2013 IEEE Symposium on Security and Privacy

An Ideal-Security Protocol for Order-Preserving Encoding

Raluca Ada Popa
MIT CSAIL

Frank H. Li
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

***Abstract*—Order-preserving encryption—an encryption scheme where the sort order of ciphertexts matches the sort order of the corresponding plaintexts—allows databases and other applications to process queries involving order over encrypted data efficiently. The ideal security guarantee for order-preserving encryption put forth in the literature is for the ciphertexts to reveal *no information about the plaintexts besides order.* Even though more than a dozen schemes were proposed, all these schemes leak more information than order.**

This paper presents the first order-preserving scheme that achieves ideal security. Our main technique is *mutable ciphertexts*, meaning that over time, the ciphertexts for a small number of plaintext values change, and we prove that mutable ciphertexts are needed for ideal security. Our resulting protocol is interactive, with a small number of interactions.

We implemented our scheme and evaluated it on microbenchmarks and in the context of an encrypted MySQL database application. We show that in addition to providing ideal security, our scheme achieves 1–2 orders of magnitude higher performance than the state-of-the-art order-preserving encryption scheme, which is less secure than our scheme.

***Keywords*-order-preserving encryption, encoding**

I. Introduction

Encryption is a powerful technique for protecting confidential data stored on an untrusted server, such as in cloud computing [10, 12, 37]. One limitation of encrypting confidential data is that the data must usually be decrypted for processing by an application—such as querying an encrypted database or sorting encrypted email messages—which requires trusting the server running the application. The approach of computing on encrypted data avoids the need of decrypting the data by a potentially untrustworthy server. While recent work on fully homomorphic encryption shows it is, in principle, possible to perform arbitrary computations over encrypted data [14], the performance overheads are prohibitively high, on the order of 10^9 times [15].

A practical approach for computing over encrypted data is to use encryption schemes that allow an untrusted server to execute specific computation primitives over the ciphertexts. A common operation is order comparison, used for sorting, range checks, ranking, etc. To allow an untrusted server to perform order comparison on ciphertexts, many systems in both research and industry use *order-preserving encryption or encoding* schemes—that is, schemes where $Enc(x) > Enc(y)$ iff $x > y$. We abbreviate an order-preserving encryption or encoding scheme (the latter may not be strictly an encryption scheme) by *OPE*. OPE is primarily used in databases for processing SQL queries over encrypted data [2, 13, 19, 21, 23, 26, 27, 33, 38], although it has also been used in the context of mail servers [3, 9, 32], web applications [9, 32], CRM software [9, 35], and others. OPE is appealing because systems can perform order operations on ciphertexts in the same way as on plaintexts: for example, a database server can build an index, perform SQL range queries, and sort encrypted data, all in the same way as for plaintext data. This property results in good performance and requires minimal changes to existing software, making it easier to adopt.

Order-preserving scheme		Guarantees	Leakage besides order
Özsoyoglu et al.'03	[30]	None	Yes
Agrawal et al.'04	[2]	None	Yes
Boldyreva et al.'09	[6, 7]	ROPF [6], §II-A	Half of plaintext bits
Agrawal et al.'09	[1]	None	Yes
Lee et al.'09	[23]	None	Yes
Kadhem et al.'10	[20]	None	Yes
Kadhem et al.'10	[21]	None	Yes
Xiao et al.'12	[38]	None	Yes
Xiao et al.'12	[39]	IND-OLCPA [39]	Yes
Yum et al.'12	[40]	ROPF [6], §II-A	Half of plaintext bits
Liu and Wang'12	[26]	None	Most of the plaintext
Ang et al.'12	[3]	None	Yes
Liu and Wang'13	[27]	None	Most of the plaintext
This paper*		**Ideal: IND-OCPA**	**None**

Figure 1. Security provided by previous order-preserving encryption or encoding schemes and our work, including the cryptographic security guarantees provided by each scheme, and the information revealed by each scheme in addition to the order of the plaintext values. We elaborate on this information in §II. (*) Unlike prior schemes, our scheme uses an interactive protocol and mutable ciphertexts.

The *ideal* security goal for an order-preserving scheme, IND-OCPA [6], is to reveal no additional information about the plaintext values besides their order (which is the minimum needed for the order-preserving property). Despite a large body of work on OPE schemes [1–3, 6, 7, 20, 21, 23, 26, 27, 30, 38, 39], none of the prior schemes achieve ideal security: as shown in Fig. 1, they all leak more than just the order of values. As we discuss in §II, many schemes assume adversaries try to learn information from ciphertexts in specific ways, but provide no security guarantees for general adversaries. Boldyreva et al. [6] were the first to provide a rigorous treatment of the problem; in fact, they showed that it is infeasible to achieve ideal security for OPE, under certain implicit assumptions. As a result, they settled on a weaker security guarantee that was later shown to leak at least half of the plaintext bits [7]. Thus, current OPE schemes allow an adversary to compromise the privacy of confidential data, beyond just learning the order of the items.

This paper presents the first ideal-security order-preserving encoding scheme where the ciphertexts reveal nothing except for the order of the plaintext values. The insight that allows

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.38

us to avoid Boldyreva et al.'s infeasibility result [6] is that most applications of OPE only require a relaxed OPE interface that need not be as restrictive as the interface of an encryption scheme. In particular, it is acceptable for the encryption protocol to be interactive and for a small number of ciphertexts of already-encrypted values to change as new plaintext values are encrypted (e.g., it is straightforward to update a few ciphertexts stored in a database). We call such a scheme *mutable order-preserving encoding*, or mOPE, to indicate the mutability of ciphertexts, and we use the word *encoding* instead of encryption to emphasize our deviation from the standard model of encryption. Building on this insight, this paper makes several contributions, as follows:

1) We present the first order-preserving scheme that achieves an ideal, rigorous security guarantee for OPE called IND-OCPA [6], which requires that an adversary learns nothing except for the order of values.
2) We show that mutability is in fact required for ideal security: specifically, we show that, without prior knowledge of the values to be encrypted, ideal security for OPE is infeasible even for a relaxed encryption model (stateful and interactive) if ciphertexts are non-mutable.
3) We observe that when considering a database system, an even stronger notion of security is possible and desirable: we define *same-time OPE security* (stOPE), which requires that an adversary only learns the order of items present in the database *at the same time*, and we provide a refinement of mOPE that achieves this stronger definition.

Intuitively, mOPE works by building a balanced search tree containing all of the plaintext values encrypted by the application. The order-preserving encoding of a value is the path from the root to that value in the search tree. Thus, if x is less than y, the path to x will be to the left of the path to y; we represent tree paths using a binary encoding where the encodings increase from left to right in a tree. The search tree is stored on the same untrusted server that stores the encrypted data, and the trusted client encrypts values by inserting them into the tree using an interactive protocol. The length of the path encoding is equal to the depth of the tree, so to ensure ciphertexts do not become arbitrarily long, mOPE rebalances the search tree. This requires updating ciphertexts corresponding to any items whose location in the tree changed as a result of rebalancing, but we show that only a small number of already-encrypted values change ciphertexts for each newly encoded value.

To understand the performance of mOPE, we implemented mOPE (and the same-time OPE security variant) under both a honest-but-curious and a malicious server model, and evaluated it using a range of microbenchmarks. We show that mOPE achieves 1–2 orders of magnitude higher performance than the state-of-the-art OPE scheme by Boldyreva et al. [6] (which does not achieve ideal security).

To demonstrate how mOPE can be used in an application, we use mOPE to execute SQL queries over encrypted data in a MySQL database. We present a *transformation summary* technique to efficiently update mutable ciphertexts using a single `UPDATE` SQL query, and show that updating ciphertexts in the encrypted database application incurs low overheads on queries from the industry-standard TPC-C benchmark.

The rest of this paper is organized as follows. §II discusses related work. §III formally presents our threat model. §IV presents mOPE in more detail, and §V shows that non-mutable secure OPE is infeasible to achieve. §VI extends our mOPE construction to provide *same-time* security, and §VII describes how mOPE handles malicious servers. §VIII illustrates how mOPE can be integrated into a database application. §IX describes our implementation, and §X evaluates the performance of mOPE. Finally, §XI concludes.

II. Related work

There has been a significant amount of work on OPE schemes both in the research community [1, 2, 6, 7, 20, 21, 23, 26, 27, 30, 38, 39] and in industry [3, 9, 32, 35]. The key contribution of this paper lies in providing the first OPE scheme, mOPE, which achieves ideal IND-OCPA security; we discuss prior schemes in more detail shortly (§II-A).

Even an ideal order-preserving scheme must reveal the order of items. Kolesnikov and Shikfa [22] discuss the leakage associated with revealing order in practice, and techniques for minimizing such leakage, which is applicable to mOPE as well as other OPE schemes.

mOPE is also related to cryptographic schemes for performing range queries over encrypted data [8, 19, 28, 36]. These range query schemes aim for a different goal than OPE: instead of requiring that the ciphertext values literally preserve the order of the plaintext values, they separately encrypt *data values* and *query values*, and provide an algorithm by which one can learn the order between a query value and a data value (but not between two query or two data values, and ideally no other information). The fact that the ciphertexts are not themselves order-preserving means such schemes cannot be used with unmodified software as is the goal with OPE. Moreover, no current efficient constructions achieve this ideal goal; some schemes reveal all query values, other schemes reveal some data values, and some schemes provide only approximate answers.

Pandey and Rouselakis [31] introduce the notion of property-preserving encryption (PPE): an encryption scheme where a public predicate can be tested on any k ciphertexts. OPE is a PPE for the order comparison predicate with $k = 2$. Pandey and Rouselakis [31] provide a construction for the inner product predicate, but not for the order predicate.

Because our protocol is interactive, it resembles two-party computation. In fact, our security definition for the malicious server case (§VII) follows the general form of two-party computation definitions. There exist protocols for two-party

computation for any arbitrary function [18], but using such a protocol for order would result in prohibitive costs and is not needed since the state of the server is not secret from the client and the client is trusted.

Finally, OPE is related to work on order-preserving compression [4, 5]. Although some OPE schemes use similar techniques, the requirements for OPE are different: compression is not a requirement for OPE, whereas security (plaintext privacy) is not a requirement for compression.

A. OPE schemes

To put prior work on OPE schemes in perspective, it is easiest to think about the security guarantees achieved by each scheme, which translates into the information leaked by that scheme beyond order. Fig. 1 summarizes the prior work on OPE schemes, which we will now discuss.

The strongest security definition considered by prior work is IND-OCPA, proposed by Boldyreva et al. [6]. IND-OCPA captures the ideal security for order-preserving schemes: an adversary with access to a set of ciphertexts cannot learn anything about the plaintext values except for their order. No existing scheme achieves IND-OCPA; this paper's construction, mOPE, is the first to achieve IND-OCPA.

Boldyreva et al. [6] prove that it is impossible for any OPE scheme to achieve IND-OCPA, under some implicit assumptions about how an OPE scheme works—that ciphertexts are immutable and that encryption is stateless. In §V, we strengthen this impossibility result by showing IND-OCPA is impossible even for stateful OPE schemes, but also show that IND-OCPA is achievable if ciphertexts are mutable.

In this paper we also propose a new security definition, called *same-time OPE security*, which requires that an adversary can learn the order only for items that are stored in the application at the same time. This definition is stronger than IND-OCPA, and we show how an extension of mOPE achieves this stronger definition.

Since none of the prior schemes achieve the ideal IND-OCPA goal, they put forward a variety of alternative security definitions. For example, Boldyreva et al. [6] define the notion of a random order-preserving function (ROPF), and construct an OPE scheme that is indistinguishable from a ROPF, which we will call BCLO [6]. Yum et al. [40] improve the performance of BCLO. It was subsequently shown that the ROPF definition inherently reveals more than order—in fact, at least half of the plaintext bits [7, 24, 25].

Other schemes either provide weaker security definitions by making assumptions about attacks, which are unlikely to hold in practice, or do not provide a security definition or guarantees at all [1–3, 20, 21, 23, 26, 27, 30, 38, 39, 39, 40]. For example, Xiao et al. define IND-OLCPA security by requiring that the adversary learns the encryptions only for "nearby" values [39], although it is unclear how a practical system would enforce this. Other security definitions assume that adversaries are restricted to a specific attack strategy, or that they do not have any additional side information about the values being encrypted, which is similarly hard to ensure in practice. Thus, while such schemes make the job of the adversary more difficult, the level of security they provide is hard to quantify, and many of them allow an adversary to extract a significant amount of information on top of order.

To understand how an adversary can learn additional information from schemes without rigorous guarantees, consider the scheme of Liu and Wang [26], which works as follows. The secret key consists of two integers a and b and the encryption of a value v is $av+b+\mathsf{noise}$, for some randomly chosen $\mathsf{noise} \in \{0,\ldots,a-1\}$, small enough to preserve order. Note that a and b are *the same* across all encryptions. Intuitively, this scheme is insecure because a and b are a one-time pad that is being reused many times. To attack Liu and Wang's scheme, suppose an attacker obtains two pairs of plaintexts and ciphertexts: c_1 is the encryption of 0 (i.e., $c_1 = b + n_1$ for some noise n_1), and c_2 is the encryption of k, where k is some large value (i.e., $c_2 = ak + b + n_2$). By computing the difference between c_2 and c_1, the attacker obtains $c_2 - c_1 = ak + n_2 - n_1$, and since $0 \le n_1, n_2 < a$, the attacker learns that a is in the interval $I_1 = \frac{c_2-c_1}{k+1} \le a \le I_2 = \frac{c_2-c_1}{k-1}$. If k is large, then one or few integers will be between $[I_1, I_2]$ so the attacker gets a. Even if there are a few options in $[I_1, I_2]$, by getting an encryption of another large value k^*, and intersecting the resulting $[I_1^*, I_2^*]$, the adversary gets a. Having a allows an attacker to decrypt all ciphertexts. Liu and Wang's later scheme [27] suffers from the same attack.

III. Model

Our model consists of an OPE client and an OPE server that interact with each other. The client is the owner of the data to be encrypted and thus the client is trusted. The OPE server is untrusted; we consider two settings, corresponding to the following threat models regarding the server:

- *Passive (or honest-but-curious) adversary:* The server follows our protocol correctly and returns correct answers to the OPE client, but it tries to learn information about the data beyond order. For example, the server may log the entire history of operations and data, provide the data to some other party, and even use some side information to try to learn more about the data.
- *Active (or malicious) adversary:* The server can misbehave in any way, such as returning incorrect answers to the client.

We construct two OPE schemes that leak only the order of values in the two respective threat models above. We consider these settings separately because our protocol for the malicious setting is an incremental change to the one for the passive setting, but it also adds more performance overhead. For some applications, the passive setting suffices.

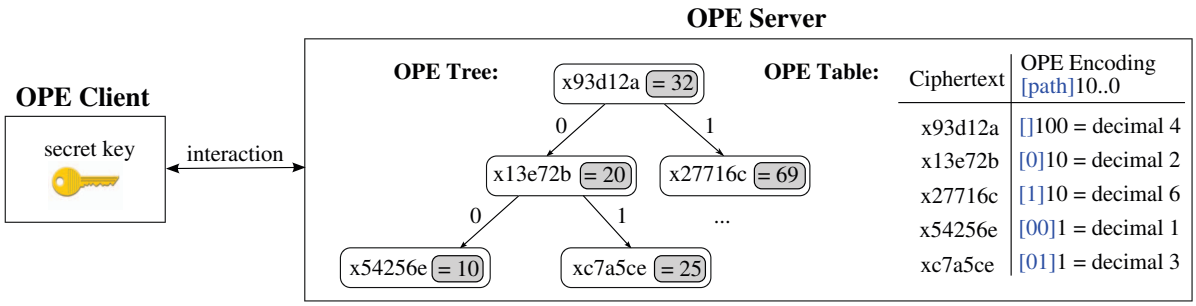

Figure 2. Overview of mOPE's data structures. Each node in the OPE Tree contains a DET ciphertext (the hexadecimal value); for the reader's information, the gray block shows the corresponding plaintext value, but this is not stored in the tree. Child pointers are labeled with 0 or 1 to indicate the path encoding.

A. *Cryptographic preliminaries and notation*

For a distribution $\mathcal{D}$, we write $d \leftarrow \mathcal{D}$ when d is sampled at random from the distribution $\mathcal{D}$. If S is a set, $d \leftarrow S$ means d is sampled from the uniform distribution over the set S. The security parameter throughout the paper is κ. We say that $f(\kappa)$ is negligible if $f(\kappa) = o(\kappa^{-c})$ for every fixed constant c. We write $\mathsf{negl}(\kappa)$ to denote a negligible function.

When saying that a Turing machine A is p.p.t. we mean that A is a uniform probabilistic polynomial time machine.

By RND, we will refer to any standard IND-CCA2 symmetric key encryption scheme, $\mathsf{RND} = (\mathsf{RND.KeyGen}, \mathsf{RND.Enc}, \mathsf{RND.Dec})$. By DET, we will refer to any deterministic encryption scheme whose security property is that of a pseudo-random function [16], $\mathsf{DET} = (\mathsf{DET.KeyGen}, \mathsf{DET.Enc}, \mathsf{DET.Dec})$. Both RND and DET can be implemented with any standard cipher, for example AES, except that DET always uses a constant IV.

IV. Mutable order-preserving encoding (mOPE)

We first present the intuition for our scheme. Let's start by considering an ideal order-preserving encoding and see how we can achieve it. Suppose that the client wants to encrypt five values: 69, 32, 20, 10, and 25. A potential set of order-preserving encodings for these values is 5, 4, 2, 1, and 3, respectively. We can see that this encoding is ideal because it simply informs the server of the order of the values, and nothing else. However, the challenge in achieving such an encoding is that, when the client encodes an input x (e.g., 69 above), it does not know the future values to be encoded, so it does not know the order of x with respect to them (e.g., that 69 will be the fifth value).

Our insight for addressing this challenge is for the client to selectively read previously encoded ciphertexts stored at the server and to mutate a small number of ciphertexts when needed. We show that it is possible to ensure the client reads only a small number of ciphertexts (logarithmic in the total number of encoded values).

Tree construction. Our basic idea is to have the encoded values organized at the server in a *search tree*. A *binary search tree* is a tree in which for each node v, all the nodes in the left subtree of v are strictly smaller than v and all the nodes in right subtree of v are strictly larger than v.

For simplicity of exposition, we present our construction in terms of a binary tree, even though our implementation uses a b-ary B-tree. All of our techniques extend to b-ary trees in a straightforward fashion. We chose a B-tree because it has many advantages for mOPE: it has logarithmic worst-case cost for insert, delete, and lookup, it enables efficient ciphertext updates based on concise transformation summaries (§VIII), and it facilitates verifying that a malicious server produced correct answers (§VII).

In our tree, each node of the tree contains the DET encryption of a value (under the client's secret key sk), but the ciphertexts are arranged in the tree based on the order of the *plaintext* values. Fig. 2 shows such a tree, which we denote the *OPE Tree*. Since the client may encrypt a large number of values, we store this tree on the untrusted server. We can see that the plaintext values in the left subtree of each node are smaller than the node value, and the values in the right subtree are larger.

Of course, the server does not know the underlying plaintext values, so it cannot arrange the tree in this form; the client will help the server find the location in the tree where to insert a value. To illustrate how the client does this, suppose the client wants to encode 55 using the tree state shown in Fig. 2. The client first requests the root node of the tree, and the server returns x93d12a, which the client decrypts to 32. Since $55 > 32$, the client requests the right child of the root from the server, and the server responds with x27716c, which the client decrypts to 69. Finally, the client requests the left child of the last node requested, and the server responds that there is no such child. This means that the client can insert a new node in this position, containing the DET encryption of 55. Note that, crucially, the client told the server *only* order information (namely, if the value being encoded is to the left or to the right of another value), and thus this interaction does not reveal anything else besides the order. Fig. 3 provides the general algorithm.

Binary encoding. So what is the OPE encoding of the newly inserted value 55? Observe that the path from the root

Algorithm 1 (OPE Tree traversal for a value v)**.**

1: Cl $\leftrightarrow$ Ser: The client asks the server for the ciphertext c' at the root of the OPE tree.
2: Cl: The client decrypts c' and obtains v'.
3: Cl $\rightarrow$ Ser: If $v < v'$, the client tells the server "left"; if $v = v'$ the client tells the server "found"; if $v > v'$, the client tells the server "right".
4: Ser: The server returns the next ciphertext c'' based on the client's information, and goes back to step 2.
5: Ser, Cl: The algorithm stops when v is found, or when the server arrives at an empty spot in the tree. The outcome of the algorithm is the resulting pointer in the OPE Tree, the path in the OPE Tree from the root, and whether v was found.

Figure 3. OPE Tree traversal algorithm. The text in blue indicates at which party each piece of computation happens (Cl or Ser).

down to the node indicates the relative order of the node with respect to the other tree nodes. If we label each left edge with a "0" bit and each right edge with a "1" bit, we can represent the path to a node from the root using the bitwise concatenation of labels from the corresponding tree edges. For example, the path for the value 10 is (binary) 00, which is decimal 0; the path of 25 is (binary) 01, which is decimal 1; and the path of 55 is (binary) 10, which is decimal 2. We can see that these values preserve the order of the plaintexts. One has to be careful about nodes higher in the tree. For example, the path of 32 (the root) is the empty string. The empty string is not larger than 0 and smaller than 2. Therefore, we pad all paths to the same length (e.g., 32 or 64 bits in practice) by defining the OPE encoding of a value as follows:

$$\text{OPE encoding} = [\text{path}]10\ldots0, \tag{1}$$

where there are as many zero bits as necessary to pad the value to a desired ciphertext size m. For example, if $m = 3$ as in Fig. 2, the encoding of the root value 32 is decimal 4, the encoding of 10 is decimal 1 and the encoding of 55 is decimal 5. We can see that the order of encodings is preserved for all values.

Tree balancing. To ensure that OPE encodings do not grow too large, mOPE must maintain a logarithmic tree height, which requires occasional balancing operations. For example, in a B-tree, if a node contains too many items, the node gets split into two nodes and the parent node receives an additional child. If the parent node also contains too many items, the split propagates upward.

Tree balancing is precisely what mutates the OPE encoding: after a rebalancing, a node may move to a different part of the tree, thus changing its path in the tree. As we show later in §V, any OPE scheme without mutation must have infeasibly long OPE encodings, and we can see how mOPE's mutation ensures that OPE encodings stay short.

After balancing the OPE Tree, the OPE server must update any server-side storage containing OPE encodings (e.g., update the relevant values in a database). This is why it is important for the OPE server to be co-located with the system using our OPE scheme.

Locating and modifying previously stored encodings can also require a significant amount of time. In §VIII we present a technique called a *transformation summary* that allows us to concisely describe tree rebalancing operations with a short $O(\log n)$ summary, and to precisely scope the range of affected encoding values, so that encoding updates can be performed efficiently in *one pass* over the affected range of values.

Amortized cost. The client work in this protocol is $O(\log n)$, where n is the total number of values encoded, and the order-preserving encoding similarly requires just $O(\log n)$ bits. This is because the tree has logarithmic height. Furthermore, the client need not be involved in rebalancing: even though the server does not know the underlying plaintext values, the server can perform tree rebalancing without any client involvement, because it needs to know only order information, which is already available from the tree on the server.

Let us now examine the server-side cost of updating encodings. Traditional logarithmic cost bounds for trees (such as a B-tree) are computed by considering only the number of nodes touched during a balance. However, the number of affected ciphertexts is the number of children in the subtrees of nodes moved during a balancing operation, which can be asymptotically larger. For example, if one node moves higher up in the tree, only a few nodes may be touched by this rebalancing operation, but all the children of this node change their OPE encodings. In theory, a scapegoat tree provides $O(\log n)$ cost in this model. However, in practice we use a B-tree, even though they have non-logarithmic worst-case cost, because their actual cost in our experiments was less than the cost of scapegoat trees, and few ciphertexts are updated on average (§X). We recommend scapegoat trees be used only when embedding our scheme in theoretical schemes with constraints on server-side asymptotic performance.

Stale encodings. Tree rebalancings pose another challenge because an OPE encoding of a particular value can become stale. Consider a situation where an application first obtains an OPE encoding of some value, by invoking the OPE client, and then performs more work, which causes inserts and tree rebalancings at the server. The rebalancing operations can cause the application's original OPE encoding to become stale, meaning that the encoding no longer corresponds to the value's position in the tree. If the application were to use a stale encoding, it could obtain incorrect results.

To prevent staleness, we introduce a mapping at the server called the *OPE Table*, as shown in Fig. 2. Whenever a new

item is inserted into the tree, or the server rebalances the tree, the OPE Table is updated to map the DET ciphertexts to their new or updated OPE encodings. Using this OPE Table, a client needs to keep track only of DET ciphertexts, which never become stale. Given a DET ciphertext, the server can compute the OPE encoding at any time, without the client's help. Thus, we envision that OPE encodings would be stored only internally at the server, where they can be rewritten on demand. We remark that the OPE Table also helps as a cache, saving client work when encoding a repeated value.

We will next present a formal description of the syntax of this scheme, the scheme itself, and the ideal IND-OCPA security definition for our construction.

A. Syntax and correctness

Since our encoding scheme can mutate ciphertexts, it has a different syntax from a regular encryption scheme. The encoding of a value v can be of two types:

- a *permanent* ciphertext c that refers to v and does not change (corresponding to the DET encryption), and
- a *transient* order-preserving encoding e, where the encoding corresponding to a given value can change over time as the tree is rebalanced at the server; we will refer to this as the *OPE encoding*.

Definition 1 (Mutable order-preserving encoding (mOPE))**.** *A mutable order-preserving encryption scheme for plaintext domain $\mathcal{D}$ is a tuple of polynomial-time algorithms* $\mathsf{mOPE} = (\mathsf{KeyGen}, \mathsf{InitState}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Order})$ *run by a client and a stateful server, where* KeyGen *is probabilistic and the rest are deterministic, and* Enc *is interactive.*

- ***Key generation:*** $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$. KeyGen *runs at the client, takes as input the security parameter κ, and outputs a secret key* sk.
- ***Initializing server state:*** $\mathsf{st} \leftarrow \mathsf{InitState}(1^\kappa)$. InitState *runs as the server, takes as input the security parameter κ, and outputs an initial state* st.
- ***Encryption:*** $(c, \mathsf{st}') \leftarrow \mathsf{Enc}(\mathsf{sk}, v, \mathsf{st})$. Enc *is an interactive algorithm between the client and the server. The inputs to the client are* sk *and v, and the input to the server is the state* st. *At the end, the client obtains a ciphertext c and the server obtains a new state* st'. Enc*'s running time is a fixed polynomial in* $|\mathsf{sk}|$ *and* $|v|$.
- ***Decryption:*** $v \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$. *The client runs* Dec *on the secret key and a ciphertext c, and obtains a plaintext v.*
- ***Ordering:*** $e \leftarrow \mathsf{Order}(\mathsf{st}, c)$. Order *runs at the server, takes as input a state* st *and a ciphertext c, and outputs an order-preserving encoding e.*

In this syntax, $\mathsf{Order}(\mathsf{st}, c)$ produces the OPE encoding of the value corresponding to ciphertext c. In practice, $\mathsf{Order}(\mathsf{st}, c)$ rarely changes with st (since a given node in the tree is rebalanced rarely); thus, we expect the application would store the $\mathsf{Order}(\mathsf{st}, c)$ value on disk, and update it as necessary when tree rebalancings happen. Another nice feature of this syntax is that the algorithms that run at the client return only permanent ciphertexts, so the client need not worry about encodings becoming stale. Fig. 4 presents our mOPE scheme in terms of this syntax.

Algorithm 2 ($\mathsf{KeyGen}(1^\kappa)$ – runs at Cl)**.**

1: Return $\mathsf{sk} \leftarrow \mathsf{DET.KeyGen}(1^\kappa)$.

Algorithm 3 ($\mathsf{InitState}(1^\kappa)$ – runs at Ser)**.**

1: Initialize server state, st, with OPE Tree and OPE Table containing only the values $\pm\infty$.
2: Return st.

Algorithm 4 ($\mathsf{Enc}(\mathsf{sk}, v, \mathsf{st})$ – runs at Cl and Ser)**.**

1: Cl: compute $c \leftarrow \mathsf{DET.Enc}(\mathsf{sk}, v)$ and send c to Ser.
2: Ser: if c is in the OPE Table,
   Ser: return st unchanged. Cl: return c.
3: Ser: else
   a) Cl $\leftrightarrow$ Ser run the OPE Tree traversal (Alg. 1) so the server inserts c in the OPE Tree and obtains the path of c. Ser then computes the OPE encoding of c based on Eq. 1 and stores it in OPE Table.
   b) Ser: If the OPE Tree needs to be rebalanced as a result of the insertion, rebalance the tree and update all affected encodings in the OPE Table.
   c) Ser: return the new state. Cl: return c.

Algorithm 5 ($\mathsf{Dec}(\mathsf{sk}, c)$ – runs at Cl)**.**

1: Return $v \leftarrow \mathsf{DET.Dec}(\mathsf{sk}, c)$.

Algorithm 6 ($\mathsf{Order}(\mathsf{st}, c)$ – runs at Ser)**.**

1: If c is in OPE Table return the corresponding OPE encoding from the OPE Table, else signal error.

Figure 4. The mOPE scheme. The algorithms that have client-server interaction contain text in blue indicating at which party each piece of computation happens (Cl or Ser).

We now turn to defining what it means for the scheme to be correct. Intuitively, the scheme should decrypt the correct values and Order should indeed output order-preserving ciphertexts. Consider encrypting a sequence of values $\mathsf{seq} = v_1, \ldots, v_n$. The state evolves after each encryption, from st_0 to st_n, by successively computing $(c_i, \mathsf{st}_i) \leftarrow \mathsf{Enc}(\mathsf{sk}, v_i, \mathsf{st}_{i-1})$ for $i = 1 \ldots n$, where $\mathsf{st}_0 \leftarrow \mathsf{InitState}(1^\kappa)$.

Definition 2 (Correctness)**.** *A mOPE scheme for plaintext domain $\mathcal{D}$ is correct if, for all security parameters κ, for all* $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$,

1) *for all $v \in \mathcal{D}$ and for all states* st*, for every c outcome of* $\mathsf{Enc}(\mathsf{sk}, v, \mathsf{st})$, $\mathsf{Dec}(\mathsf{sk}, c) = v$*; and,*
2) *for all sequences* $\mathsf{seq} = \{v_1, \ldots, v_n\} \in \mathcal{D}^n$*, for all pairs $v_i, v_j \in$* seq*, for all c_i, c_j obtained as above, we have*

$$v_i < v_j \Leftrightarrow \mathsf{Order}(\mathsf{st}_n, c_i) < \mathsf{Order}(\mathsf{st}_n, c_j).$$

B. Security definition

We now define the "ideal" security of mOPE, which intuitively says that the scheme must not leak anything besides order. The security definition is the IND-OCPA definition presented in Boldyreva et al. [6], except that we adapt it to the syntax of our encoding scheme. The definition says that an adversary cannot distinguish between encryptions of two sequences of values as long as the sequences have the same order relation.

In this section, we assume that the server is passive (see §III), and treat malicious adversaries in §VII.

IND-OCPA security game. The security game between a client Cl and an adversary Adv for security parameter κ proceeds as follows:

1) The client Cl generates $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$ and chooses a random bit b.
2) The client Cl and the adversary Adv engage in a polynomial number of rounds of interaction in which the adversary is adaptive. At round i:
   a) The adversary Adv sends values $v_i^0, v_i^1 \in \mathcal{D}$ to the client Cl.
   b) The client Cl leads the interaction for the Enc algorithm on inputs sk and v_i^b with the server Ser, with Adv observing all the state at Ser.
3) The adversary Adv outputs b', its guess for b.

We say that the adversary Adv wins the game if (1) its guess is correct ($b = b'$), and (2) the sequences $\{v_i^0\}_i$ and $\{v_i^1\}_i$ have the same order relations (namely, for all i, j, $v_i^0 < v_j^0 \Leftrightarrow v_i^1 < v_j^1$). Let $\mathsf{win}^{\mathsf{Adv},\kappa}$ be the random variable indicating the success of the adversary in the above game.

Definition 3 (IND-OCPA: indistinguishability under an ordered chosen-plaintext attack)**.** *A mOPE scheme is IND-OCPA secure if for all p.p.t. adversaries* Adv*, for all sufficiently large* κ*,* $\Pr[\mathsf{win}^{\mathsf{Adv},\kappa}] \leq 1/2 + \mathsf{negl}(\kappa)$.

C. Security proof

Theorem 1. *Our mOPE scheme is IND-OCPA secure.*

Due to space constraints, we point to Appendix B of our extended paper [34] for the proof, and provide intuition here.

Proof intuition. Consider any adversary Adv and any two sequences of values Adv asks for in the security game: $\mathbf{v} = (v_1, \ldots, v_n)$ and $\mathbf{w} = (w_1, \ldots, w_n)$. The view of Adv consists of the information the server receives in the security game. The first step is to use the security guarantees of the DET encryption scheme, and assume that DET encryptions are computationally indistinguishable from random values that have the same pattern of repetitions (e.g., produced by a random oracle). Next, we examine the information the adversary learns in case the client encrypts $\mathbf{v}$ and in case the client encrypts $\mathbf{w}$, and show that this information is information-theoretically the same.

For this goal, we proceed inductively in the number of values to be encrypted. The base case is when no value was encrypted and we can see that Adv starts off with the same information. Now consider that after i encryptions, Adv obtains the same information in both cases, and we show that the information after step $i+1$ also remains the same. At step $i+1$, Cl and Adv run Enc from Fig. 4 for v_i and w_i.

We have two possibilities. The first possibility is that the encoding of v_i is in the OPE Table. Then the encoding of w_i is also in the OPE Table (and vice versa) because $\mathbf{v}$ and $\mathbf{w}$ have the same order relation; in particular, $v_i = v_j$ iff $w_i = w_j$ so the pattern of repetitions will be the same. In this case, Cl does not give any information to Adv.

The second possibility is that the encoding of v_i (and therefore of w_i) is not in the OPE Table. Cl and Adv interact according to Alg. 4 in both cases. Since $\mathbf{v}$ and $\mathbf{w}$ have the same order relation, the path down the tree taken by Cl and Adv must be the same. Also, the only information the client gives the server is which edges to take in this path, which is also the same for both cases.

Therefore, Adv receives the same information in both cases, and hence cannot distinguish between them. □

V. Impossibility of non-mutable stateful OPE

The previous section showed that mOPE achieves IND-OCPA security, but it changes the traditional model of an encryption scheme, most notably via ciphertext mutability (there have been interactive and/or stateful encryption schemes before). A natural question is whether there exists an IND-OCPA secure scheme that is stateful, but does not mutate ciphertexts. In this section, we demonstrate that mutable ciphertexts are needed, by showing that even a stateful and interactive encryption scheme cannot feasibly achieve IND-OCPA without prior knowledge of the values to be encrypted.

Boldyreva et al. [6] showed that any IND-OCPA-secure OPE scheme must have ciphertext sizes at least exponential in the size of the plaintext. For example, encrypting 64-bit numbers would require 2^{64}-bit ciphertexts, which is impractical. Their impossibility result implicitly assumes a traditional model where encryption is a stateless function, thus leaving open the question of whether an IND-OCPA-secure stateful OPE scheme is possible. We show that even such a scheme is infeasible: namely, any such scheme would also have exponentially large ciphertexts. Due to space constraints, we present only a proof sketch, and refer the reader to Appendix C of our full paper [34] for more details.

Let us define IND-OCPA security for stateful OPE informally. Let EncOrd be the encryption algorithm of a stateful OPE scheme. EncOrd takes as input a secret key, a current state, and a plaintext x, and outputs a ciphertext c and a new state. Let $\mathsf{EncOrd}(x)$ denote informally the encryption of x under the appropriate secret key and state.

We say that EncOrd is IND-OCPA-secure if all polynomial-time adversaries Adv can win the following game with probability of at most $1/2 + \mathsf{negl}(\kappa)$:

- Adv provides two sequences of numbers $\mathbf{x} = \{x_1, \ldots, x_n\}$ and $\mathbf{y} = \{y_1, \ldots, y_n\}$, such that they have the same order relation (namely, $x_i < x_j \Leftrightarrow y_i < y_j$, for all i, j).
- Adv is given an encryption of one of these sequences: either $\{\mathsf{EncOrd}(x_1), \ldots, \mathsf{EncOrd}(x_n)\}$ or $\{\mathsf{EncOrd}(y_1), \ldots, \mathsf{EncOrd}(y_n)\}$. The values in these sequences are encrypted in order; that is, when encrypting x_i, EncOrd receives the state from encrypting x_{i-1}.
- Adv wins the game if it guesses correctly whether it received an encryption of $\mathbf{x}$ or $\mathbf{y}$.

Theorem 2. *Any stateful OPE scheme that is IND-OCPA-secure has ciphertext size exponential in the plaintext size.*

Proof intuition. (See the full proof in Appendix C of the extended paper [34].) We need to show that there exists a polynomial-time adversary Adv that can break any scheme that has shorter than exponentially large ciphertexts. Pick any stateful IND-OCPA-secure OPE scheme and let EncOrd be its encryption algorithm.

Let us first build some intuition, by considering the impossibility result of Boldyreva et al. [6] and showing why it does not suffice here. Let $1, \ldots, N$ be the possible plaintext values, with a corresponding plaintext size of $\approx \log N$. Boldyreva et al. specifies an attacker Adv that can break any stateless IND-OCPA scheme, unless the scheme has large ciphertexts. Adv picks a random value m in $\{2, \ldots, N-2\}$ and outputs the sequences $\mathbf{x} = (1, m, m+1)$ and $\mathbf{y} = (m, m+1, N)$. Then, when receiving $\mathbf{c} = (c_1, c_2, c_3)$, the encryption of one of these sequences, Adv checks if c_1 and c_2 are further apart than c_2 and c_3 are, in which case it outputs "I guess $\mathbf{x}$", else it outputs "I guess $\mathbf{y}$". The attack will fail if, for example $\mathsf{EncOrd}(m) - \mathsf{EncOrd}(1) < \mathsf{EncOrd}(m+1) - \mathsf{EncOrd}(m)$. Boldyreva et al. use the term *long jump* to denote a large difference between two consecutive values, such as $\mathsf{EncOrd}(m+1) - \mathsf{EncOrd}(m)$. The adversary fails only if there are many long jumps, but as Boldyreva et al. show, many long jumps imply an exponential ciphertext size.

However, when the scheme is stateful, this argument no longer applies. One can no longer argue that if $\mathsf{EncOrd}(m) - \mathsf{EncOrd}(1) < \mathsf{EncOrd}(m+1) - \mathsf{EncOrd}(m)$, then the ciphertext space is huge: there can be a scheme EncOrd that is specialized to counter this attack using state. Such a scheme can adjust the encryption of $m+1$ based on the previous two values encrypted, 1 and m, which it can recall using state.

To show that the ciphertext size blows up even for stateful schemes, our idea is to design an attack such that no stateful scheme can specialize to counter it. For this, an adversary provides a long trace of challenges but chooses only one at random on which to guess.

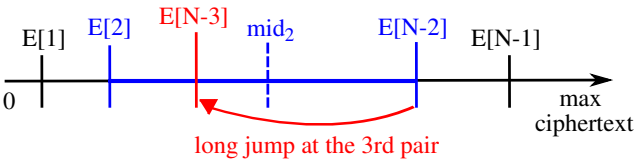

Figure 5. The OPE encryptions of the values in the trace $(1, N-1), (2, N-2), N-3$. E denotes encryption with EncOrd. The axis shows the ciphertext space from a ciphertext of zero to a maximum ciphertext value.

Let us define our adversary Adv. Adv uses a trace of T pairs of challenges $\{(1, N-1), (2, N-2), (3, N-3), \ldots, (T, N-T)\}$, for some polynomial $T = T(\kappa)$. Adv shuffles the order of values in each pair, by choosing random bits: we define $(v, w)^b$ to be (v, w) if $b = 0$, or (w, v) if $b = 1$. Then, Adv chooses a random pair t on which to guess on.

Algorithm 7 (Adversary $\mathsf{Adv}(\kappa)$)**.**

1: Choose t randomly from $1, \ldots, T$.
2: Choose t random bits $b_1, \ldots, b_t$.
3: Construct and output two challenge sets:

$$\mathbf{x} = \{(1, N-1)^{b_1}, (2, N-2)^{b_2}, \ldots, (t, N-t)^{b_t}, t+1\}, \text{ and}$$
$$\mathbf{y} = \{(1, N-1)^{b_1}, (2, N-2)^{b_2}, \ldots, (t, N-t)^{b_t}, N-(t+1)\}.$$

4: Receive encryptions $c_1, \ldots, c_{2t+1}$; Adv has to decide if they correspond to $\mathbf{x}$ or $\mathbf{y}$.
5: Let mid_t be the middle of the interval of $[\mathsf{EncOrd}(t), \mathsf{EncOrd}(N-t)]$. If $c_{2t+1} < \mathsf{mid}_t$, output "it is $\mathbf{x}$", else output "it is $\mathbf{y}$".

Intuitively, the role of the bits b_i is to ensure that EncOrd does not know which challenge pair the adversary chose; namely, EncOrd does not know the value of t.

The adversary Adv exploits the intuition that $\mathsf{EncOrd}(i+1)$ should be closer to $\mathsf{EncOrd}(i)$ than to $\mathsf{EncOrd}(N-i)$ (and the reverse for $\mathsf{EncOrd}(N-(i+1))$. Whenever this is not the case, we have a *long jump*; concretely, we say there is a long jump at pair $i+1$ if $c_{2i+1} > \mathsf{mid}_i$ and c_{2i+1} is the encryption of $i+1$ or if $c_{2i+1} < \mathsf{mid}_i$ and c_{2i+1} is the encryption of $N-(i+1)$. Fig. 5 shows an example. We can see the OPE encryptions resulting from encrypting the trace $1, N-1, 2, N-2, N-3$. This trace corresponds to $\mathbf{y}$ for $t = 2$ and $b_1 = b_2 = 0$. There is a long jump at pair 3.

When there is a long jump at pair $t+1$, Adv guesses incorrectly; otherwise, Adv guesses correctly. If there are few long jumps in the pairs $1, \ldots, T$, then the chance that pair $t+1$ has a long jump is small, so Adv guesses correctly most of the time. But EncOrd is secure against any adversary, so it must be the case that there are many long jumps. This means that the ciphertext size is exponentially large in the plaintext size, because each long jump halves the ciphertext space (see full proof for details).

□

Algorithm 8 (InitState(1^κ) – runs at Cl and Ser)**.**

1: Cl: sk $\leftarrow$ RND.KeyGen(1^κ). The client tells the server to initialize itself.
2: Ser: initializes state st with OPE Tree and OPE Table containing only $\pm\infty$ (with a placeholder $\perp$ for the ciphertext).

Algorithm 9 (Insert(v) – runs at Cl and Ser)**.**

1: Cl: computes $c \leftarrow$ RND.Enc(sk, v) and sends c to the server.
2: Cl $\leftrightarrow$ Ser run the OPE Tree traversal (Alg. 1) so the server locates where c should be inserted, and computes the corresponding OPE encoding e based on Eq. 1. If a ciphertext c_t for v was already in the OPE Tree, the server just increases its ref-count. Otherwise, if v was not already in the tree, Ser inserts c in the tree with a ref-count of 1 and stores it in OPE Table.
   - Ser: If the OPE Tree needs to be rebalanced, the server rebalances it and updates the OPE Table with the new OPE encodings, and any other storage system containing these OPE encodings (e.g., database).
3: Cl, Ser: The algorithm returns c if v did not exist in the OPE Tree, else it returns the ciphertext from the tree c_t.

Algorithm 10 (Remove(v) – runs at Cl and Ser)**.**

1: Cl: computes $c \leftarrow$ RND.Enc(sk, v) and sends c to the server.
2: Cl $\leftrightarrow$ Ser run the OPE Tree traversal (Alg. 1) so the server finds the node to remove. If such a node does not exist, signal an error. If the node exists (and has ciphertext c_t), the server decreases its ref-count and, if the ref-count becomes zero, removes the node from the tree and removes c_t from the OPE Table. If node removal triggers tree balancing, the server similarly updates the OPE Table and any storage containing these OPE encodings.
3: The algorithm returns c_t, the ciphertext corresponding to v from the OPE Tree.

Algorithm 11 (Query(v) – runs at Cl and Ser)**.**

1: Cl: computes $c \leftarrow$ RND.Enc(sk, v) and sends c to the server.
2: Cl $\leftrightarrow$ Ser run the OPE Tree traversal (Alg. 1). The server uses the strategy in Claim 3 to locate the tightest interval.
3: The server returns the ciphertexts for these interval margins and the flag indicating if v is equal to the left margin.

Algorithm 12 (Order(c) – runs at Ser)**.**

1: If c is in the OPE Table, return the corresponding OPE encoding, else signal an error.

Figure 6. The stOPE scheme. The algorithms contain text in blue indicating at which party (or parties) each piece of computation happens (Cl or Ser).

VI. Storage-aware order-preserving encoding

Previous work [6, 7] considered IND-OCPA to be the ideal security for OPE schemes because it captures the desired "only order leaks" idea. We argue that in a real system, a stronger security notion is possible and in fact more desirable.

Consider a database containing three values, $\{20, 32, 69\}$, encrypted with some IND-OCPA scheme. The user may remove the value 32 and later insert a new value 55. The server will learn the order relations of $\{20, 32, 69\}$ and $\{20, 55, 69\}$ as desired, but it will also learn the order of 55 with respect to 32. In the extreme case, if all possible values get encrypted over the lifetime of a system, the server could learn the exact values of the items from their order, even if only a few are present in the database at the same time.

To process order queries over the database, the server needs to know order relations only among values currently in the database; the order of current values with respect to old values need not be revealed. Ideally, a scheme should leak only the order of values that were present in the database *at the same time*. Furthermore, some values may be used in database queries without being inserted into the database. For those values, the server should learn their order with respect to the stored values in the database at the time of the query, so it can process queries efficiently. However, the server should not learn anything else (e.g., their order w.r.t. old values or old queries).

We call the resulting security notion *same-time OPE security* because it reveals order relations only among items present at the same time in a system. We define this stronger security notion and also provide a protocol that provably achieves it. We call the scheme *storage-aware OPE* (stOPE) because it considers the model of a storage system (and also, it applies more generally to storage, and not only to database-type storage). stOPE is a refinement of mOPE so when we refer to both schemes, we simply use mOPE.

The reason why the existing "ideal" definition (IND-OCPA) does not capture this problem is that the encryption model itself does not even permit such a definition: there is no way to indicate that an item was removed using the standard interface of an encryption scheme. The encoding model we

adopt allows us to capture such a definition because it is closer to what happens in a storage system.

A. Syntax

Definition 4 (Storage-aware order-preserving encoding (stOPE))**.** *A stOPE scheme for a plaintext domain $\mathcal{D}$ is a tuple of polynomial-time algorithms* $\mathsf{stOPE} = (\mathsf{Init}, \mathsf{Insert}, \mathsf{Remove}, \mathsf{Query}, \mathsf{Order})$*, where the first four are interactive between two stateful machines, client* Cl *(probabilistic) and server* Ser *(deterministic). All algorithms take as input the states of the two parties;* Init*,* Insert*, and* Remove *also update these states (but we do not make state explicit for simplicity).*

- ***Initialization*** $\mathsf{Init}(1^\kappa)$***:*** *The client generates a secret key* sk *as part of its initial state, while the server initializes its state as well.*
- ***Insert*** $\mathsf{Insert}(v)$***:*** *The client inserts an encoding of a value $v \in \mathcal{D}$ at the server. Both the client and the server obtain as output an encryption c of v.*
- ***Remove*** $\mathsf{Remove}(v)$***:*** *The client removes an encoding of a value $v \in \mathcal{D}$ from the server, if such a value exists at the server. The client's output is a ciphertext c of v.*
- ***Query:*** $\mathsf{Query}(v)$ *takes as input a value $v \in \mathcal{D}$ and returns the tightest enclosing interval for v (two ciphertexts c_1, c_2 and a bit b), where c_1 and c_2 are the ciphertexts of two values v_1, v_2 that are in storage such that $v_1 \leq v < v_2$ and this is the tightest such interval, and the bit b is true iff $v_1 = v$.*
- ***Order:*** $\mathsf{Order}(c)$ *takes as input a ciphertext c and outputs an order-preserving encoding e.*

Correctness. The correctness property required of the scheme is straightforward: Query should return only ciphertexts that exist at the server and that represent the tightest enclosing interval. To define what it means for a value to "exist" at the server, we use a reference count or *ref-count*. The ref-count of a value v is the number of times v was inserted at the server by Insert minus the number of times v was removed by Remove. The correctness requirement for Order is the same as in mOPE and need hold only for values at the server. For simplicity, stOPE does not define a decryption algorithm. One can always support decryption by appending a RND encryption to the ciphertext c.

B. The stOPE scheme

Fig. 6 presents the stOPE scheme, based on mOPE; the rest of this section explains the differences.

To ensure that the order of current values w.r.t. old values does not leak, the first step is to remove values from the OPE Tree when they get deleted from the database (Alg. 10). This ensures that when the client encrypts a new value v and traverses the OPE Tree together with the server, the client will not reveal the order relation of v to removed values. However, if there are duplicates of a value in a database, we should not remove them all, so we use ref-counts at each node of the OPE Tree to keep track of these.

To give an example of why order between removed and new values will not leak, consider an encrypted database containing three values $\{20, 32, 69\}$ with order-preserving encodings 2, 4, and 6. Suppose the user deletes 32 and later inserts 55. After the OPE Tree traversal, 55 will be assigned an OPE encoding of 4, the same as the previous encoding of 32, thus not revealing the order between 55 and 32.

Note that we can no longer use DET for the ciphertexts because DET reveals whether a new value equals some previously deleted value, and we want to leak order information only about items present at the server at the same time. Therefore, the ciphertexts in the OPE Tree now use RND. Every time we request a value v as part of Insert, Remove, or Query, the client encrypts it with new randomness for RND, which is required by the desired security goal. Thus, the OPE Table can no longer function as a cache; it serves only to implement the Order algorithm (Alg. 12) efficiently.

Finally, we explain how the server finds the tightest enclosing interval for Query. The following claim provides a strategy and its proof follows from the properties of a B-tree.

Claim 3. *For any node v in a B-tree containing both $\pm\infty$ values, the largest value v_{left} such that $v_{\mathsf{left}} \leq v$ is:*

- *the rightmost child in the left subtree of v, or*
- *v_{left} is the first value on the path from v to the root such that the right edge of that node is on the path.*

Since $\pm\infty$ are in the tree, a left margin always exists. A symmetric claim holds for the right margin. To compute the margins for a value v that is not in the tree, consider the empty spot returned by Alg. 1 where such a value v would be inserted, and apply Claim 3 to this virtual node.

The intuition for why this scheme is secure is simple: based on the security of mOPE, only order among data items in the OPE Tree leaks, and because old values are removed from the tree, the client will not disclose order with respect to them. We present a full security definition and a proof in Appendix D of the extended paper [34].

The problem becomes more challenging when the adversary is malicious, which we consider next.

VII. Malicious (active) server

A malicious server may alter its responses to the client in an attempt to learn additional information on top of the order of encrypted values. For example, it might not delete old values from the OPE Tree, which would leak the order with respect to old values, violating same-time OPE security.

To force the server to perform these operations correctly, we add Merkle hashing on top of the OPE Tree and use it to check the correctness of the server's responses. Merkle trees [29] are a common method for enforcing integrity properties, so using them for integrity is not novel to this work; only the way we integrate it with our scheme is new. Fig. 7 shows the contents of an OPE Tree with Merkle hashes.

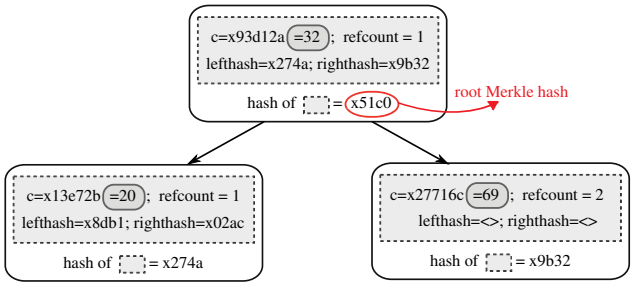

Figure 7. OPE Tree with Merkle hash and metadata; compare to the Merkle-free version shown in Fig. 2.

The Merkle hash at a node is a hash over the data at this node and the Merkle hashes of its children.

A. Merkle tree verification

To verify the correctness of each operation performed by the server, the client stores a copy of the root Merkle hash, and requests proofs from the server for each operation. For example, in order for the client to check if a node v is in the OPE Tree, the server has to provide a *sibling-path*: this consists of all the information (including hashes) at the nodes on the path from v up to the root, and the information at all the siblings of these nodes. Using the properties of the Merkle tree, in order to establish that v is in the tree, it suffices for the client to compute the Merkle root corresponding to the sibling-path and see if it matches the client's stored hash.

In the rest of this section, we describe how the client checks each OPE Tree operation for stOPE. This approach can be similarly applied to mOPE, but we omit it for brevity. We also omit the formal security definition and proof, which are included in Appendix D of our extended paper [34].

Proofs of deletion and insertion. When the client requests insertion or deletion of an item, the server needs to provide a proof that it indeed inserted or deleted the item. The proof consists of:

1) Old Merkle information: the information at the nodes in the tree that were affected by the insertion/deletion, together with the sibling paths of these nodes. (In a B-tree, there is just one sibling-path corresponding to the lowest node in the tree involved in the operation.)
2) New Merkle information: the new sibling-path with values and hashes after the deletion.

The client checks the insertion or deletion proof by:

1) Using the old Merkle information, the client computes the root of the Merkle tree and verifies that it agrees with the current Merkle root the client has.
2) The client checks that the new information is correct: the node was inserted or removed correctly in the B-tree or the appropriate ref-count was updated, and any other metadata at nodes was not altered.
3) The client computes the root of the new tree and stores this updated Merkle hash.

Proofs of correctness for Order **and** Query**.** To prove correctness of Order results, the server includes the sibling-path from where the node is located in the tree to the root. Using the client's Merkle root hash, the client checks that the hash of the path corresponds to the client's Merkle root, the path starts from the desired node, and the OPE encoding is correct based on the path from the root to the node.

To check the results of Query(v), the client has to ensure that the returned interval is not only enclosing but also tight. For this, the client uses the characterization of the interval margins from Claim 3. To prove tightness of the left margin for a value v, let N be the node in the tree that either contains v or that would be the node where v would be inserted (as determined during the client–server interaction). The server supplies the sibling-path of the rightmost element in the left subtree of N, or the sibling-path of N if N has no left child. The client checks against its root Merkle hash that this path is indeed a valid path from the tree containing N. To verify the result is tight, the client checks that it satisfies Claim 3 using the information from each node on this path (which contains whether a node has a right or left child and the children hashes). Treatment of the right margin is symmetric.

VIII. Using mOPE in a database application

In this section, we explain how to use mOPE in a database, which is the primary application for OPE schemes and for mOPE. As mentioned in the introduction, OPE allows efficient order computations on an encrypted database because the database server can compute order on OPE encodings in the same way as on unencrypted data (e.g., database indexes work the same way), and the database server software does not need to be modified. Using mOPE in an encrypted database improves security over other OPE schemes such as BCLO [6], currently used in CryptDB [33], because mOPE does not leak any information besides order. Using stOPE would provide an even stronger *same-time* security guarantee.

Setup. Using mOPE (or similarly stOPE) in a database requires the following setup:

- A trusted client-side application that uses the mOPE client to encode values. Unmodified applications can be supported by using a proxy that intercepts and rewrites the application's SQL queries [33].
- User-defined functions (UDFs) in the database server that implement mOPE's Order function and update encodings in the database.

The client application maintains separate mOPE client state (e.g., secret key) for every column encrypted with OPE.

Insert queries. To understand how values in a query are encrypted, consider an application that wants to execute the query `INSERT INTO secret VALUES (5)`. The application first encrypts 5 using mOPE.Enc (Fig. 4) and obtains $c \leftarrow$ DET.Enc(5). It then issues the query `INSERT INTO secret`

VALUES (MOPE_ORDER(*c*)), where MOPE_ORDER is a UDF implementation of mOPE.Order.

Transformation summaries. If mOPE rebalances its tree, it must update OPE encodings stored in the database. One challenge lies in describing the transformation from old to new encodings in a succinct way; a naïve approach of storing a mapping between all old and new encodings would scale poorly when a large subtree (e.g., the root) is rebalanced.

Updates due to a B-tree balancing can be summarized succinctly in a *transformation summary* as a sequence of split and merge operations. Consider a b-ary B-tree. For each node that split, we record the location where an item was inserted in the node causing the split, and the location of the split (from 1 to b). If there are N items in the tree, there are at most $\log N$ such splits for a single mOPE insertion, because the height of the B-tree is bounded by $\log N$. We record similar information for node merges in the case of deletions in stOPE. We can further determine a tight interval $[\mathsf{low}, \mathsf{high}]$ of the OPE encodings that must change. The server can then update all affected OPE encodings with one SQL query of the form UPDATE *secret* SET *item*=MOPE_TRANSFORM(*item*, *summary*) WHERE *item*>=*low* AND *item*<=*high*, where MOPE_TRANSFORM is a UDF that adjusts encodings based on the transformation summary *summary*.

Select queries. First consider a simple query: SELECT * FROM *secret* WHERE *val*>5. As before, the proxy computes mOPE.Enc of 5 to obtain c and then rewrites the query into SELECT * FROM *secret* WHERE *val*>MOPE_ORDER(*c*). At the server, the UDF MOPE_ORDER returns the OPE encoding of 5, and the server executes the query on encrypted data as if the data were not encrypted.

For simple queries that perform comparison on exactly one encoded column (as in the example above), the work of the client is the same as the work of the server because executing the query consists of performing one tree traversal and collecting the output; in this case, the client saves only on storage. However, most realistic queries perform filtering on other fields at the same time. In such cases, the work of the server is more significant than the work of the client because the server has to process joins, compute filters on various columns, intersect results, and so on, before sending a potentially small result set to the client. For example, one query from the industry-standard TPC-C benchmark requests stock items from warehouse number 2 with order numbers between 1245 and 1873 and quantity less than 1.0:

```
SELECT COUNT(DISTINCT(s_i)) FROM order_line, stock
WHERE ol_w=2 AND s_w=2 AND ol_d=3 AND s_i=ol_i AND
      ol_o<1873 AND ol_o>1245 AND s_quantity<1.0
```

Such queries also show why it is important to store the OPE encoding explicitly in the database rather than just using the OPE Tree, which implicitly stores the order of values: such a complicated query can now be processed without changing the DBMS and without incurring overheads from comparing the relative positions of values in the OPE Tree.

Bulk loading. mOPE allows efficient bulk loading for a database by constructing the entire mOPE state in a single pass. Bulk loading is often used to initially load a large data set into a database. To encrypt a large batch of plaintext values, mOPE simply sorts the values, builds a B-tree on top of the sorted values in linear time, and uploads the tree to the server. The resulting OPE Tree is a valid state, and can support subsequent individual insertion operations. We demonstrate the efficiency of this approach in §X-B.

Concurrency. Databases often allow multiple queries on the same column to execute concurrently. With mOPE, modifications of either the OPE state (e.g., due to Enc) or the encoded values stored in a database table (e.g., due to tree balancing) must be carefully ordered with respect to other modification or lookup operations (e.g., invocations of Order). Our current prototype issues one query at a time, although more fine-grained ordering may be possible, perhaps using fine-grained tree locking [11].

IX. Implementation

We implemented mOPE and stOPE, including malicious server protection (§VII), in 3,480 lines of C++ code (of which the largest portion is 950 lines for Merkle verification). We also added support for our scheme in MySQL, with transformation summaries, UDFs, and bulk loading as discussed in §VIII, to demonstrate its usage for performing SQL queries on encrypted data. We made no change to the MySQL code because UDFs are part of the MySQL interface. For DET and RND, we use Blowfish for 32- and 64-bit plaintexts and AES for larger plaintext sizes.

X. Evaluation

To evaluate the performance of mOPE, we answer the following questions in the respective subsections:

- How is the mOPE encoding time affected by various parameters, such as plaintext size, number of total items encoded, number of items encoded in a batch, order of items encoded, malicious vs. honest-but-curious server scenarios, and network latency? (§X-B)
- How does the encoding time using mOPE compare to the encryption time using the state-of-the-art OPE scheme by Boldyreva et al. [6] (BCLO)? (§X-B)
- What are the storage costs of mOPE? (§X-C)
- How many ciphertext updates does mOPE perform, and how costly are they in a real application? (§X-D)

A. Experimental setup

We measured the performance of mOPE and BCLO (our shorthand for the scheme of Boldyreva et al. [6]) on a machine with an Intel Core 2 Q9400 processor running

Linux 3.3 with only a single core enabled for consistency, running both the client and the server on the same machine. For network experiments, we use a second identical machine connected over a Gigabit network link, and we use Linux `tc` to simulate additional network latency. We configure mOPE to produce 64-bit OPE encodings, although none of our experiments come close to a tree that deep. We evaluate our schemes for the honest-but-curious (*HbC*) as well as malicious server model.

We use an optimized implementation of the BCLO scheme from CryptDB [33]. We extend it to support more efficient batch encryption by caching HGD samples when encrypting a batch of values at once.

B. Throughput

Fig. 8 shows the throughput of mOPE in several configurations as a function of the number of items encoded. Overall, mOPE's throughput in each configuration goes down slightly as the number of encrypted items goes up by orders of magnitude. This is caused by the OPE Tree's depth increasing logarithmically with the number of encrypted items. The top two curves show mOPE's throughput when the items are encrypted in sequential order starting from 1, and when they are encrypted in random order. mOPE achieves nearly identical performance in these two cases, because the B-tree always remains balanced. Enabling malicious server checking using Merkle hashing reduces mOPE's throughput by 3.3× compared to an honest-but-curious scenario. The bottom curve shows the throughput of BCLO; it does not depend on the number of items, and mOPE outperforms it by 43× at 10^3 items in the honest-but-curious case.

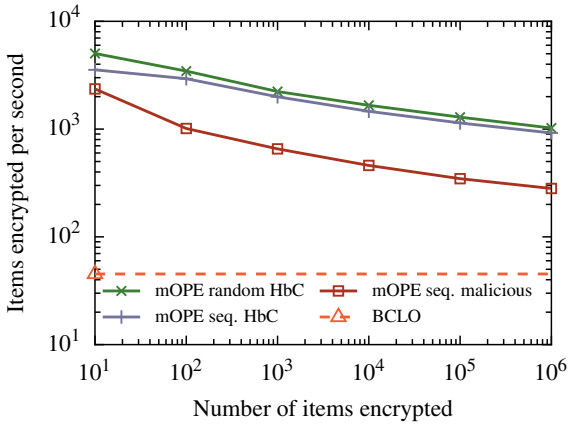

Figure 8. Throughput of mOPE and BCLO for 64-bit plaintext values in the honest-but-curious (HbC) server setting and in the malicious server setting, under either a random or a sequential distribution of inputs. BCLO's performance is independent of the number of items encrypted.

Applications often need to load large amounts of data, making batch encryption an important workload. Fig. 9 shows the throughput of mOPE for batch encryption. mOPE achieves significantly higher throughput in batch mode than in single-item encryption (shown in Fig. 8), since it does not traverse the tree for every item. mOPE's throughput at small batch sizes is dominated by tree construction startup costs. Constructing the Merkle tree adds a 1.6× overhead. mOPE in HbC mode achieves 83× higher batch throughput than BCLO at 10^5 items, even when using our HGD caching optimization for BCLO with sequential inputs (the ideal case for HGD caching).

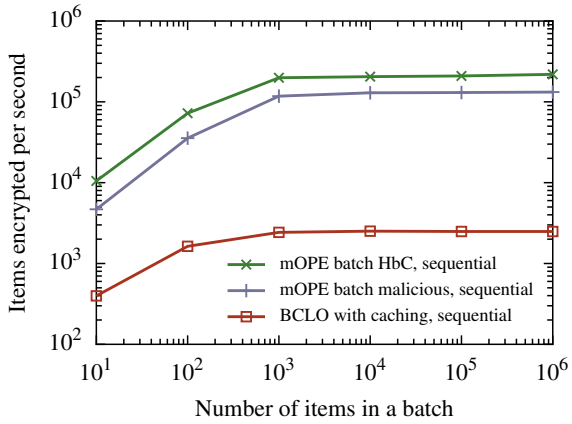

Figure 9. Throughput of mOPE and BCLO when encrypting a batch of 64-bit plaintext values at once, as a function of the batch size.

Fig. 10 shows the effect of plaintext sizes on throughput. mOPE is largely unaffected by plaintext sizes, in both the HbC and malicious server settings; plaintext size affects only the size of tree nodes and the time spent encrypting and decrypting tree nodes using a block cipher. The throughput of BCLO, however, drops *exponentially* as the plaintext size increases; for 256-bit values (which correspond to relatively short strings), BCLO takes 176 msec to encrypt a single value, compared to < 1 msec for mOPE in the HbC model.

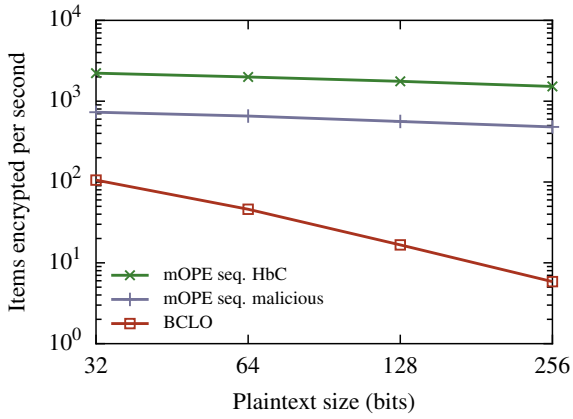

Figure 10. Throughput of mOPE and BCLO when encrypting 1000 plaintext values of different size.

Since mOPE requires interaction with a server to encode a value, its performance depends on the network latency between the client and the server. To evaluate this, we considered two scenarios. The first is a worst-case scenario where the client encrypts one value at a time; this forces the

application to suffer the network latency for each value. The second is an application with a parallel workload that allows it to encrypt multiple values at the same time, so that the network latency can be amortized and overlapped for multiple values; to ensure there is always enough parallel work to keep the pipeline full, we used 5,000 concurrent threads. Fig. 11 shows the results. As expected, the single-threaded client's performance drops as the network latency increases. However, mOPE maintains high throughput for a concurrent client that performs multiple operations in parallel.

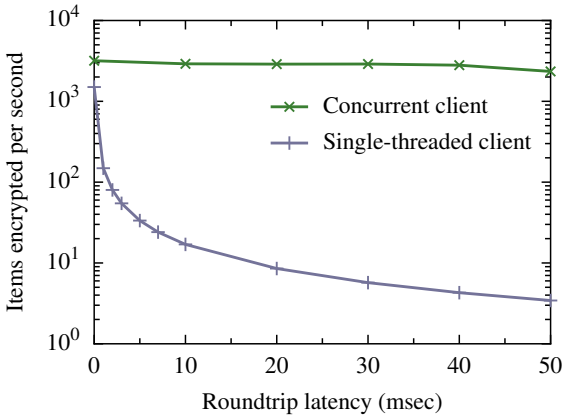

Figure 11. Throughput of mOPE as a function of the client-server network latency, for a single-threaded client and for a concurrent client that performs multiple encryptions in parallel.

C. Storage and ciphertext sizes

To evaluate the storage cost of mOPE, we measured the size of the on-disk representation of mOPE's B-tree. On average, we found that mOPE stores 40 bytes per encrypted value, when encrypting 64-bit values.

D. Ciphertext update cost

To understand the impact of ciphertext updates in mOPE, we measured the average number of existing ciphertexts updated in a database when encrypting a new value. Fig. 12 shows the results. We can see that the number of rewrites is around 2–4, and grows slowly as the number of items increases. This shows that mOPE's order-preserving encodings are relatively stable, and do not change often as new values are encrypted. A sequential workload incurs fewer rewrites, because our 4-way B-tree fills nodes from left to right, and incurs fewer rebalancings when inserting increasing values.

To understand the cost of ciphertext updates in a real system, we used mOPE to encrypt data in a SQL database. We analyzed the throughput of the scheme on both a sequential workload and on a trace of `INSERT` and `UPDATE` queries from TPC-C, an industry-standard database benchmark (as mentioned in §VIII, we issue only one query at a time). Fig. 13 shows the throughput when ciphertexts are updated in the database (using our transformation summaries) and when the ciphertexts are not updated. Overall, the cost of ciphertext

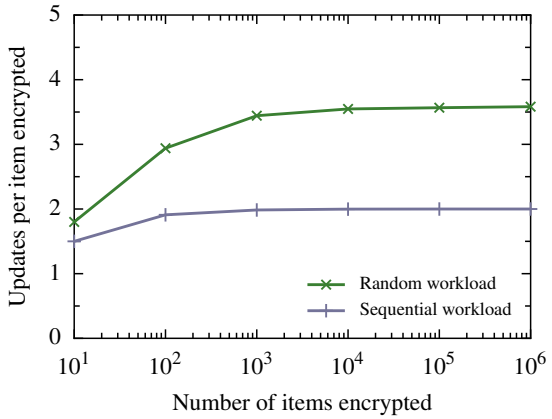

Figure 12. Number of ciphertexts updated for each additional item encrypted in mOPE, as a function of the total number of encrypted items.

updates is modest. We can see that for the sequential workload, rewrites cause a small drop in throughput of less than 15%. For TPC-C, the cost of rewrites is even smaller because TPC-C inserts a large number of repeating values, which do not modify mOPE's tree, and thus do not trigger rewrites. For TPC-C, the first few values take longer to encrypt because they are unique, and later throughput increases due to repetitions.

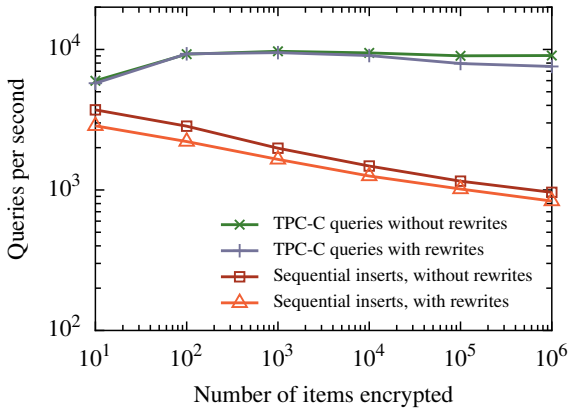

Figure 13. Throughput of SQL queries per second for sequential `INSERT` queries and for the TPC-C mix of `INSERT` and `UPDATE` queries, as a function of the database size.

XI. Conclusion

We presented mOPE, the first order-preserving encoding scheme that achieves ideal IND-OCPA security, where an adversary learns nothing but the order of elements based on the ciphertexts. mOPE uses the idea of mutable ciphertexts, and we show that mutable ciphertexts are required to achieve IND-OCPA. We propose a stronger notion of *same-time OPE security* that allows an adversary to learn only the order of elements present in an encrypted database at the same time, and present an extension of mOPE, called stOPE, that achieves this stronger definition. We also present versions of mOPE and stOPE that protect against a malicious server

by using Merkle hashing. Finally, we show that mOPE achieves good performance both in microbenchmarks and in the context of an encrypted database running TPC-C queries, and that it outperforms the state-of-the-art OPE scheme by 1-2 orders of magnitude.

Acknowledgments

We thank Emily Stark, the anonymous reviewers, and our shepherd, Srđan Čapkun, for their feedback. This work was supported by NSF award IIS-1065219 and by Google.

References

[1] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database management as a service: Challenges and opportunities. In *ICDE*, 2009.

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

[3] G. W. Ang, J. H. Woelfel, and T. P. Woloszyn. System and method of sort-order preserving tokenization. US Patent Application 13/450,809, 2012.

[4] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving compression. In *ICDE*, 1996.

[5] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *ACM SIGMOD*, 2009.

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

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

[8] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography*, 2007.

[9] CipherCloud. Tokenization for cloud data. http://www.ciphercloud.com/tokenization-cloud-data.aspx.

[10] R. Falkenrath and P. Rosenzweig. Encryption, not restriction, is the key to safe cloud computing. *Nextgov*, 2012. http://www.nextgov.com/cloud-computing/2012/10/o/58608/.

[11] K. Fraser and T. Harris. Concurrent programming without locks. *ACM TOCS*, 25(2), 2007.

[12] Gazzang. Gazzang zNcrypt: Transparent data encryption to fit your cloud. http://www.gazzang.com/products/zncrypt.

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

[14] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC*, 2009.

[15] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, 2012.

[16] O. Goldreich. *Foundations of Cryptography: Volume I Basic Tools*. Cambridge University Press, 2001.

[17] O. Goldreich. *Foundations of Cryptography: Volume II Basic Applications*. Cambridge University Press, 2004.

[18] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM STOC*, 1987.

[19] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *ACM SIGMOD*, 2002.

[20] H. Kadhem, T. Amagasa, and H. Kitagawa. MV-OPES: Multivalued-order preserving encryption scheme: A novel scheme for encrypting integer value to many different values. *IEICE Trans. on Info. and Systems*, E93.D(9), 2010.

[21] H. Kadhem, T. Amagasa, and H. Kitagawa. A secure and efficient order preserving encryption scheme for relational databases. In *International Conference on Knowledge Management and Information Sharing*, 2010.

[22] V. Kolesnikov and A. Shikfa. On the limits of privacy provided by order-preserving encryption. *Bell Labs Technical Journal*, 17(3), 2012.

[23] S. Lee, T.-J. Park, D. Lee, T. Nam, and S. Kim. Chaotic order preserving encryption for efficient and secure queries on databases. *IEICE Trans. on Info. and Systems*, E92.D(11), 2009.

[24] X. Liangliang, O. Bastani, and D. Lin. Security analysis for order preserving encryption schemes. Technical Report UTDCS-01-12, University of Texas at Dallas, 2012.

[25] I.-L. Y. Liangliang Xiao and D. Lin. Security analysis for an order preserving encryption scheme. Technical Report UTDCS-06-10, University of Texas at Dallas, 2010.

[26] D. Liu and S. Wang. Programmable order-preserving secure index for encrypted database query. In *IEEE International Conference on Cloud Computing*, 2012.

[27] D. Liu and S. Wang. Nonlinear order preserving index for encrypted database query in service cloud environments. *Concurrency and Computation: Practice and Experience*, 2013.

[28] Y. Lu. Privacy-preserving logarithmic-time search on encrypted data in cloud. In *NDSS*, 2012.

[29] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[30] G. Özsoyoglu, D. A. Singer, and S. S. Chung. Anti-tamper databases: Querying encrypted databases. In *IFIP WG 11.3 Working Conf. on Database and Applications Security*, 2003.

[31] O. Pandey and Y. Rouselakis. Property preserving symmetric encryption. In *EUROCRYPT*, 2012.

[32] Perspecsys. The PRS server: Data protection for cloud applications. http://www.perspecsys.com/perspecsys-cloud-protection-gateway/.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM SOSP*, 2011.

[34] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. Cryptology ePrint Archive, Mar. 2013. http://eprint.iacr.org/.

[35] F. Y. Rashid. Salesforce.com acquires SaaS encryption provider Navajo Systems. *eWeek.com*, 2011.

[36] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig. Multi-dimension range query over encrypted data. In *IEEE Symposium on Security and Privacy*, 2007.

[37] Vormetric. Cloud encryption. http://www.vormetric.com/products/encryption/cloud-encryption/.

[38] L. Xiao, I.-L. Yen, and D. T. Huynh. Extending order preserving encryption for multi-user systems. Cryptology ePrint Archive, Report 2012/192, 2012.

[39] L. Xiao, I.-L. Yen, and D. T. Huynh. A note for the ideal order-preserving encryption object and generalized order-preserving encryption. Cryptology ePrint Archive, Report 2012/350, 2012.

[40] D. Yum, D. Kim, J. Kim, P. Lee, and S. Hong. Order-preserving encryption for non-uniformly distributed plaintexts. In *Intl. Workshop on Information Security Applications*, 2011.