's inferences about the value order. Our end result is similar to adding counterfeit records in our database. However, the security provided by our scheme is higher than that provided by mere counterfeit insertion. In the case of adding counterfeits, an adversary with sufficient background knowledge may identify a *single* record of interest and infer information-leaking observations about its position in the index. On the other hand, with our scheme, the position of a record of interest in the index is uncertain *even when* that record of interest is identified, since each single record spawns two possible interpretations; this state of affairs confers additional security to our construction.

4.3 Managing an Encrypted AVL Tree

Our encryption scheme enables a server to manage a column of encrypted values, even with ambiguity inserted, and conduct comparisons between query bounds and data values. Yet, our system should also construct an AVL tree; in particular, query bounds, which correspond to the breakpoints b, are used as key values in tree nodes and utilized in subsequent tree traversals. During such a traversal, a new bound value b' has to be compared to a previous one, b, now used as key; then, at each node we need to compare two breakpoints values, b' and b, to each other.

In order to adapt this tree traversal mechanism to our encrypted data, the server needs to be able to compare two breakpoint values, b' and b, to each other. However, our encryption scheme uses two different encryption modes, one for breakpoints b, and one for values v, constructed so as to allow us to compare b to v only, but not different b and v values to each other. We solve this problem by having the query-issuing client encrypt a breakpoint b in both ways, i.e., in its native way, as $\mathbf{E}_b(b)$, and as an attribute value, $\mathbf{E}_v(b)$, and pass them on to the server. Subsequently, the former value is used for inequality checks with attribute values, while the latter is employed for storing it as a key in the constructed AVL tree index. When searching the encrypted AVL tree for a new breakpoint value b', the required comparison is conducted by calculating $\mathbf{E}_b(b') \cdot \mathbf{E}_v(b) = \xi(b) \cdot (b - b')$.

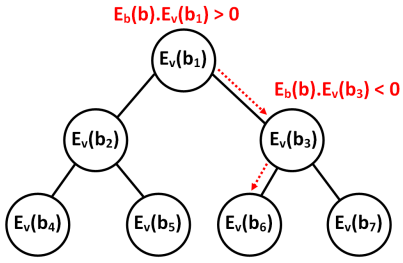


Figure 4: Finding a piece in an encrypted AVL Tree

Consider the example encrypted AVL tree in Figure 4. Finding a piece in this tree utilizes a query bound b encrypted as $\mathbf{E}_b(b)$, and key values in the tree itself encrypted as $\mathbf{E}_v(b)$; thus, the query bound and key values interface with each other and allow for conducting comparisons, resulting in the traversal shown in the figure.

Algorithm findpiece(objAVLTree,N,E_b,$posL$,$posH$)
Traverses AVL Tree and returns upper bound ($posH$) and lower bound ($posL$) values of the piece in which query bound vector $\mathbf{E}_b$ is located.

```
1:  posL = 0
2:  posH = N
3:  isFound = true
4:  rootNode = objAVLTree.getRoot()
5:  if rootNode is not empty then
6:     minNode = objAVLTree.findMinNode(root)
7:     maxNode = objAVLTree.findMaxNode(root)
8:     fNode = objAVLTree.findNode(Eb)
9:     if ScalarProduct(Eb,maxNode.key) > 0 then
10:       isFound = false
      **Case 1**
11:    if isFound == false && fNode is not minNode then
12:       posL = maxNode.position
      **Case 2**
13:    else if fNode == minNode then
14:       if ScalarProduct(Eb,fNode.key) < 0 then
15:          posH = fNode.position
16:       else
17:          posL = fNode.position
18:          fNode = objAVLTree.findSuccessor(fNode)
19:          if ScalarProduct(fNode.key,maxNode.key) < 0 then
20:             posH = fNode.position
      **Case 3**
21:    else if ScalarProduct(Eb,fNode.key) < 0 then
22:       posH = fNode.position
23:       posL = objAVLTree.findPredecessor(fNode).position
      **Case 4**
24:    else
25:       posL = fNode.position
26:       fNode = objAVLTree.findSuccessor(fNode)
27:       if ScalarProduct(Eb,fNode.key) < 0 then
28:          posH = fNode.position
```

Algorithm `findpiece` illustrates the finding of the crack positions for a query bound b; it makes use of largest and smallest values in the AVL tree in order to determine whether the query bound is out of the tree's range; otherwise it obtains the leaf node $fNode$ by a search operation on the tree based on the given query bound $\mathbf{E}_b$; overall, it distinguishes the following cases:

- *Case 1* holds in case b is greater than the largest value in the tree; then the largest value in the tree is returned as the lower bound of the piece range.
- *Case 2* holds when the returned fNode is equal to the smallest value in the tree; then, if the query bound b is greater than that value, the range from that value to its successor is returned; if the query bound is smaller, then the smallest value is returned as the upper bound of the piece range.
- *Case 3* holds in case b is less than fNode; then the range between fNode and its predecessor is returned.
- *Case 4* holds when b is greater than fNode; then a range between fNode and its successor, if such exists, is returned.

In all cases, comparisons are performed via scalar vector products according to our scheme.

Having located a piece where a query bound b belongs, we need to complete the cracking operation by adding the bound b, represented by both $\mathbf{E}_v(b)$ and $\mathbf{E}_b(b)$, at a leaf node position in the AVL tree itself (and possibly rebalance the tree), so as to facilitate future searches. Figure 5 presents an AVL-tree traversal along with the addition of a leaf node for b_8 at the last step, after the piece in which b_8 belongs has been cracked in two. Algorithm `addCrack` illustrates how this operation is performed; its first three cases examine the situation where a node for b already exists in the tree; the fourth case adds a new node.

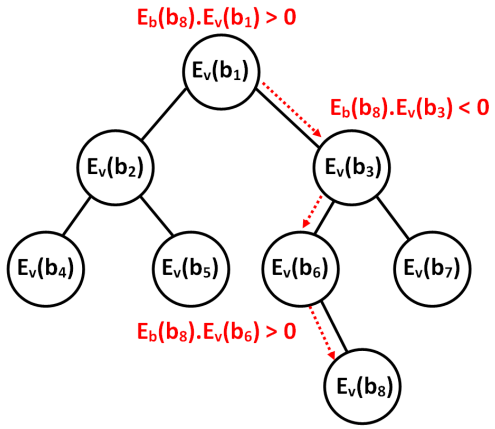


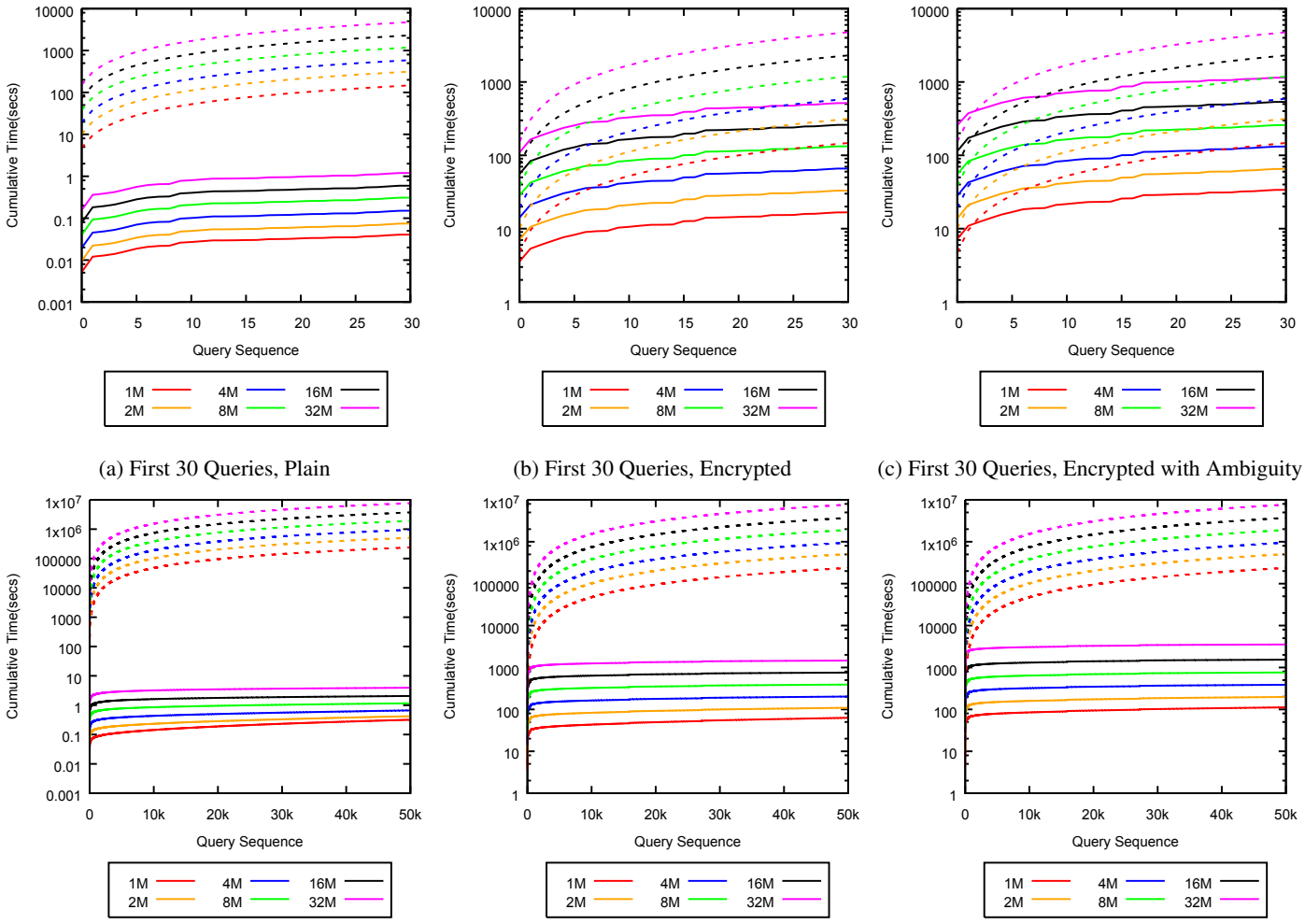
Figure 5: Adding a node in an encrypted AVL tree

Algorithm addCrack(objAVLTree,N,E_v,E_b,pos)
Adds a new node, for query bound b, to the AVL tree indexing N values, corresponding to position pos in the data array.

```
1:  if pos == 0 or pos ≥ N then return
2:  rootNode = objAVLTree.getRoot()
3:  isFound = true
5:  if rootNode is not empty then
6:     minNode = objAVLTree.findMinNode(root)
7:     maxNode = objAVLTree.findMaxNode(root)
8:     fNode = objAVLTree.findNode(Eb)
9:     if ScalarProduct(Eb,maxNode.key) > 0 then
10:       isFound = false
      **Case1**
11:    if isFound == true then
12:       if fNode.position == pos then return
12:       tmp = fNode
13:       if ScalarProduct(tmp.key,Eb) == 0 then
14:          tmp = objAVLTree.findSuccessor(tmp)
15:       if ScalarProduct(tmp.key,Eb) > 0 then
16:          if tmp.position == pos then return
      **Case2**
17:    if fNode is not minNode then
18:       if isFound == false then fNode = maxNode
19:       else fNode = objAVLTree.findPredecessor(fNode)
20:       if fNode.position == pos then return
      **Case3**
21:    if isFound == true && ScalarProduct(fNode.key,Eb) == 0 then
22:       fNode.setPosition(pos)
23:       return
      **Case4**
24:    objAVLTree.insert(Ev,pos,Eb)
25:    return
```

5. EXPERIMENTAL RESULTS

In this section we present an experimental analysis, demonstrating that secure adaptive indexing applying our schemes maintains all the adaptive properties and performance benefits of adaptive indexing while working over encrypted data. Our implementation is based on a stand-alone prototype in C++ that precisely implements the select operator of a modern column-store; it receives a column of values (fixed-width dense array) as input and returns a set of positions that mark qualifying values, represented again in a fixed-width dense array, as output. We keep this interface, common to all modern column-stores and all adaptive indexing approaches, intact. Our design affects only data manipulations that avoid data leakage. As our encryption scheme is based on precision-sensitive matrix operations, such as matrix inversion, matrix-vector multiplication, and vector-vector multiplication, we require high arithmetic precision; we employ the GNU Multi Precision Arithmetic



(a) First 30 Queries, Plain

(b) First 30 Queries, Encrypted

(c) First 30 Queries, Encrypted with Ambiguity

(d) 50k Queries, Plain

(e) 50k Queries, Encrypted

(f) 50k Queries, Encrypted with Ambiguity

Figure 6: Total Cumulative Time for different data sizes and types

Library (GMP)[1], a free library for arbitrary precision arithmetic, which operates on signed integers, rational numbers, and floating-point numbers, providing fast arithmetic operations for applications that need higher precision than is directly supported by the basic C types; GMP uses full $words$ as the basic arithmetic type.

Our infrastructure consists of a 3.5 GHz Intel-Xeon machine with 12 CPU(s) and 32 GB of memory. In our core experiments, we run a query sequence that incrementally reorganizes a single column, and observe performance as the sequence evolves. In order to measure the effect of encryption, we perform such experiments on three types of data: (a) plain data without encryption, (b) data encrypted by the scheme introduced in Section 3, and (c) data encrypted along with the provision introducing added ambiguity, as described in Section 4.2. Moreover, we compare our cracking-based results against a plain scan of the encrypted numeric data, evaluating queries using comparisons via scalar products without any indexing or cracking; we call this approach **SecureScan**.

In all experiments, we encrypt data with default key size $\ell = 4$. The data are unique integers, drawn uniformly at random from $[0, 2^{31})$. The default workload is a sequence of 50K random selection queries with selectivity 1%; such workloads have been shown to be representatively challenging in terms of index adaptation [20].

[1]http://gmplib.org/

5.1 Cumulative Cost

We first present results based on the cumulative cost for processing a query workload. Figure 6 depicts our results for increasing data sizes ranging from 1M to 32M tuples, for Plain Data, Encrypted Data, and Encrypted Data with Ambiguity, along with the SecureScan method (based on simply scanning encrypted data with no ambiguity) in dashed lines; the x-axis in all graphs represents the query sequences, and the y-axis the cumulative response time up to query x. The top 3 graphs in Figure 6 focus on the first part of the workload, i.e., the first 30 queries, while the bottom 3 graphs show the complete query sequences. We focus separately on the beginning of a query sequence as this is where most of the heavy adaptive indexing actions take place, as the pieces being reorganized are much larger at this point [17].

The first row of figures shows that, in the first few queries, cumulative time grows significantly; this effect follows from the nature of cracking, as in the beginning more data is physically reorganized. On the other hand, in the second row of figures we notice that, as the workload evolves, for all data types, the cumulative time almost flattens, a result indicating that progressive physical reorganization renders query processing increasingly more efficient; moreover, this result applies irrespectively of data type, with encryption and with ambiguity as well as without. Furthermore, all

figures show that, as the data size grows, the cumulative time increases in a scalable manner.

By comparing the plots across the three columns in Figure 6, we deduce that the trends observed for plain data are reproduced for encrypted data, as well as for encrypted data with ambiguity (which doubles data size). Remarkably, even the case of encryption with added ambiguity retains the fast convergence benefits of database cracking and vastly outperforms the SecureScan method in cumulative time on large query workloads. Besides, we note that the cumulative time for SecureScan keeps growing even after 50K queries have been processed, while the cumulative time for all cracking-based methods has flattened by that time.

Figure 7 puts together the cumulative times for the three tested types of data for different data sizes, as well as SecureScan for Encrypted Data. The figure shows that plain data require much less time than encrypted data to be processed; e.g., the cumulative time for the first 100 queries for the plain data of size 32M is 1.6 seconds, while the cost of the first query for the encrypted data of size 8M is 28.4 seconds. This is due to the fact that encrypted data necessitate costly vector-based comparisons and the high-precision arithmetic to retain correctness, whereas in the case of plain data we merely compare numbers in integer data type. However, scalability with encrypted data follows the trends for plain data, while the overall overhead for the sake of security remains manageable.

Besides, the figures shows that the cumulative time for encrypted data with ambiguity is double of that for encrypted data of the same size. This is due to the fact that, in the former case, the data size doubles, as each real value spawns a fake value in the database.

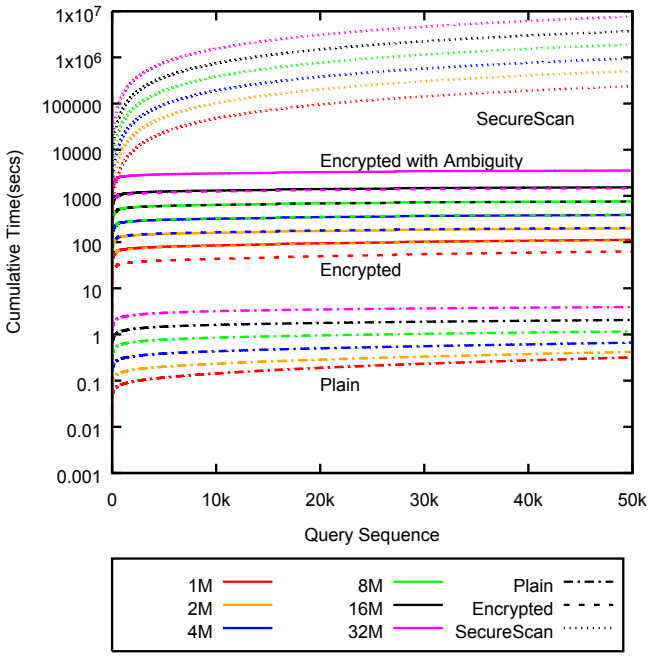


Figure 7: Total Cumulative Time Comparisons

5.2 Cost per Operation

As we have discussed, a cracking system invests its time in three main operations: (a) Cracking (b) Inserting query bounds in the AVL Tree (c) Searching for query bounds in the AVL Tree. We measure the time spent for each of these three purposes. Figure 8 shows the breakdown of the cost per operation for plain data. We observe that the core cracking operation is most expensive for the initial queries, while it becomes progressively cheaper as the query workload evolves. In the case of smaller data sizes, as 1M and 2M, the Crack operation eventually becomes cheaper than the Insert and Search operations within the lifespan of the 50K workload, even while those other operations only take a few microseconds. Yet, overall, the Crack operation is more costly than the Insert and Search operations.

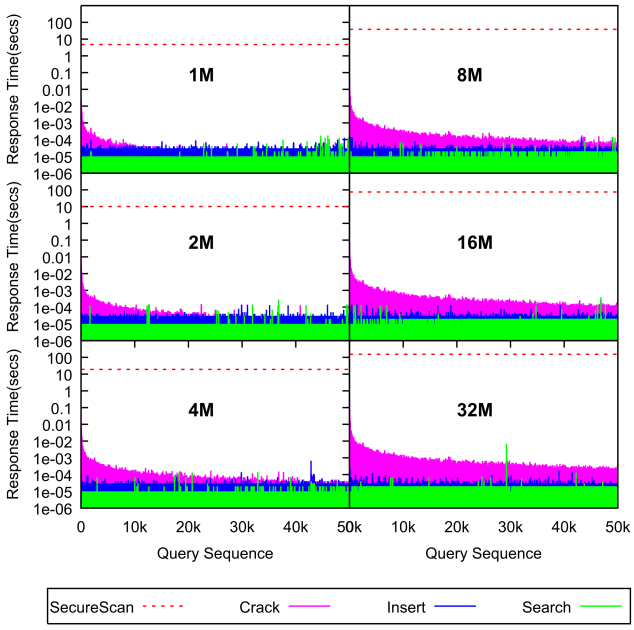


Figure 8: Cost per operation per query on Plain Data

Figure 9 shows the cost per operation and per query for Encrypted Data. As in the case of plain data, the Crack operation be-

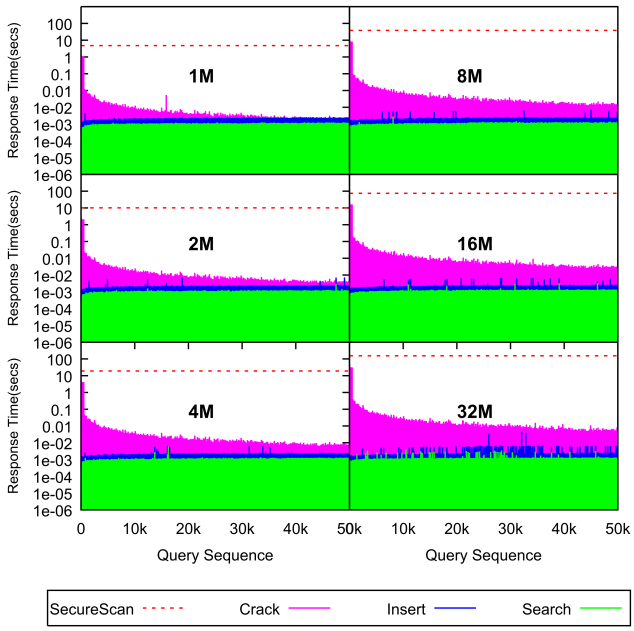


Figure 9: Cost per operation per query on Encrypted Data

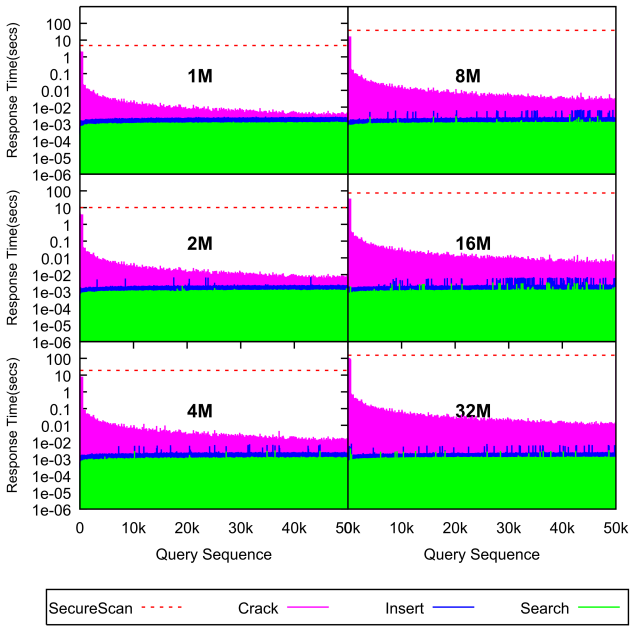


Figure 10: Cost per operation per query on Data with Ambiguity

comes cheaper as the query workload evolves, with response time decreasing significantly. We also note that the Insert and Search operations become more costly from the initial query onwards, with their response times increasing from a few microseconds to a few milliseconds. Moreover, while cracking is more expensive at the early stages, after about 1000 queries for each data size the cost of cracking becomes less than 0.2 second.

Next, Figure 10 shows the cost per query for encrypted data with ambiguous values. All operations' trends remain the same as for encrypted data, with some higher peaks for cracking time, especially in the initial queries. This is due to the fact that physical reorganization is also done for ambiguous values along with real values. Thus, cracking becomes more expensive and burdens the initial queries of the workload. Even so, after about 2000 queries, response time again becomes approximately equal to 0.2 second. Besides, the time for cracking fluctuates a bit, as the cost of a cracking operation depends on the issued query's bounds; if the query falls on a previously unindexed piece, it requires more physical reorganization, hence higher response time.

We now focus on the time for cracking per query in particular. Figure 11 gathers together results for all data types, showing how cracking time grows as data sizes grow. Once gain, we observe that different data types present similar and comparable trends, while the overhead incurred for the sake of encryption and ambiguity, i.e., for the sake of security, remains manageable and reasonable.

5.3 Effect of Key Size

We now study the effect of our encryption key size on the system's performance. Figure 12 presents the total response time per query over encrypted data of size 10M, while varying encryption matrix key size from 4 to 64. This experiment is performed only for regular encrypted data, so as to examine the effect of key size in its native setting. The results we obtain show that a change in the key size bears a significant effect on performance across the query workload. This effect is felt most intensely with initial queries, which carry the heaviest burden anyway. However, in all cases, the

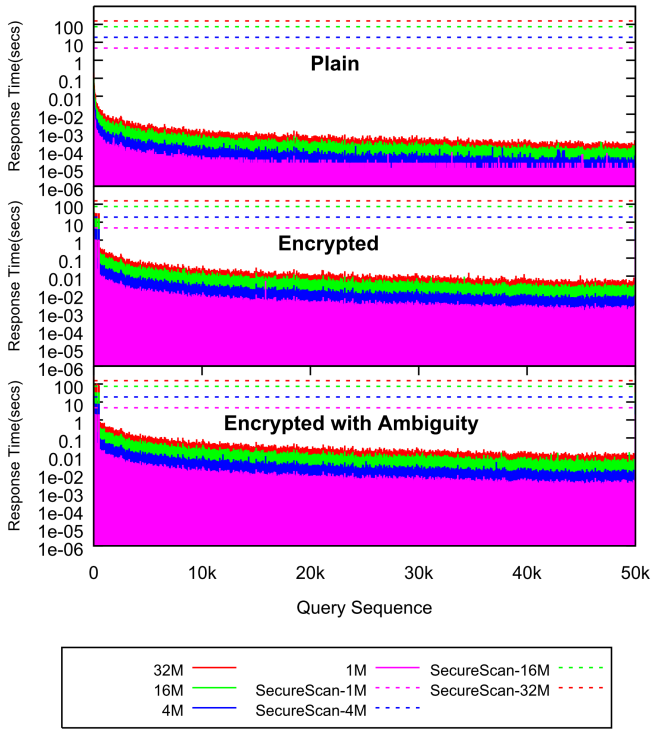


Figure 11: Cracking Time per query per data type

response time increases proportionally to the key size change. For example, the initial response times shown in the figure are 2, 4, 8, 17, and 40 seconds for key sizes of 4, 8, 16, 32, and 64, respectively. The observed effect is due to the fact that vector comparisons become linearly more expensive with an increase of vector size. Nevertheless, as the cracking process progressively amortizes its cost while the query workload advances, the effect of encryption key size becomes negligible; for example, it makes a difference from a millisecond to 0.01 seconds between key size 4 and 64; we argue that this is an overhead that can be accepted for the sake of additional security.

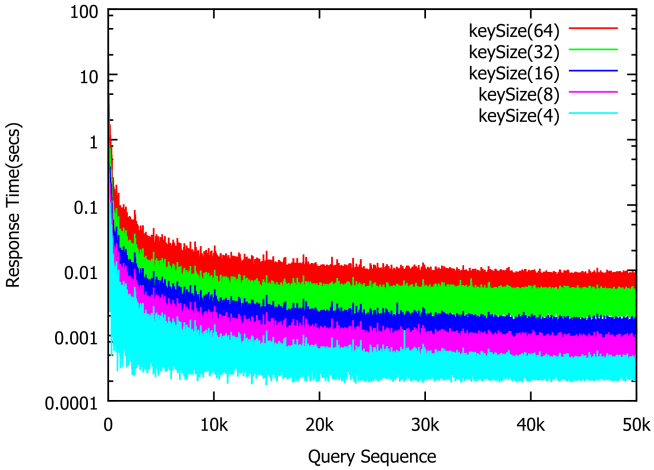
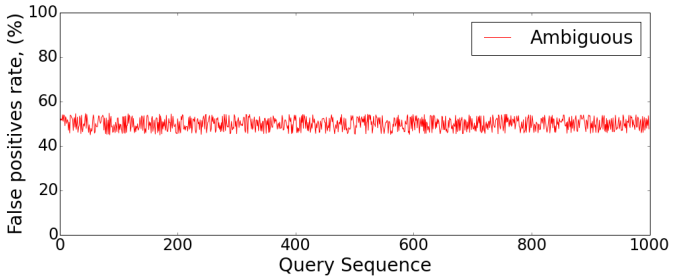


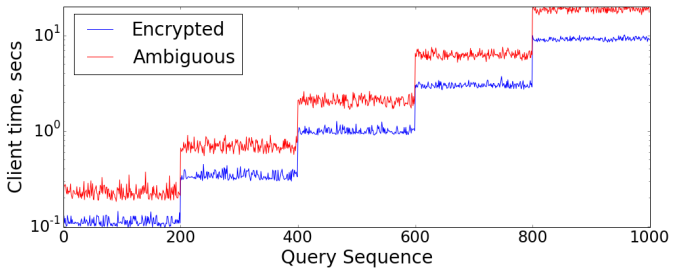
Figure 12: Cracking Time per query for different key sizes for Encrypted Data of size 10M

5.4 Client-Side Performance

All our experiments have hitherto studied the view from the server, as the server performs adaptive indexing actions. Yet a question arises about performance at the client, who receives query results, decrypts them, and, in the case of ambiguity, identifies and removes false positives. To address this question, we run a query workload of 1K random range queries of increasing selectivity from 0.1% upwards in geometric progress (0.1%, 0.3%, 0.9%, 2.7%, 8.1%) over data of size 10M; each group of 200 queries obtains a new selectivity value. Figure 13 shows our results. Figure 13a presents the *false positives rate* (FPR) at the client, i.e., the ratio of fake results over total results delivered to the client per query, while Figure 13b presents the runtime for decryption and filtering per query, comparing the case of Encrypted Data to that of Encrypted Data with Ambiguity, on a logarithmic time axis. We observe that the FPR fluctuates around 50%, as expected, and is unaffected by selectivity; besides, its fluctuation, with a variance of 7.1%, provides an additional security feature, as the exact number of returned results is not revealed and cannot be inferred by an adversary. Furthermore, we note that the decryption overhead is doubled due to ambiguity, presenting stability across different queries of the same selectivity, as well as scalability with increasing selectivity, with each new selectivity group coming one step upwards in the logarithmic time scale in the case with ambiguity just as in the case without ambiguity. Overall, the overhead remains manageable throughout the workload. We conclude that the client-side impact of our scheme is predictable and not detrimental to efficiency.



(a) False Positives Rate



(b) Runtime for Decryption and Filtering

Figure 13: Client-Side Performance with increasing Selectivity

5.5 Discussion

In summary, our experimental analysis shows that our secure adaptive indexing schemes maintain the basic properties of adaptive indexing while working over encrypted data; by incrementally and adaptively focusing on hot data areas without requiring any initialization effort, they achieve significantly better performance compared to simply scanning encrypted data.

Naturally, working over encrypted data is more costly than working over plain data, for two reasons. First, we read and write more data due to auxiliary data for the sake of encryption and ambiguity. Second, we perform more computations, as comparisons are performed by vector operations. Even so, our schemes are several orders of magnitude faster than a secure scan.

An optimization that can be applied on top of our work is the utilization of modern hardware capabilities, such as multi-core CPUs and SIMD (Single Instruction Multiple Data) instructions. Past work has shown how to exploit multi-core CPUs for plain cracking, letting each core work on a single subpartition of a column in parallel and then merging these partitions to create new contiguous partitions [35]. A similar approach can be used in secure adaptive indexing out of the box. On the other hand, past work also attempted to reap the benefits of SIMD for plain cracking, with inconclusive results [35], as it is hard to map cracking comparison and swap actions to current SIMD instruction sets. On the other hand, the inner-product operations, on which our scheme relies (on top of plain cracking) to perform comparisons, are in principle amenable to SIMD-based vectorization: we multiply size-ℓ vectors; such vectorization should bring a significant performance boost, in the best case even *absorb* the computational cost of security at the server (assuming ℓ is not larger than modern SIMD vectors). Thus, secure cracking stands to gain from such optimizations. In our prototype implementation, we have opted for the arithmetic precision obtained by carrying out linear-algebra operations using the GMP arithmetic library, which does not inherently support vectorization, as it does not use primitive data types. In that sense, we have presented a *worst-case scenario* for our scheme in this paper. In our future work, we intend to examine the opportunity for SIMD utilization thoroughly by vectorizing GMP multiplication.

Past work has also speculated that sorting algorithms in highly-parallel environments could pose serious alternatives to plain cracking as modern CPUs evolve [4]. In that regard, we note that a highly-parallel sorting algorithm could not pose a legitimate algorithmic alternative to secure cracking, since our encryption scheme *does not allow* for sorting by cross-tuple comparisons; as we discussed, data encrypted by our scheme can be sorted *only* in a query-triggered manner, relying on encrypted pivot values provided by the client; thereby, our scheme *reinforces the rationale for cracking*.

6. CONCLUSION

This paper presents a novel, lightweight, linear-algebra-based encryption scheme that allows for (i) range query processing over encrypted numeric data outsourced in the cloud, and thereby for (ii) incremental, adaptive indexing whereby only data that are queried by trusted clients get indexed. Our scheme represents numerical values as short vectors, and relies on simple linear-algebra operations for encryption and decryption; it allows neither the actual data values nor their order to be disclosed. While the structure of the index may reveal order in the long-term, this only happens after crucial indexing operations have been performed; furthermore, we propose an additional obfuscation component in our scheme, which deliberately introduces ambiguity in our construction by allowing two variant interpretations of each encrypted value vector. We propose that our scheme assures the security needed in time-critical operations such as high-frequency trading and financial transaction processing over the cloud. We designed and implemented a prototype system that performs basic adaptive indexing operations, in the form of database cracking, over encrypted numeric data. Our experimental study demonstrates that our scheme preserves the graceful adaptation and hands-free self-reorganization advantages of cracking with a reasonable overhead incurred due to encryption.

In the future, we plan to (i) thoroughly analyze the properties of our encryption scheme, and (ii) utilize modern hardware capabilities for improved performance in cryptographic operations.

Acknowledgments

We thank Nickolai Zeldovich, Jaideep Vaidya, and David Cash for fruitful discussions on this topic.

7. REFERENCES

[1] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] A. Agopyan, E. Şener, and A. Beklen. Financial business cloud for high-frequency trading: A research on financial trading operations with cloud computing. *International Journal on Advances in Intelligent Systems*, 4(3):203–2017, 2011.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *SIGMOD*, 2004.

[4] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter. Main memory adaptive indexing for multi-core systems. In *DaMoN*, pages 3:1–3:10, 2014.

[5] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. Transaction processing on confidential data using Cipherbase. In *ICDE*, pages 435–446, 2015.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, pages 224–241, 2009.

[7] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[8] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[9] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *TCC*, pages 535–554, 2007.

[10] S. Bothe, P. Karras, and A. Vlachou. eSkyline: Processing skyline queries over encrypted data. *PVLDB*, 6(12):1338–1341, 2013.

[11] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB*, pages 1–10, 2000.

[12] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *ACM CCS*, pages 93–102, 2003.

[13] T. Ge and S. B. Zdonik. Fast, secure encryption for indexing in a column-oriented DBMS. In *ICDE*, pages 676–685, 2007.

[14] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[15] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, pages 850–867, 2012.

[16] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.

[17] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *TPCTC*, pages 169–184, 2010.

[18] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[19] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[20] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[21] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *VLDB Journal*, 21(3):333–358, 2012.

[22] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, pages 720–731, 2004.

[23] S. Idreos. Database cracking: Towards auto-tuning database kernels. *CWI, PhD Thesis*, 2010.

[24] S. Idreos. Cracking big data. *ERCIM News*, 2012(89), 2012.

[25] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[26] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424, 2007.

[27] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, pages 297–308, 2009.

[28] S. Idreos, S. Manegold, and G. Graefe. Adaptive indexing in modern database kernels. In *EDBT*, pages 566–569, 2012.

[29] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.

[30] E. Laursen. High-end trading strategists see cost savings in cloud computing. http://www.institutionalinvestor.com/Article/2750046/Search/High-End-Trading-Strategists-See-Cost-Savings-in-Cloud.html. [Online; posted on January 14, 2011].

[31] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964, 2014.

[32] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[33] M. Mani, K. Shah, and M. Gunda. Enabling secure database as a service using fully homomorphic encryption: Challenges and opportunities. *CoRR*, abs/1302.2654, 2013.

[34] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, pages 1153–1166, 2015.

[35] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. L. Kersten. Database cracking: fancy scan, not poor man's sort! In *DaMoN*, pages 4:1–4:8, 2014.

[36] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[37] T. Sanamrad, L. Braun, D. Kossmann, and R. Venkatesan. Randomly partitioned encryption for cloud databases. In *DBSec*, pages 307–323, 2014.

[38] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy*, pages 350–364, 2007.

[39] M. Stonebraker et al. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[40] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[41] S. Wang, D. Agrawal, and A. El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *Secure Data Management*, pages 52–69, 2011.

[42] W. H. Wang and L. V. S. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *VLDB*, pages 127–138, 2006.

[43] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis. Secure kNN computation on encrypted databases. In *SIGMOD*, pages 139–152, 2009.

[44] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on encrypted data. In *ESORICS*, pages 479–495, 2006.

[45] M. L. Yiu, G. Ghinita, C. S. Jensen, and P. Kalnis. Enabling search services on outsourced private spatial data. *The VLDB Journal*, 19(3):363–384, 2010.