

# Arx: An Encrypted Database using Semantically Secure Encryption

Rishabh Poddar  
UC Berkeley  
rishabh@berkeley.edu

Tobias Boelter  
UC Berkeley  
t.b@berkeley.edu

Raluca Ada Popa  
UC Berkeley  
raluca.popa@berkeley.edu

## ABSTRACT

In recent years, encrypted databases have emerged as a promising direction that provides data confidentiality without sacrificing functionality: queries are executed on encrypted data. However, many practical proposals rely on a set of weak encryption schemes that have been shown to leak sensitive data. In this paper, we propose Arx, a practical and functionally rich database system that encrypts the data only with semantically secure encryption schemes. We show that Arx supports real applications such as ShareLaTeX with a modest performance overhead.

### PVLDB Reference Format:

Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An Encrypted Database using Semantically Secure Encryption. *PVLDB*, 12(11): 1664-1678, 2019. DOI: <https://doi.org/10.14778/3342263.3342641>

## 1. INTRODUCTION

Due to numerous data breaches [7, 25], the public concern over privacy and confidentiality is likely at one of its peaks today. In recent years, encrypted databases [5, 67, 72, 83] (EDBs) have emerged as a promising direction towards achieving both confidentiality and functionality: queries run on encrypted data. CryptDB [72] demonstrated that such an approach can be practical and can support a rich set of queries; it then spurred a rich line of work including Cipherbase [5] and Monomi [83]. The demand for such systems is demonstrated by the adoption in industry such as in Microsoft’s SQL Server [58], Google’s Encrypted Big Query [33], and SAP’s SEEED [34] amongst others [20, 39, 45, 78]. Most of these services are *NoSQL databases* of various kinds showing that a certain class of encrypted computation suffices for many applications.

Unfortunately, this area faces a **challenging privacy-efficiency tradeoff**, with no known practical system that does not leak information. The leakage is of two types: leakage from data and leakage from queries.

**Leakage from data** is leakage from an encrypted database, *e.g.*, *relations among data items*. In order to execute queries efficiently, the EDBs above use a set of encryption schemes

some of which are *property-preserving* by design (denoted PPE schemes), *e.g.*, order-preserving encryption (OPE) [8, 9, 71] or deterministic encryption (DET). OPE and DET are designed to reveal the *order* and the *equality relation* between data items, respectively, to enable fast order and equality operations. However, while these PPE schemes confer protection in some specific settings, a series of recent attacks [26, 37, 61] have shown that given certain auxiliary information, an attacker can extract significant sensitive information from the order and equality relations revealed by these schemes. These works demonstrate *offline* attacks in which the attacker steals a PPE-encrypted database and analyzes it offline.

**Leakage from queries** refers to what an (*online*) attacker can see during query execution. This includes all observable state in memory, along with which parts of the database are touched (called access patterns), including *which (encrypted) rows* are returned and *how many*, which could be exploited in certain settings [15, 35, 44, 48]. Unfortunately, hiding the leakage due to queries is very expensive as it requires oblivious protocols (*e.g.*, ORAM [81]) to hide access patterns, along with aggressive padding [60] to hide the result size. For instance, Naveed [60] shows that in some cases it is more efficient to stream the database to the client and answer queries locally than to run such a system on a server.

A natural question is then: how can we protect a database from offline attackers as well as make progress against online attackers, while *still providing rich functionality and good performance*?

We propose Arx, a practical and functionally rich database system that takes an important step in this direction by *always* keeping the data encrypted with semantically secure encryption schemes. Semantic security implies that no information about the data is leaked (other than its size and layout), preventing the aforementioned offline attacks on a stolen database. For an online attacker, Arx incurs pay-as-you-go information leakage: the attacker no longer learns the frequency count or order relations for *every* value in the database, but *only for data involved in the queries it can observe*. In the worst case (*e.g.*, if the attacker observes many queries over time), this leakage could add up to the leakage of a PPE-based EDB, but in practice it may be significantly more secure for short-lived online attackers. As prior work points out [10, 18, 53], this model fits the “well-intentioned cloud provider” which uses effective intrusion-detection systems to prevent attackers from observing and logging queries over time, but fears “steal-and-run” attacks. For example, Microsoft’s Always Encrypted [58] advocates this model.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

*Proceedings of the VLDB Endowment*, Vol. 12, No. 11

ISSN 2150-8097.

DOI: <https://doi.org/10.14778/3342263.3342641>

Unfortunately, there is little work on such EDBs, with most work focusing on PPE-based EDBs. The closest to our goal is the line of work by Cash *et al.* [16, 17] and Faber *et al.* [28], which builds on searchable encryption. As a result, these schemes are significantly limited in functionality—they do not support common queries such as order-by-limit, aggregates over ranges, or joins—and are also inefficient for write operations (*e.g.*, updates, deletes). Furthermore, for certain *online* attackers, these systems have some extra leakage not present in PPEs, as we elaborate in §11. To replace PPE-based EDBs, we need a solution that is always at least as secure as PPE-based EDBs.

Overall, by exclusively using semantically secure encryption, Arx prevents the offline attacks above [26, 37, 61] from which PPE-based EDBs suffer. For online attackers, Arx is *always* either more or as secure as PPE-based EDBs.

## 1.1 Techniques and contributions

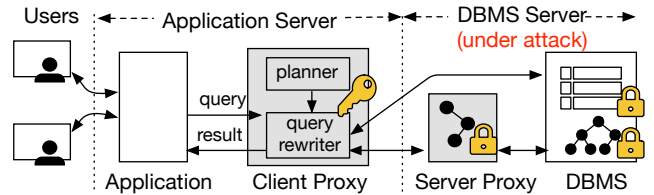
A simple way to protect against offline attacks is to keep the data encrypted at rest in the database and only decrypt it when it is in use; however, such an approach directly leaks the secret key even to a short-lived attacker who succeeds in taking a well-timed snapshot of memory. A better alternative is to consider a hybrid design that uses a PPE-based EDB, but employs a second layer of encryption for data at rest on the disk. However, an attacker who similarly captures the decryption key when it is in memory gets access to the PPE-encrypted data, rendering the second layer of encryption useless. While both approaches enable rich functionality, they leak more information than our goal in Arx.

Instead, Arx introduces two new database indices, **ArxRange** and **ArxEq** that encrypt the data with semantic security; queries on these indices reveal only a limited per-query access pattern. **ArxRange** is for range and order-by-limit queries, and **ArxEq** is for equality queries. While **ArxRange** can be used for equality queries as well, **ArxEq** is substantially faster.

To enable range queries, **ArxRange** builds a tree over the relevant keywords, and stores at each node in the tree a *garbled circuit* for comparing the query against the keyword in the node [31, 86]. Our tree is history-independent [4] to reduce structural leakage. The main challenge with **ArxRange** is to avoid interaction (*e.g.*, as needed in BlindSeer [68]) at every node on a tree path. To address this challenge, Arx draws inspiration from the theoretical literature on Garbled RAM [29]. Arx chains the garbled circuits on a tree in such a way that, when traversing the tree, a garbled circuit produces input labels for the child circuit to be traversed next. Thereby, the whole tree can be traversed in a single round of interaction. For security, each such index node may only be used once, so **ArxRange** essentially *destroys itself for the sake of security*. Nevertheless, only a logarithmic number of nodes are destroyed per query, and Arx provides an efficient repair procedure.

**ArxEq** builds a regular database index over encrypted values by embedding a counter into repeated values. This ensures that the encryption of two equal values is different and the server does not learn frequency information. To search for a value  $v$ , the client provides a small token to the server, which the server expands into many search tokens for all occurrences of  $v$ . **ArxEq** provides forward privacy [12], preventing old tokens from being used to search new data.

Building on top of **ArxRange**, Arx speeds up aggregations by transforming them into tree lookups via **ArxAgg**.



**Figure 1:** Arx’s architecture: Shaded boxes depict components introduced by Arx. Locks indicate that sensitive data at the component is encrypted.

Because of the new indices, index and query planning become challenging in Arx. The application’s administrator specifies a set of regular indices, thereby expecting a certain asymptotic performance. However, regular indices do not directly map to Arx’s indices because Arx’s indices pose new constraints. The main constraints are: Arx cannot use the same index for both  $=$  and  $\geq$  operations, an equality index on  $(a, b)$  cannot be used to compute equality on  $a$  alone, and range queries requires an **ArxRange** index. With this in mind, we designed an **index planning algorithm** that guarantees the expected asymptotic performance while building few additional indices.

Finally, we designed **Arx’s architecture** so that it is amenable to adoption. Two lessons [70] greatly facilitated the adoption of the CryptDB system: do not change the DB server and do not change applications. Arx’s architecture, presented in Fig. 1, accomplishes these goals. The difference over the CryptDB architecture [72] is that it has a server-side proxy, a frontend for the DB server. The server proxy converts encrypted processing into regular queries to the DB, allowing the DB server to remain unchanged.

We **implement and evaluate Arx** on top of MongoDB, a popular NoSQL database. We show that Arx supports a wide range of real applications such as ShareLaTeX [77], the Chino health data platform [19], NodeBB forum [63], and Leanote [51] amongst others. In particular, Chino is a cloud-based platform that serves the European medical project UNCAP [84]. Chino provides a MongoDB-like interface to medical web applications (running on hospital premises) but currently operates on *plaintext* data. The project’s leaders confirmed that Arx’s model fits Chino’s setup perfectly. Finally, we also show that Arx’s overheads are modest: it impacts the performance of ShareLaTeX by 11% and the YCSB [22] benchmark by 3–9%.

## 2. OVERVIEW

In the rest of this paper, we use MongoDB/NoSQL terminology such as collections (for RDBMS tables), documents (for rows), and fields (for columns), but we use SQL format for queries because we find MongoDB’s JS format harder to read. While we implement Arx for MongoDB, its design applies to other databases as well.

### 2.1 Architecture

Arx considers the model of an application that stores sensitive data at a database (DB) server. The DB server can be hosted on a private or public cloud. Fig. 1 shows Arx’s architecture. The application and the database system remain unmodified. Instead, Arx introduces two components between the application and the DB server: a trusted client proxy and an untrusted server proxy. The client proxy

exports the same API as the DB server to the application so the application does not need to be modified. The server proxy interacts with the DB server by invoking its unmodified API (*e.g.*, issuing queries); in other words, the server proxy behaves as a regular client of the DB server. Unlike CryptDB, Arx cannot use user-defined functions instead of the server proxy because the proxy must interact with the DB server multiple times per client query.

The client proxy stores the master key. It rewrites queries, encrypts data, and forwards the rewritten queries to the server proxy for execution along with helper cryptographic tokens. It forwards all queries without any sensitive fields directly to the DB server. The client proxy is *lightweight*: it does not store the DB and does much less work than the server. The client proxy stores metadata (schema information), a small amount of state, and optionally a cache. The server runs the expensive part of DB queries, filtering and aggregating many documents into a small result set.

In most cases, the client proxy processes only the results of queries (*e.g.*, to decrypt them). However, in some corner cases, it performs some post-processing; as a result, our implementation needs to duplicate some parts of the typesystem and expression evaluation logic of the server database.

## 2.2 Threat Model

Arx targets attackers to the database server. Hence, our threat model assumes that the attacker does not control or observe the data or execution on the client-side, and may only access the server-side which consists of Arx’s server proxy and the database servers.

Arx considers *passive* (honest-but-curious) server attackers: the attackers examine server-side data to glean sensitive information, but follow the protocol as specified, and do not modify the database or query results. The active attacker is interesting future work, that can potentially leverage complementary techniques [43, 54, 57, 88]. Further, in the Arx model, an attacker cannot inject any *new* queries as she does not have access to the client application or to the secret keys at the client proxy, but only to the server.

We consider two types of passive attackers, offline and online attackers, and provide different guarantees for each. The *offline* attacker manages to steal *one* copy of the database, consisting of (encrypted) collections and indices. It does not contain in-memory data related to the execution of current queries (which falls under the online attacker). The *online* attacker is a generic passive attacker: it can log and observe any information available at the server (*i.e.*, all changes to the database, all in-memory state, and all queries) at any point in time for any amount of time.

## 2.3 Security guarantees

**Offline attacker.** Arx’s most visible contribution over PPE-based EDBs is for the offline attacker. Such an attacker corresponds to a wide range of real-world instances including hackers who extract a dump of the database, or insiders who managed to steal a copy of the database.

For this attacker, Arx provides strong security guarantees revealing nothing about the data beyond the schema and size information (the number of collections, documents, items per field, size of items, which fields have indices and for what operations). The contents of the database (collections and indices) are protected with *semantically secure* encryption, and the decryption key is never sent to the server. In contrast,

in PPE-based EDBs, the attacker readily sees the order or frequency of all the values in the database for PPE-encrypted fields. In particular, Arx prevents the offline attacks of [26, 37, 61] from which PPE-based EDBs suffer.

**Online attacker.** The online attacker additionally watches *queries* and their execution. Arx hides the parameters in the queries, but not the operations performed. In particular, Arx does not hide metadata (*e.g.*, query identifiers and timestamps) or access patterns during execution (*e.g.*, which positions in the database or index are accessed/returned and how many). Prior work has shown that, if an attacker can observe such information from many queries and if certain assumptions and conditions hold, the attacker can reconstruct data [15, 35, 44, 48]. Since each query in Arx reveals only a limited amount of metadata, the sooner an attacker is detected (*e.g.*, the fewer queries they observe), the less information they are able to glean.

For this attacker, Arx aims to always be more or as secure as PPE-based EDBs. Indeed, for all operations, Arx’s leakage is always upper-bounded by the leakage in PPE-based EDBs. This is non trivial: for example, a prior EDB aiming for semantic security [28] is not always more secure than PPE-based EDBs, as we explain in §11.

**Security definition.** To quantify the leakage to online attackers, we provide a security definition for Arx and its protocols that is parameterized by a *leakage profile*  $\mathcal{L}$ , which is a function of the database and the sequence of the queries issued by the client. Our security definition is fairly standard, and similar to prior work [17, 28].

We say that a DB system (or a query execution protocol) is  $\mathcal{L}$ -semantically secure if, for any PPT adversary  $\mathcal{A}$ , the entirety of  $\mathcal{A}$ ’s view of the execution of the queries is efficiently *simulatable* given only  $\mathcal{L}$ .  $\mathcal{A}$  invokes the interface exposed by the client to submit any sequence of queries  $Q$ .  $\mathcal{A}$  then observes the execution of the queries from the perspective of the server, *i.e.*, it can observe all the state at the server, as well as the full transcript of communication between the client and server. Formally,  $\mathcal{A}$ ’s task is to distinguish between a real world execution of the queries (**Real**) between the client and server, and an ideal world execution (**Ideal**) where the transcript is generated by a PPT simulator  $S$  that is only given access to the leakage function  $\mathcal{L}$ .

**DEFINITION 1.** *Let  $\mathcal{L}$  be a leakage function. We say that a protocol  $\Pi$  is  $\mathcal{L}$ -semantically-secure if for all adversaries  $\mathcal{A}$  and for all sequences of queries  $Q$ , there exists a PPT simulator  $S$  such that:*

$$\Pr[\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda, Q) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A}, S, \mathcal{L}}^{\Pi}(\lambda, Q) = 1] \leq \text{negl}(\lambda)$$

where  $\lambda$  is the security parameter and  $\text{negl}(\lambda)$  a negligible function in  $\lambda$ .

We formalize the leakage profile of Arx and its protocols in §8, and provide proofs of security with respect to Definition 1 for non-adaptive adversaries (who select the query sequence beforehand) in the full version of our paper. Informally, the leakage for ArxEq includes the list of queries that search for the same keyword; for ArxRange, the leakage includes the ranks of the bounds in the range query.

## 2.4 Admin API

We describe the API exposed by Arx to application admins. The admin can take an existing application and enhance it with Arx annotations. Arx’s planner, located at the client proxy, uses this API to decide the data encryption plan, the



list of Arx indices to build, and a query execution plan for each query pattern.

Following the example of Microsoft’s SQL Server [58] and Google’s Encrypted BigQuery [33], Arx requires the admin to declare what operations will run on the database fields. By default, Arx considers *all* the fields in the database to be sensitive, unless specified otherwise. To use Arx, the admin specifies the following information during system setup:

1. (Optional) Annotated schema: fields that are unique, fields that are nonsensitive (if any), and field sizes;
2. The operations that run on sensitive fields;
3. The fields that should be indexed.

For the first, the admin uses the API:  $collection = \{ field_1: info_1, \dots, field_n: info_n \}$ , to annotate the fields in a collection. This annotation is optional, but it benefits the performance of Arx if provided. *info* should specify “unique” if the values in the field are unique, *e.g.*, SSN. Arx automatically infers primary keys to be unique. *info* may also specify a maximum length for the field, which helps Arx choose a more effective encryption scheme.

Arx encrypts all the fields in the DB by default. However, the admin may explicitly override this behavior by specifying *info* as “nonsensitive” for a particular field. This option should only be used if (1) the admin thinks this field is not sensitive and desires to reduce encryption overhead, or (2) Arx does not support the computation on this field but the admin still wants to use Arx for the rest of the fields. However, we caution that though some fields may not be sensitive themselves, they may leak auxiliary information about other fields in the database. Hence, the admin should select such fields with care.

Second, Arx needs to know the query patterns that will run on the database. Concretely, Arx needs to know what operations run on which fields, though not the constants that will be queried—*e.g.*, for the query `select * from T where age = 10`, Arx needs to know there will be an equality check on `age`. The admin can either specify these operations directly, or provide a trace from a run of the application and Arx will automatically identify them.

Third, Arx needs to know the list of regular indices built by the application. Arx needs this information in order to provide the same asymptotic performance guarantees as an unencrypted database. Note that this requirement poses no extra work on the part of the admin, and is the same as required by a regular database.

## 2.5 Functionality

We now describe the classes of read and write queries that Arx can execute over encrypted data. As we show in §9, this functionality suffices for a wide range of applications.

**Read queries.** Arx supports queries of the form:

```
select [agg doc] fields from collection
where clause [orderby fields] [limit ℓ]
```

*doc* denotes a document and `[agg doc]` aggregations over documents, which take the form  $\sum Func(doc)$ .  $\sum$  can be any associative operator and *Func* an arbitrary, efficiently-computable function. Examples include sum, count, sum of squares, min, and max. More aggregations can be computed with minimal postprocessing at the client proxy by combining a few aggregations, such as average or standard deviation. The predicate *clause* is  $[\wedge_i op(f_i)]$  where  $op(f_i)$  denotes equality/range operations over a field  $f_i$  such as  $=, \geq$  and  $<$ .

In addition, Arx supports a common form of joins—namely, foreign-key joins—which we defer to the full version of our paper due to space constraints.

**Write queries.** Arx supports standard write queries such as inserts, deletes, and updates.

**Constraints.** Not all range/order queries are supported by Arx. First, queries may not contain range operations over more than one encrypted field—*i.e.*,  $(5 \geq f_1 \geq 3) \wedge (f_2 \leq 10)$  is not supported unless  $f_2$  is unencrypted. Second, if the query contains a limit along with range operations over an encrypted field, then it may contain an order-by operation over the encrypted field *alone*.

## 3. ENCRYPTION BUILDING BLOCKS

Besides its indices, Arx relies on three semantically-secure encryption schemes. These schemes already exist in the literature, so we do not elaborate on them.

**BASE** is standard probabilistic encryption, *e.g.*, AES-CTR.

**EQ** enables equality checks using a searchable encryption scheme similar to existing work [16, 49]. One cannot build an index on EQ ciphertexts directly because they are randomized, and Arx uses this scheme only for non-indexed fields (*i.e.*, linear scans). When the developer desires an index for equality checks, Arx uses our new ArxEq index.

EQunique is a special case of EQ. In many applications, some fields have unique values, *e.g.*, primary keys, SSN. In this case, Arx makes an optimization: instead of implementing EQ with the scheme above, it uses deterministic encryption. Deterministic encryption does *not* leak frequency when values are unique. Such a scheme is very fast: the server can simply use the equality operator as if the data were unencrypted. Databases can also build indices on the field as before, so this case is an optimization for ArxEq too.

**AGG** enables addition using the Paillier scheme [66].

## 4. ArxRange & ORDER-BASED QUERIES

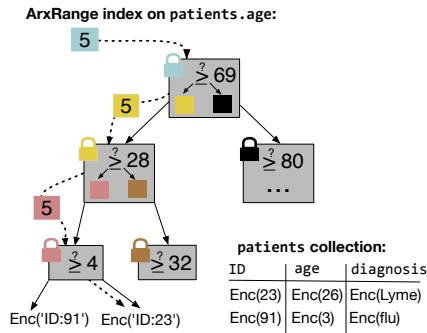
We now present our index enabling range queries and order-by-limit operations.

### 4.1 Strawman

We begin by presenting a helpful but inefficient strawman, that corresponds to the protocols in mOPE [71] and the startup ZeroDB [27]. For simplicity, consider the index to be a binary tree (instead of a regular B+ tree). To obtain the desired security, each node in the tree is encrypted using a standard encryption scheme. Because such encryption is not functional, the server needs the help of the client to traverse the index. To locate a value  $a$  in the index, the server and the client interact: the server provides the root node to the client, the client decrypts it into a value  $v$ , compares  $v$  to  $a$ , and tells the server whether to go left or right. The server then provides the relevant child to the client, and the procedure repeats until it reaches a leaf. As a result, each level in the tree requires a roundtrip, making the process inefficient.

### 4.2 Non-interactive index traversal

ArxRange enables the server to traverse the tree by itself. Say the server receives  $BASE_k(a)$  and must locate the leaf node corresponding to  $a$ . To achieve this goal, the server must be able to compare  $BASE_k(a)$  with the encrypted value at a node, say  $BASE_k(v)$ . Inspired from the theoretical literature



**Figure 2:** ArxRange example. Enc is encryption with BASE.

on garbled RAM [29, 30], we store a **garbled circuit at each tree node** that performs the comparison, while hiding  $a$  and  $v$  from the attacker.

A garbling scheme is a set of algorithms (Garble, Encode, Eval) [31, 86]. Using a garbling scheme, the client can invoke the algorithm **Garble** on a boolean circuit  $f$  to obtain a *garbled* version  $F$  of the circuit, along with some secret encoding information  $e$ . Given an input  $a$ , the client can run **Encode**( $e, a$ ) to produce an encoding  $e_a$  corresponding to the input. Then, the server can run **Eval**( $F, e_a$ ) and obtain the output  $y = f(a)$ . The security of garbled circuits guarantees that the server learns *nothing* about  $a$  or the data hardcoded in  $f$  other than the output  $f(a)$  (and the size of  $a$  and  $f$ ). This guarantee holds as long as the garbled circuit is used *only once*. That is, if the client provides two encodings  $e_a$  and  $e_b$  using the same encoding information  $e$  to the server, the security guarantees no longer hold. Hence, our client provides at most one input encoding for each garbled circuit.

To allow the server to traverse the index without interaction, each node in the index must *re-encode* the input for the next node in the path, because the encoding  $e_a$  of the input to a node differs from the encoding for its children. We therefore **chain the garbled circuits** so that each circuit outputs an encoding compatible with the relevant child node.

Let  $N$  be a node in the index with value  $v$ , and let  $L$  and  $R$  be the left and right nodes. Let  $e^N$ ,  $e^L$ , and  $e^R$  be the encoding information for these nodes. The garbled circuit at  $N$  is a garbling of a boolean circuit that compares the input with the hardcoded value  $v$  and additionally outputs the re-encoded input labels for the next circuit:

```

if  $a < v$  then
   $e'_a \leftarrow \text{Encode}(e^L, a)$ ; output  $e'_a$  and 'left'
else
   $e'_a \leftarrow \text{Encode}(e^R, a)$ ; output  $e'_a$  and 'right'

```

Fig. 2 shows how **the server traverses the index without interaction**. The number at each node indicates the value  $v$  hardcoded in the relevant garbled circuit. Now consider the query: `select * from patients where age < 5`. The client provides an encoding of 5,  $\text{Encode}(5)$  encrypted with the key for the root garbled circuit. The server runs this garbled circuit on the encoding and obtains “left” as well as an encoding of 5 for the left garbled circuit. The server then runs the left circuit on the new encoding, and proceeds similarly until it reaches the desired leaf node. Note that since each node encodes the  $<$  operation, in order to perform  $\leq$  operations the client needs to first transform the query into an equivalent query with the  $<$  operation; *e.g.*, `age  $\leq$  5` is transformed to `age < 6` instead.

**Repairing the index.** A part of our index gets destroyed during the traversal because each garbled circuit may be used at most once. To repair the index, the client needs to supply new garbled circuits to replace the circuits consumed. Fortunately, only a logarithmic number of garbled circuits get consumed. Suppose a node  $N$  and its left child  $L$  get consumed. For each such node  $N$ , the client needs two pieces of information from the server: the value  $v$  encoded in  $N$ , and the encoding information for the right child  $R$ . The server therefore sends an encryption of  $v$  (*i.e.*,  $\text{BASE}(v)$ , stored separated in the index), and the ID of the circuit at  $R$ . The ID of each circuit is a unique, random value that is used by the client proxy (together with the secret key) to generate the encodings for the circuit; *i.e.*, the ID of the circuit at  $R$  was used to compute  $e^R$ . Sending ID instead of  $e^R$  saves bandwidth because the encoding information is not small (1KB for a 32-bit comparison).

### 4.3 The database index

We need to take two more steps to obtain an index with the desired security.

First, the shape of the index should not leak information about the order in which the data was inserted. Hence, we use a history-independent treap [4, 59] instead of a regular search tree. This data structure has the property that its shape is independent of the insertion or deletion order.

Second, we store at each node in the tree the encrypted primary key of the document containing the value. This enables locating the documents of interest. Note that the index *does not leak the order of values in the database* even though the leaves are ordered: the mapping between a leaf and a document is encrypted, and the index can be simulated from size information. If the primary key were not encrypted, the server would learn such an order.

**Query execution.** Consider the query `select * from patients where 1 < age  $\leq$  5`. Each node in the index has two garbled circuits to allow concurrent search for the lower and upper bounds. The client proxy provides tokens for values 1 and 5 to the server, which then locates the leftmost and rightmost leaves in the interval (1, 5] and fetches the encrypted primary keys from all nodes in between. The server sends the encrypted keys to the client proxy which decrypts them, shuffles them, and then selects the documents mapped to these primary keys from the server. The shuffling *hides from the server the order of the documents in the range*.

For order-by-limit  $\ell$  queries, the server simply returns the leftmost or rightmost  $\ell$  nodes. Order-by operations without a limit are not performed using ArxRange. Since they do not have a limit, they do not do any filtering, so the client proxy can simply sort the result set itself.

**Updating the index.** For inserts and deletes, the server traverses the index to the appropriate position, performs the operation, and rebalances the index if required. For updates, the server first performs a delete followed by an insert. As a result of the rebalancing, all nodes that have at least one different child node are also marked as consumed (in addition to those consumed during traversal), and are sent for repair to the client proxy; however, the total number of consumed nodes is always upper bounded by the height of the index.

Some update or delete queries may first perform a filter on a field using a different index, but also requiring deletes from an ArxRange index as a result. To support this case, we maintain an encrypted *backward* pointer from the document

to the corresponding node in the tree. The backward pointers enable the identification of these nodes *without* having to traverse the ArxRange index. Decryption of these pointers requires a single round of interaction with the client proxy.

Additionally, for *monotonic inserts*—a common case where inserts are made in increasing or decreasing order—a cheap optimization is for the client proxy to remember the position in the tree of the last value, so that most values can be inserted directly without requiring traversal and repair.

**Concurrency.** ArxRange provides limited concurrency because each index node needs to be repaired before it can be used again. To enable a degree of concurrency, the client proxy stores the top few levels of the tree. As a result, the index at the server essentially becomes a forest of trees and accesses across different trees can be performed in parallel. At the same time, the storage at the client proxy is very small because trees grow exponentially in size with the number of levels. Queries to the same subtree, however, are still sequential. This technique improves performance without impacting the security guarantees of the index.

## 4.4 Optimizations

We employ several techniques to further improve the performance of ArxRange: (1) we chain garbled circuits together using *transition tables* instead of computing the encoding function inside the circuit; (2) we incorporate recent advances in garbling in order to make our circuits short and fast; and (3) we remove index repair from the critical path of a query, and return the query results to the client before starting repair. We defer the details to the full version of our paper.

## 5. ArxEq & EQUALITY QUERIES

The ArxEq index enables equality queries and builds on insights from the searchable encryption literature [11], as explained in §11. We aim for ArxEq to be *forward private*, a property shown to increase security significantly in this context [12]: the server cannot use an old search token on newly inserted data. We begin by presenting a base protocol that we improve in stages.

### 5.1 Base protocol

Consider an index on the field `age`. ArxEq will encrypt the value in `age` (as follows) and it will then tell the DB server to build a *regular index* on `age`.

The case when the fields are unique (*e.g.*, primary key, IDs, SSNs) is simple and fast: ArxEq encrypts the fields with EQunique and the regular index suffices. The rest of the discussion applies to non-unique fields.

The client proxy stores a map, called `ctr`, mapping each distinct value  $v$  of `age` that exists in the database to a counter indicating the number of times  $v$  appears in the database. For `age`, this map has about 100 entries.

**Encrypt and insert.** Suppose the application inserts a document where the field `age` has value  $v$ . The client proxy first increments `ctr[v]`. Then, it encrypts  $v$  into:

$$\text{Enc}(v) = H(\text{EQunique}(v), \text{ctr}[v]) \quad (1)$$

where  $H$  is a cryptographic hash (modeled as a random oracle). This encryption provides semantic security because EQunique( $v$ ) is a deterministic encryption scheme which becomes randomized when combined with a unique salt per value  $v$ : `ctr[v]`. This encryption is not decryptable, but as

discussed in §7.2, Arx encrypts  $v$  with BASE as well. The document is then inserted into the database.

**Search token.** When the application sends the query `select * where age = 80`, the client proxy computes a search token using which the server proxy can search for all occurrences of the value 80. The search token for a value  $v$  is the list of encryptions from Eq. (1) for every counter from 1 to `ctr[v]`:  $H(\text{EQunique}(v), 1), \dots, H(\text{EQunique}(v), \text{ctr}[v])$ .

**Search.** The server proxy uses the search token to reconstruct the query’s `where` clause as: `age = H(EQunique( $v$ ), 1) or ... or age = H(EQunique( $v$ ), ctr[v])` (with the clauses in a random order). The DB server uses the regular index on `age` for each clause in this query and returns the results. If the number of values exceeds the maximum query size allowed by the backend database, then Arx’s server proxy split the disjunction into multiple queries (at the cost of additional index lookups).

Note that the scheme provides forward privacy: the server cannot use an old search token to learn if newly inserted values are equal to  $v$  as they would have a higher counter.

### 5.2 Reducing the work of the client proxy

The protocol so far requires the client proxy to generate as many tokens as there are equality matches on the field `age`. If a query filters on additional fields, the client proxy does more work than the size of the query result, which we want to avoid whenever possible. We now show how the client proxy can work in time  $(\log \text{ctr}[v])$  instead of `ctr[v]`.

Instead of encrypting a value  $v$  as in Eq. (1), the client proxy hashes according to the tree in Fig. 3. It starts with EQunique <sub>$k$</sub> ( $v$ ) at the root of a binary tree. A left child node contains the hash of the parent concatenated with 0, and a right child contains the hash of the parent with 1. The leaves of the tree correspond to counters 0, 1, 2, 3, ..., `ctr[v]`.

The client proxy does not materialize this entire tree. Given a counter value `ct`, the proxy can compute the leaf corresponding to `ct`, simply by using the binary representation of `ct` to compute the corresponding hashes.

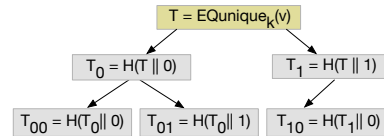


Figure 3: Search token tree.

**New search token.** To search for a value  $v$  with counter `ctr[v]`, the client proxy computes the *covering set* for leaf nodes  $0, \dots, \text{ctr}[v] - 1$ . The covering set is the set of internal tree nodes whose subtrees cover exactly the leaf nodes  $0, \dots, \text{ctr}[v] - 1$ . E.g., in Fig. 3, `ctr[v] = 3` and the covering set of the three leaves is  $\{T_0, T_{10}\}$ . The nodes in the covering set constitute the search token. The covering set can be easily deduced from the binary representation of `ctr[v] - 1`.

**Search.** The server proxy expands the covering set into the leaf nodes, and proceeds as before.

### 5.3 Updates

We have already discussed inserts. For deletes, Arx simply deletes the document. An update is a delete followed by an insert. As a result, encrypted values for some counters will not return matches during search. This does not affect accuracy, but as more counters go missing, it affects throughput



because the DB server wastes cycles looking for values with no matches. It also provides a small security leakage because a future search leaks how many items were deleted. As a result, **ArxEq** runs a cleanup procedure after each deletion. As a performance optimization, one can run a cleanup procedure when a search query for a value  $v$  indicates more than a threshold of missing counters, relaxing security slightly.

**Cleanup.** The server proxy tells the client proxy how many matches were found for a search, say  $ct$ . The client proxy updates  $ctr[v]$  with  $ct$ , chooses a new key  $k'$  for  $v$ , and generates new tokens as in Fig. 3:  $T'_{00}, \dots, T'_{ct}$  using  $k'$ . It gives these tokens to the server, which replaces the fields found matching with these.

## 5.4 Counter map

To alleviate the burden of storing the counter map at the client proxy, it is possible to store it encrypted at the server instead while still providing strong guarantees against offline attackers. However, we recommend storing it at the client proxy for increased security against the online attacker. We now discuss both design points and accompanying tradeoffs.

**Counter map at server.** The counter map can be stored encrypted at the server. An entry of the sort  $v \rightarrow ct$  becomes  $\text{EQunique}_{k_1^*}(v) \rightarrow \text{EQunique}_{k_2^*}(ct)$ , where  $k_1^*$  and  $k_2^*$  are two keys derived from the master key, used for the counter map. When encrypting a value in a document or searching for a value  $v$ , the client proxy first fetches the encrypted counter from the server by providing  $\text{EQunique}_{k_1^*}(v)$  to the server. Then, the algorithm proceeds the same as above.

To avoid leaking the number of distinct fields, **Arx** pads the counter map to the number of documents in the relevant collection. This scheme satisfies **Arx**'s security goal in §2.3: a stolen database remains encrypted with semantic security and leaks nothing except size information.

**Counter map at client.** However, we recommend keeping the counter map at the client proxy for higher security against an online attacker. If the counter map is stored at the server, then with every newly inserted value, an online attacker can see which entry of the counter map is accessed and which document is inserted in the database. Storing the counter map at the client hides such correlations entirely.

Though the size of the counter map grows with the number of different values a field can take, in many cases, the storage overhead is small—*e.g.*, for low-cardinality fields such as gender, age, and letter grades. Moreover, in the extreme case when all values are unique (*i.e.*, the maximum possible size for a counter map), **ArxEq** defaults to the regular index built over **EQunique** encryptions, which doesn't need a counter map at all. The case when there are many distinct values with few repetitions is less ideal, and we implement an optimization for this case: to decrease the size of the counter map, **Arx** groups multiple entries into one entry by storing their prefixes. As a tradeoff, the client proxy has to filter out some results.

## 6. ArxAgg & AGGREGATION QUERIES

We now explain **Arx**'s aggregation over the encrypted indices. It is based on AES and is faster than homomorphic encryption schemes like Paillier [66]. Many aggregations happen over a range query, such as computing the average days in hospital for people in a certain age group. **Arx** computes the average by computing sum and count at the server, and then dividing them at the client proxy. Hence,

let's focus on the query: `select sum(daysAdmitted) from patients where 70 ≤ age ≤ 80.`

The idea behind aggregations in **Arx** is inspired from literature on authenticated data structures [54]. This work targets integrity guarantees (not confidentiality), but interestingly, we use it for computations on encrypted data. Consider the **ArxRange** index in Fig. 2 built on **age**. At every node  $N$  in the tree, we add the *partial aggregate* corresponding to the subtree of  $N$ . For the query above,  $N$  contains a partial sum of **daysAdmitted** corresponding to the leaves under  $N$ . The root node thus contains the sum of all values. This value is stored encrypted with **BASE**.

To compute the sum over an arbitrary range such as [70, 80], the server first locates the edges of the range as before, and then identifies a *perfect covering set*. Note that the covering set is *logarithmic* in the size of the index. For each node in this set, the server returns the encrypted aggregates of all its children and the encrypted value of the node itself to the client proxy, which decrypts them and sums them up.

## 7. ARX'S PLANNER

**Arx**'s planner takes as input a set of query patterns, **Arx**-specific annotations, and a list of regular indices (per §2.4), and produces a data encryption plan, a list of **Arx**-style indices, and a query plan for each pattern.

### 7.1 Index planning

Before deciding what index to build, note that **ArxRange** and **ArxEq** support *compound* indices, which are indices on multiple fields. For example, an index on (**diagnosis**, **age**) enables a quick search for **diagnosis** = 'flu' and **age** ≥ 10. **Arx** enables these by simply treating the two fields as one field alone. For example, when inserting a document with **diagnosis**= 'flu', **age** = 10, **Arx** merges the fields into one field 'flu' || 00010, prefixing each value appropriately to maintain the equality and order relations, and then builds a regular **Arx** index.

When deciding what indices to build, we aim to provide the *same asymptotic performance* as the application admin expects: if she specified an index over certain fields, then the time to execute queries on those fields should be logarithmic and not require a linear scan. At the same time, we would like to build few indices to avoid the overhead of maintaining and storing them. Deciding what indices to build automatically is challenging because (1) there is no direct mapping from regular indices to **Arx**'s indices, and (2) **Arx**'s indices introduce various constraints, such as:

- A regular index serves for both range and equality operations. This is not true in **Arx**, where we have two different indices for each operation. We choose not to use an **ArxRange** index for equality operations because of its higher cost and different security.
- Unlike a regular index, a compound **ArxEq** index on  $(a, b)$  cannot be used to compute equality on  $a$  alone because **ArxEq** performs a complete match.
- A range or order-by-limit on a sensitive field can be computed only via an **ArxRange** index, so it can no longer be computed after applying a separate index.

All these are further complicated by the fact that the application admin can explicitly specify certain fields to be nonsensitive (as described in §2.4), and simultaneously declare compound indices on a mixture of fields, both sensitive

and not. Similarly, queries can have both sensitive as well as nonsensitive fields in a **where** clause.

Despite all these constraints, our index planning algorithm is quite simple. It runs in two stages: per-query processing and global analysis. Only the **where** clauses (including order-by-limit operations) matter here. The first stage of the planner treats sensitive and nonsensitive fields equally. For clarity, we use two query patterns as examples. Their **where** clauses are:  $W_1$ : “ $a = \text{and } b =$ ”,  $W_2$ : “ $x = \text{and } y \geq \text{and } z =$ ”. The indices specified by the admin are on  $x$  and  $(a, b)$ .

**Stage 1: Per-query processing.** For each **where** clause  $W_i$ , extract the set of filters  $S_i$  that can use the indices in a regular database. Example: For  $W_1$ ,  $S_1 = \{(a =, b =)\}$  and for  $W_2$ ,  $S_2 = \{(x =)\}$ .

Then, if  $W_i$  contains a sensitive field with a range or order-by-limit operation, append a “ $\geq$ ” filter on this field to each member of  $S_i$ , if the member does not already contain this. Based on the constraints in §2.5, a where clause cannot have more than one such field. Example: For  $W_1$ ,  $S_1 = \{(a =, b =)\}$ , and for  $W_2$ ,  $S_2 = \{(x =, y \geq)\}$ .

**Stage 2: Global analysis.** Union all sets  $S = \cup_i S_i$ . Remove any member  $A \in S$  if there exists a member  $B \in S$  such that an index on  $B$  implies an index on  $A$ . The concrete conditions for this implication depend on whether the fields involved are sensitive or not, as we now exemplify.

Example: If  $a$  and  $b$  are nonsensitive, and  $S$  contains both  $(a =, b =)$  and  $(a =, b \geq)$ , then  $(a =, b =)$  is removed. If all of  $a$ ,  $b$  and  $c$  are sensitive and  $S$  contains both  $(a =, b =, c \geq)$  and  $(a =, b \geq)$ , then  $(a =, b \geq)$  is removed. For  $W_1$  and  $W_2$  above, if  $b$  and  $y$  are sensitive ( $a, x, z$  can be either way), the indices Arx builds are: **ArxEq**  $(a, b)$  and **ArxRange**  $(x, y)$ .

One can see why our planner maintains the asymptotic performance of the admin’s index specification: each expression that was sped up by an index remains sped up. In §9, we show that the number of extra indices Arx builds is modest and does not blow up in practice.

## 7.2 Data layout

Next, laying out the encryption plan is straightforward:

- All values of a sensitive field are encrypted with the same key, but this key is different from field to field.
- For every aggregation in a query, decide if the **where** clause in this query can be supported entirely by using **ArxRange** or **ArxEq**. Concretely, the **where** clause should not filter by additional fields not present in the index. If so, update the metadata of the respective index to follow our aggregation strategy per §6. If not, encrypt the respective fields with AGG if the aggregate requires the computation of a sum.
- For every query pattern, if the **where** clause  $W_i$  checks equality on a field  $f$  that is not part of every element of  $S_i$ , encrypt  $f$  with EQ (since at least one query plan will need to filter this field by equality without an index).
- For every sensitive field projected by at least one query, additionally encrypt it with BASE. The reason is that EQ and our indices are not decryptable.

## 8. SECURITY ANALYSIS

We now formalize the security guarantees of Arx. We first develop a formal model of a database system, and then provide leakage definitions with respect to offline and online attackers. Proofs of security follow in the full version.

| Protocol ( $\epsilon$ ) | Operation  | $\mathcal{L}_\epsilon(\text{DB}, \mathbf{q})$                         |
|-------------------------|--|---|
| <b>EQ</b><br>(§3)       | <b>where</b> $field = w$<br><b>insert</b><br><b>delete</b>                                     | $\text{sp}(w), \text{Hist}(w)$<br>$\text{sp}(w)$<br>–                 |
| <b>ArxEq</b><br>(§5)    | <b>where</b> $field = w$<br><b>cleanup</b> (§5.3)<br><b>insert, delete</b>                     | $\text{sp}(w), \text{Hist}(w)$<br>$\text{sp}(w), \text{Hist}(w)$<br>– |
| <b>ArxRange</b><br>(§4) | <b>where</b> $a \leq field \leq b$<br><b>orderby limit</b> $\ell$<br><b>insert, delete</b> $v$ | $\text{rk}(a - 1), \text{rk}(b)$<br>$\ell$<br>$\text{rk}(v)$          |

Figure 4: Query leakage in Arx’s protocols.

**Notation.** We denote the set of all binary strings of length  $n$  as  $\{0, 1\}^n$ . We write  $[a_i]_{i=1}^n$  to denote the list of values  $[a_1, \dots, a_n]$ . If  $S$  is a list or a set, then  $|S|$  denotes its size.

## 8.1 Preliminaries

A *database system* is a pair of stateful random access machines (Client, Server). The server Server stores a *database* DB, and the client Client can through interaction with Server compute *queries* out of a set of supported queries, which may modify the database.

**Database.** A *database*  $\text{DB} = \{\text{T}_1, \dots, \text{T}_n\}$  is a set of *collections*. Each collection  $\text{T}_i = (\text{F}_i, \text{Ind}_i, [(\text{id}_j, \text{D}_j)]_j)$  comprises a set of *fields*  $\text{F}_i = \{f_1, \dots, f_{m_i}\}$  of size  $m_i$ ; a set of *indices*  $\text{Ind}_i$ ; and a list of identifier-document pairs, where  $\text{id}_j$  is the *identifier* for document  $\text{D}_j$ . A *document*  $\text{D}_j = [w_1, \dots, w_{m_i}]$  is a list of keywords where  $w_i$  is indexed by field  $f_i$  (denoted  $w_i = \text{D}_j[f_i]$ ). Here,  $w_i \in \{0, 1\}^{\|f_i\|} \cup \{\phi\}$ , where  $\|f_i\|$  denotes the size of the keywords in the field’s domain. Also, we write  $\|\text{T}\|$  to denote the number of documents in collection  $\text{T}$ .

Given a collection  $\text{T}$ , we write  $\text{T}(w)$  to denote the set of identifiers of documents that contain  $w$ , *i.e.*,  $\text{T}(w) = \{\text{id} \mid \exists (\text{id}, \text{D}) \in \text{T} \text{ s.t. } w \in \text{D}\}$ .

**Indices.** Given a collection  $\text{T}_i$  and a field  $f$ , an index  $I \in \text{Ind}_i$  is a search tree built over the  $\text{D}_j[f]$ , for all  $\text{D}_j \in \text{T}_i$ . We represent the search tree as a tuple  $(V, E)$  of nodes and edges, where each node contains a function  $f$  that enables tree traversal. We define the shape of an index  $\text{shape}(I) = E$  to be the set of edges. Since a field may contain multiple indices (*i.e.*, both **ArxEq** and **ArxRange**), we write  $I(f)$  to refer to all the indices maintained on a field  $f$ .

**Schema.** Let  $\mathbb{E} = \{\text{BASE}, \text{EQ}, \text{EQunique}, \text{ArxEq}, \text{ArxRange}\}$  denote the set of protocols supported by Arx. We define the *schema*  $\mathbb{S}$  of a collection  $\text{T}_i$  to be its name, its set of fields, the size of each field, protocols maintained per field, and the shapes of all indices:

$\mathbb{S}(\text{T}_i) = (i, \{f_j, \|f_j\|, \text{plan}(f_j), \text{shape}(I) \forall I \in I(f_j)\}_{j=1}^{m_i})$ . Here,  $\text{plan}(f_j) = \{\mathbf{e} \in \mathbb{E}\}$  is the set of protocols maintained on field  $f_j$ . The schema of a database  $\text{DB}$  is then given by  $\mathbb{S}(\text{DB}) = \cup_i \mathbb{S}(\text{T}_i)$ .

**Queries.** A *predicate*  $\text{pred} = (f, \text{op})$  is a tuple comprising a field  $f$  and an operation  $\text{op}$  over the field, where  $\text{op} \in \{\leq, \geq, =, <, > \text{ and } \geq, \text{orderby limit}\}$ . A *query*  $\mathbf{q} = (\text{ts}, \text{T}, \text{qtype}, \text{pred}, \text{params})$  is a 5-tuple that comprises a timestamp  $\text{ts}$ , the name of a collection  $\text{T}$ , a query type  $\text{qtype} \in \{\text{read}, \text{insert}, \text{delete}\}$ , a predicate  $\text{pred}$ , and query parameters  $\text{params}$  corresponding to the predicate. We model updates as a **delete** followed by an **insert**.

Note that our definition of a query consists only of a single predicate for simplicity of exposition. We model queries with multiple predicates as a list of single-predicate queries. To insert a document  $\text{D}$ , we model the operations as a list



of `insert` queries, one per field of the document; in these `insert` queries,  $q.\text{pred.op} = \perp$ ,  $q.\text{pred.f}$  is the corresponding field, and  $q.\text{params}$  is the value to be inserted.

We write  $\text{DB}(q) = ([\text{id}_i]_i, e)$  to denote the set of identifiers of documents that satisfy  $q$  along with the protocol  $e \in \mathbb{E}$  used to execute  $q$ . For inserts, deletes, and updates,  $[\text{id}_i]_i$  indicates the list of documents inserted, deleted, or updated.

**Admin API.** During system setup, for each collection  $T_i$ , the admin supplies a *predicate set*:

$$\mathbb{P}(T_i) = \{[(f_j, \text{op}_j)]_j \mid f_j \in F_i\},$$

which is the set of query predicates that will be issued by Client over the collection (as described in §2.4). The *global predicate set* is then given by  $\mathbb{P}(\text{DB}) = \bigcup_i \mathbb{P}(T_i)$ .

## 8.2 Leakage definitions

We define the leakage profile of Arx,  $\mathcal{L} = \{\mathcal{L}_{\text{off}}, \mathcal{L}_{\text{on}}\}$ : first for the database itself (offline attacker), then for the execution of each query (online attacker).

DEFINITION 2 (OFFLINE LEAKAGE OF A DATABASE).

The leakage of a database DB is:

$$\mathcal{L}_{\text{off}}(\text{DB}) = (\mathbb{S}(\text{DB}), \mathbb{P}(\text{DB}), \{\forall T_i \parallel T_i\}),$$

where  $\mathbb{S}(\text{DB})$  is the schema of the database, and  $\mathbb{P}(\text{DB})$  is the global predicate set of the database.

Before defining the leakage of queries, we define the rank of an element  $x$  in a list  $L = [a_1, \dots, a_n]$  as  $\text{rk}(L, x) = |\{a_i \mid a_i \leq x\}|$ , and we write  $\text{rk}(x)$  if  $L$  is clear from context.

Our online leakage function  $\mathcal{L}_{\text{on}}$  is stateful, and maintains the *query history* of the database  $\mathbb{Q}(\text{DB}) = [q_i]_i$  as a list of every query issued by Client. We denote the query history of a collection  $T$  as  $\mathbb{Q}(T) = \{q \mid q \in \mathbb{Q}(\text{DB}) \text{ and } q.T = T\}$ .

Given collection  $T$  with a field  $f$  that has the EQ or ArxEq protocol maintained on it, we define the *search pattern* of a keyword  $w$  (following Bost [12]) as:

$$\text{sp}(w, T, f) = \{q.\text{ts} \mid \exists q \in \mathbb{Q}(T) \text{ s.t. } q.\text{pred} \text{ is } (f, =), \text{ and } w \in q.\text{params}\},$$

and we write  $\text{sp}(w)$  if  $T$  and  $f$  are clear from context. Essentially,  $\text{sp}$  leaks which equality queries relate to the same keyword. Similarly, for collection  $T$  with a field  $f$  containing the EQ or ArxEq protocol, we define the *write history* of keyword  $w$  as:

$$\text{WHist}(w, T, f) = \{(q.\text{ts}, q.\text{qtype}, \text{id}) \mid \exists q \in \mathbb{Q}(T) \text{ s.t. } \text{id} \in \text{DB}(q), q.\text{qtype} \in \{\text{insert}, \text{delete}\}, \text{ and } D[f] = w\},$$

where  $D$  is the document corresponding to  $\text{id}$ . We write  $\text{WHist}(w)$  if  $T$  and  $f$  are clear from context. Essentially,  $\text{WHist}$  leaks the list of all write operations on keyword  $w$ .

Finally, let  $T^0$  be the state of collection  $T$  in the initial database, before any queries are issued. Then, we define the *history* of keyword  $w$  as  $\text{Hist}(w, T) = (T^0(w), \text{WHist}(w, T))$ , and we write  $\text{Hist}(w)$  if  $T$  is clear from context.

DEFINITION 3 (ONLINE LEAKAGE OF QUERIES). Let  $([\text{id}_i]_i, [e_j]_j) \leftarrow \text{DB}(q)$ . Then, the leakage of a query  $q$  over database DB is:

$$\mathcal{L}_{\text{on}}(\text{DB}, q) = ((q.\text{ts}, q.T, q.\text{qtype}, q.\text{pred}), [\text{id}_i]_i, \mathcal{L}_e(\text{DB}, q))$$

where  $\mathcal{L}_e(\text{DB}, q)$  is additional leakage due to the protocol  $e$  used to execute the query, as detailed in Fig. 4.

We note that the leakage of ArxEq, as captured in Fig. 4, is similar to that of Sophos [12] and Diana [13].

## 9. EVALUATION

We now show that Arx supports real applications with a modest overhead. We defer microbenchmarks of the various encryption schemes to the full version of our paper.

**Implementation.** While the design of Arx is decoupled from any particular DBMS, we implemented our prototype for MongoDB 3.0. Arx’s implementation consists of  $\sim 11.5\text{K}$  lines of Java, and  $\sim 1800$  lines of C/C++ code. We also disable query logs and query caches to reduce the chance that an offline attacker gets information an online attacker would see, as discussed in Grubbs *et al.* [36].

**Testbed.** To evaluate the performance of Arx, we used the following setup. Arx’s server proxy was collocated with MongoDB 3.0.11 on 4 cores of a machine with 2.3GHz Intel E5-2670 Haswell-EP processors and 256GB of RAM. Arx’s client proxy was deployed on 4 cores of an identical machine. A separate machine with 48 cores was used to run the clients. In throughput experiments, we ran the clients on all 48 cores of the machine to measure the server at maximum capacity. All three machines were connected over a 1GbE network; to simulate real-world deployments, we added a latency of 10ms (RTT) to each server using the `tc` utility.

## 9.1 Functionality

To understand if Arx supports real applications, we evaluate Arx on seven existing applications built on top of MongoDB. We manually inspected the source code of each application to obtain the list of *unique* queries issued by them, and cross-verified the list against query traces produced during an exhaustive run of the application. All these applications contain sensitive user information, and Arx encrypts all fields in these applications by default.

Fig. 5 summarizes our results. With regard to unsupported queries across the applications, 4 of the 11 were due to timestamps; Arx can support these queries in case the timestamps are nonsensitive and explicitly specified as such by the application admin. The limitation was the number of range/order operations Arx allows in the query, as explained in §2.5. For NodeBB, the two unsupported queries performed text searches, and for Leanote, the five queries were evaluating regular expressions, both of which Arx cannot support. Even so, these are *only a small fraction* of the total queries issued, which are tens to hundreds in number. In general, the table shows that Arx can support almost all the queries issued by real applications. In cases where an application contains queries that are not supported by Arx, the application admin should consider whether the application needs the query in that form and if she can adjust it (*e.g.*, by removing a filter that is not necessary or that can be executed in the application). The admin could also consider if the unsupported data field is nonsensitive and mark it as such, but this should be done with care. The table also shows that though Arx’s planner increases the number of indices by 20%, this number does not blow up. The main reason is that the number of fields with order queries that are not indexed by the application is small.

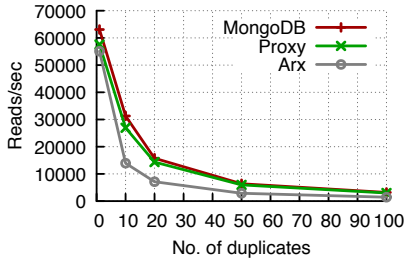
## 9.2 Performance of ArxEq

We evaluate the overall performance of ArxEq (without the optimization for unique values) using relevant queries issued by ShareLaTeX. These queries filter documents by one field using ArxEq. We loaded the database with 100K documents representative of a ShareLaTeX workload.

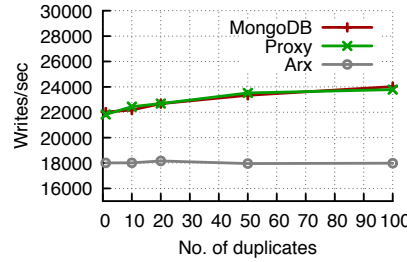
Fig. 6 compares the read throughput of ArxEq with a regular MongoDB index, when varying the number of duplicates per value of the indexed field. The ArxEq scheme expands a query from a single equality clause into a disjunction of

| Application           | Examples of fields    | Unsupported queries |                  | Number of indices |          | Total indices |     |
|-----------------------|-----------------------|---------------------|------------------|-------------------|----------|---------------|-----|
|                       |                       | Total               | Excl. timestamps | ArxEq             | ArxRange | Vanilla       | Arx |
| ShareLaTeX [77]       | document lines, edits | 1                   | –                | 12                | 4        | 12            | 16  |
| Uncap (medical) [84]  | heart rate, tests     | –                   | –                | 0                 | 2        | 2             | 2   |
| NodeBB (forum) [63]   | posts, comments       | 2                   | 2                | 13                | 4        | 12            | 17  |
| Pencilblue (CMS) [69] | articles, comments    | 3                   | –                | 46                | 27       | 70            | 73  |
| Leanote (notes) [51]  | notes, books, tags    | 5                   | 5                | 64                | 28       | 69            | 92  |
| Budget manager [14]   | expenditure, ledgers  | –                   | –                | 5                 | 0        | 5             | 5   |
| Redux (chat) [75]     | messages, groups      | –                   | –                | 3                 | 0        | 3             | 3   |

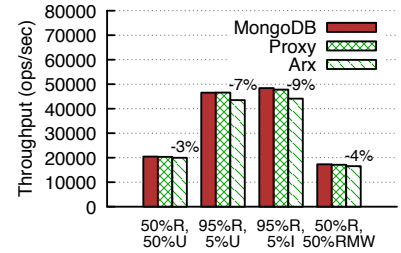
**Figure 5:** Examples of applications supported by Arx: examples of fields in these applications; the number of queries not supported by Arx when all fields were considered sensitive, and when timestamps were excluded; how many Arx-specific indices the application requires; and the total number of indices the database builds in the vanilla application and with Arx. Since ArxAgg is built on top of ArxEq and ArxRange, we do not count it separately.



**Figure 6:** ArxEq read throughput with increasing no. of duplicates.



**Figure 7:** ArxEq write throughput with increasing no. of duplicates.



**Figure 8:** YCSB throughput for different workloads.

equalities over all possible tokens. The number of tokens corresponding to a value increases with the number of duplicates. The DB server essentially looks up each token in the index. In contrast, a regular index maps duplicates to a single reference and can fetch them all in a scan. Both indices need to fetch the documents for each primary key identified as a matching, which constitutes a significant part of the execution time. Overall, ArxEq incurs a penalty of 55% in the worst case, of which 8% is due to Arx’s proxy. When all fields are unique, the added latency due to ArxEq is small—1.13ms versus 0.94ms for MongoDB. As the number of duplicates increases, the latency of both MongoDB and Arx increase as well—at 100 duplicates, Arx’s latency is 42.1ms, while that of MongoDB is 18.8ms.

Fig. 7 compares the write throughput of ArxEq with increasing number of duplicates. The write performance of a regular B+Tree index slowly improves with increased duplication, as a result of a corresponding decrease in the height of the tree. In contrast, writes to an ArxEq index are independent of the number of duplicates by virtue of security: each value looks different. Further, since each individual insert requires the computation of a single token (a constant-time operation), the write throughput of ArxEq remains stable in this experiment. As a result, the net overhead grows from 18% (when fields are unique) to 25% when there are 100 duplicates per value. Latency follows a similar trend and remains stable for ArxEq at 3.3ms. For a regular MongoDB index, the latency slowly improves from 2.7ms to 2.5ms as the number of duplicates grows to 100.

**YCSB Benchmark.** Since Arx is a NoSQL database, we also evaluate its overhead using the YCSB benchmark [22]. YCSB conforms to ArxEq’s optimized case when fields are unique. In this experiment, we loaded the database with 1M documents. Arx considers all fields to be sensitive by default, including the primary key. Hence, the primary key has an ArxEq index and the rest of the fields are encrypted with BASE. Fig. 8 shows the average performance of Arx

versus vanilla MongoDB, across four different workloads with varying proportions of reads and writes, as specified. “R” refers to proportion of reads, “U” to updates, “I” to inserts, and “RMW” to read-modify-write. The reduction in throughput is higher for read-heavy workloads as a result of the added latency due to sequential decryption of the result sets. Overall, the overhead of Arx is 3%-9% across workloads, showing that indexing primary keys is fast with Arx.

### 9.3 Performance of ArxRange

We now evaluate the latency introduced by ArxRange. We pre-inserted 1M values into the index, and assumed a length of 128 bits for the index keys, which is sufficient for composite keys. We cached the top 1000 nodes of the index at the client proxy, which amounted to a mere 88KB of memory. We subsequently evaluated the performance of read and write operations on the index. Fig. 9 illustrates the latency of each operation, divided into three parts: (1) the time taken to traverse the index, (2) the time taken to decrypt the retrieved document IDs (for reads)—this incurs a network roundtrip as described in §4.3; (3) the time taken to retrieve the corresponding results (for reads) or insert the document (for writes), and (4) the time taken to repair the index. The generation of fresh garbled circuits in order to repair the index contributes the most towards latency.

Overall, range queries cost more than writes because the former require a network roundtrip in order to decrypt the retrieved IDs before fetching the corresponding documents. The cost of traversing a path in the index is ~3ms. We note that the strawman in §4.1 incurs a roundtrip overhead for each node in the path, while our protocol incurs only a single roundtrip cost for decrypting the IDs in the leaves of the index. Fig. 9 also highlights the improvement when the index can be optimized for monotonic inserts, which was common in the applications we evaluated. We also note that though the overall latency of ArxRange is high, the results of a range query can be returned to the client *before* performing the

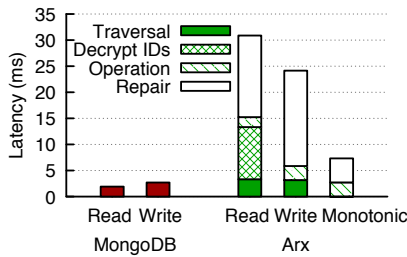


Figure 9: ArxRange latency of reads and writes.

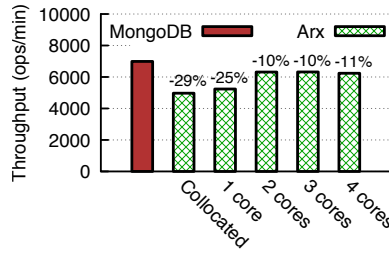


Figure 10: ShareLaTeX performance with Arx’s client proxy on varying cores

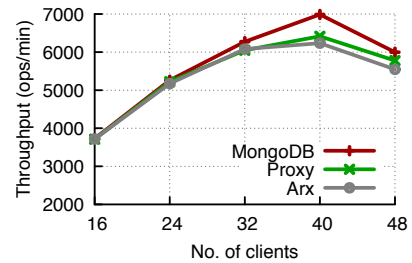


Figure 11: ShareLaTeX performance with increasing no. of client threads

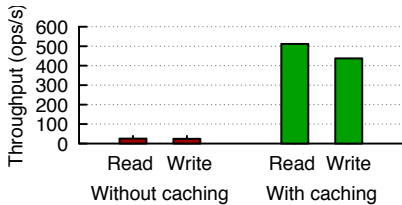


Figure 12: ArxRange throughput, with and without caching.

repair operation (see §4.4). Thus, in low load scenarios, the effective latency of a range query drops to ~15ms.

We next measure the throughput of ArxRange in Fig. 12. Without client-side caching of nodes, the throughput of the index is very limited, since each operation requires the circuit at the root of the index to be replenished, forcing the operations to be sequential. However, when the top few levels of the tree are cached at the client, multiple queries to different parts of the index can proceed in parallel, and the throughput increases by more than an order of magnitude (at which point the client proxy in our testbed gets saturated).

## 9.4 Performance of ArxAgg

The cost of computing an aggregate over a range in Arx is essentially equal to the cost of computing the range query. This is because traversing the index for a range query automatically computes the covering set. As a result, with 1M values in the index, aggregating over a range takes ~3 ms in Arx, equal to the cost of traversing the index.

## 9.5 End-to-end evaluation on ShareLaTeX

We now evaluate the end-to-end overhead of Arx using ShareLaTeX [77], a popular web application for real-time collaboration on LaTeX projects, that uses MongoDB for persistent storage. We chose ShareLaTeX because it uses both of Arx’s indices, it has sensitive data (documents, chats) and is a popular application. ShareLaTeX maintains multiple collections in MongoDB corresponding to users, projects, documents, chat messages, etc. We considered all the fields in the application to be sensitive, which is the default in Arx. The application was run on four cores of the client server.

Before every experiment, we pre-loaded the database with 100K projects, 200K users, and other collections with 100K records each. Subsequently, using Selenium (a tool for automating browsers [76]), multiple users launch browsers in parallel and collaborate on projects in pairs—(i) editing documents, and (ii) exchanging messages via chat. We ran the user processes on a separate machine with 48 cores. Fig. 10 shows the throughput of ShareLaTeX in a vanilla deployment with regular MongoDB, compared to its performance with

Arx in various configurations. The client proxy is either collocated with the ShareLaTeX application sharing the same four cores, or deployed on extra and separate cores. The application’s throughput declines by 29% when the client proxy and ShareLaTeX are collocated; however, when two separate cores are allocated to Arx’s client proxy, the reduction in throughput stabilizes at a reasonable 10%.

Fig. 11 compares the performance of Arx with increasing load at the application server, when four separate cores are allocated to Arx’s client proxy. It also shows the performance of MongoDB with the proxy without the Arx hooks. Note that each client thread issues many requests as fast as it can, bringing a load equivalent to many real users. At peak throughput with 40 clients and 100% CPU load at the application, the reduction in performance due to Arx is 11%; 8% is due to Arx’s proxy, and the remaining 3% due to its encryption and indexing schemes.

Finally, the latency introduced by Arx is modest compared to the latency of the application. In conditions of low stress with 16 clients, performance remains bottlenecked at the application, and the latency added by Arx is small in comparison, increasing from an average of 268ms per operation to 280ms. At peak throughput, the latency of vanilla ShareLaTeX is 355ms, which grows by 15% to 408ms with Arx, having marginal impact on user experience.

In sum, Arx brings a modest overhead to the overall web application. There are two main reasons for this. First, web applications have a significant overhead themselves at the web server, which masks the latency of Arx’s protocols. Second, even though ArxRange is not cheap, it’s one out of a set of multiple operations Arx runs, with the others being faster and overall more common in applications.

## 9.6 Storage

Arx increases the total amount of data stored in the database because: (1) ciphertexts are larger than plaintexts for certain encryption schemes, and (2) additional fields are added to documents in order to enable certain operations, e.g., equality checks using EQ, or tokens for ArxEq indexing. Further, ArxRange indices are larger than regular B+Trees, because each node in the index tree stores garbled circuits. Vanilla ShareLaTeX with 100K documents per collection occupied 0.56GB in MongoDB, with an extra 48.7 MB for indices. With Arx, the data storage increased by 1.9× to 1.05GB. The application required three compound ArxRange indices, which together occupied 8.4GB of memory at the server proxy while indices maintained by the database occupied 56.5MB. This resulted in a net increase of 16× at the DB server. We note, however, there remains substantial scope for optimizing index size in our implementation.



## 9.7 Comparison with CryptDB

Arx supports fewer queries than CryptDB, but we find their functionalities are nevertheless comparable. For example, CryptDB supports all the queries in the TPC-C benchmark [82], while Arx supports 30 out of 31 queries.

As regards performance, on one hand, CryptDB’s order and equality queries via PPE schemes are faster than Arx’s—with a reported overhead of  $\sim 1\text{ms}$  [72], as opposed to a few milliseconds in Arx—but also significantly less secure. On the other hand, Arx’s aggregate over a range is an order of magnitude faster for the same security, because CryptDB uses Paillier [66] to compute aggregates which requires a homomorphic multiplication per value in the range. For a range of 10,000 values, aggregates take 80ms in CryptDB compared to  $\sim 3\text{ms}$  in Arx. Overall, Arx is a heavier solution due to the significant extra security, but remains at par with CryptDB in terms of overall impact on target applications: both systems report an overhead on the order of 10%.

## 10. LIMITATIONS AND FUTURE WORK

**ArxRange extensions.** Our current ArxRange index is a binary tree. An interesting extension is to implement the index for data structures with higher fanout such as B-trees, *e.g.*, by (i) storing at each node in the tree multiple garbled circuits; and (ii) using a history-independent B-treap data structure [32], instead of a binary treap.

**History-independence.** One needs to be careful that when logically implementing a history-independent data structure (as in ArxRange), the physical implementation of it is history-independent as well. For example, in our treap data structure, we ultimately require file system support for implementing *secure deletion* [6, 74]. This is because, when a node is logically deleted, the file system needs to ensure that instead of merely unlinking the data structure in memory, all copies of the data (caches, in-memory and disk) are in fact physically removed so as to become irrecoverable to an attacker. Implementing secure deletion is complementary to our work.

**Transactions.** Arx currently does not support transactional semantics. While our techniques can be extended to transactional systems as well, it has significant practical challenges. For instance, our ArxRange index requires updates to multiple nodes in the tree per query along with interaction with the client, making support for transactions complicated. However, doing so is interesting future work.

## 11. RELATED WORK

We compare Arx with state-of-the-art EDBs, and discuss protocols related to its building blocks, ArxEq and ArxRange. We do not discuss PPE-based EDBs [5, 67, 72, 83] further as we have already compared Arx against them extensively in §1 and §2.3. Seabed [67] hides frequency in some cases, but still uses PPE.

**EDBs using semantically-secure encryption.** This category is the most relevant to Arx, but unfortunately, there is little work done in this space. First, the line of work in [16, 28] is based on searchable encryption, but is too restricted in functionality. It does not support joins, order-by-limit queries (commonly used for pagination, more common than range queries in TPC-C [82]), or aggregates over a range (because the range identifies a superset of the relevant documents for security, yielding an incorrect aggregate). Regarding security,

while being significantly more secure than PPE-based EDBs for offline attackers, for online attackers they could leak more than PPE-based EDBs because their range queries leak the number of values matching sub-ranges as well as some prefix matching information—leakage that is not implied by order. Arx addresses all these aspects. Other recent works [41, 46] also support equality-based queries but do not support range, order-by-limit, or aggregates over range queries; the former doesn’t support inserts or updates either.

Second, BlindSeer [68] is another EDB providing semantic security. BlindSeer provides stronger security than Arx and even hides the client query from the server through two-party computation. Its primary drawbacks with respect to Arx are performance and functionality. BlindSeer requires a large number of interactions between the client and server. For example, for a range query, the client and server need to interact for every data item in the range (and a few times more) because tree traversal is interactive. In Arx, there is no interaction in this case. BlindSeer also does not handle inserts easily, nor does it support deletes, updates, aggregates over ranges or order-by-limit queries.

Finally, Obladi [23] targets much stronger guarantees than Arx by combining ACID semantics with ORAM, but consequently, is also orders of magnitude slower.

**Work related to ArxEq.** ArxEq falls in the general category of searchable-encryption schemes and builds on insights from this literature. While there are many schemes proposed in this space [11, 12, 16, 24, 28, 38, 42, 47, 50, 62, 64, 79, 80], none of them meets the following desired security and performance from a database index. Besides semantic security, when inserting a value, the access pattern should not leak what other values it equals, and an old search token should not allow searching on newly inserted data (forward privacy), both crucial in reducing leakage [12]. Second, inserts, updates and deletes should be efficient and should not cause reads to become slow. ArxEq meets all these goals. Perhaps the closest prior work to ArxEq is [16]. This scheme uses revocation lists for delete operations, which adds significant complexity and overhead, as well as leaks more than our goal in Arx: it lacks forward privacy and the revocation lists leak various metadata. Sophos [12] also provides forward privacy, but uses expensive public key cryptography instead of symmetric key. Diana [13] is similar to ArxEq.

**Work related to ArxRange.** There has been a significant amount of work on OPE schemes in both industry and research communities [1–3, 8, 9, 21, 40, 52, 55, 56, 65, 71, 73, 85, 87]. OPE schemes are efficient but have significant leakage [61]. Order-revealing encryption (ORE) provides semantic security [10, 18, 53]. The most relevant of these is the construction by Lewi and Wu [53] which is more efficient than ArxRange because it does not need replenishment, but also less secure because it leaks the position where two plaintexts differ. Thus, it is not strictly more secure than OPE.

## Acknowledgments

We thank our anonymous reviewers for their invaluable feedback. This research was supported by NSF CISE Expeditions #CCF-1730628, and gifts from the Sloan Foundation, Hellman Fellows Fund, Bakar Fund, Alibaba, Amazon, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, NVIDIA, ScotiaBank, Splunk, and VMware.

## 12. REFERENCES

- [1] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database Management as a Service: Challenges and Opportunities. In *Proc. ICDE*, 2009.
- [2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. 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] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. FOCS*, 1989.
- [5] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A secure coprocessor for database applications. In *Proc. FPL*, 2013.
- [6] S. Bajaj and R. Sion. HIFS: History Independence for File Systems. In *Proc. CCS*, 2013.
- [7] S. Bearak. 2018 Data Breaches – The Worst So Far, 2018. <https://www.identityforce.com/blog/2018-data-breaches>.
- [8] A. Boldyreva, N. Chenette, Y. Lee, and A. O’Neill. Order-Preserving Symmetric Encryption. In *Proc. EUROCRYPT*, 2009.
- [9] A. Boldyreva, N. Chenette, and A. O’Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proc. CRYPTO*, 2011.
- [10] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *Proc. EUROCRYPT*, 2014.
- [11] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A Survey of Provably Secure Searchable Encryption. *ACM Computing Surveys (CSUR)*, 2014.
- [12] R. Bost. Sophos - Forward Secure Searchable Encryption. In *Proc. CCS*, 2016.
- [13] R. Bost, B. Minaud, and O. Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *Proc. CCS*, 2017.
- [14] Budget Manager. <https://goo.gl/chFmct>.
- [15] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks against Searchable Encryption. In *Proc. CCS*, 2015.
- [16] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Proc. NDSS*, 2014.
- [17] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Proc. CRYPTO*, 2013.
- [18] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. Practical Order-Revealing Encryption with Limited Leakage. In *Proc. IACR-FSE*, 2016.
- [19] Chino.io: Security and Privacy for Health Data in the EU. <https://chino.io/>.
- [20] CipherCloud. CASB+ Platform. <http://www.ciphercloud.com>.
- [21] CipherCloud. Tokenization. <http://www.ciphercloud.com/tokenization>.
- [22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SOCC*, 2011.
- [23] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proc. OSDI*, 2018.
- [24] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. CCS*, 2006.
- [25] H. Daitch. 2017 Data Breaches, 2017. <https://www.identityforce.com/blog/2017-data-breaches>.
- [26] F. B. Durak, T. M. DuBuisson, and D. Cash. What Else is Revealed by Order-Revealing Encryption? In *Proc. CCS*, 2016.
- [27] M. Egorov and M. Wilkison. ZeroDB white paper. *arXiv:1602.07168*, 2016.
- [28] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Proc. ESORICS*, 2015.
- [29] S. Garg, S. Lu, and R. Ostrovsky. Black-Box Garbled RAM. In *Proc. FOCS*, 2015.
- [30] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption. Cryptology ePrint Archive, Report 2015/1010, 2015. <http://eprint.iacr.org/2015/1010>.
- [31] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proc. STOC*, 1987.
- [32] D. Golovin. B-Treaps: A Uniquely Represented Alternative to B-Trees. In *Proc. ICALP*, 2009.
- [33] Google. Encrypted BigQuery client. <https://github.com/google/encrypted-bigquery-client>.
- [34] P. Grofig, M. Haerterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schroepfer, and W. Tighzert. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Sicherheit*, 2014.
- [35] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *Proc. IEEE S&P*, 2019.
- [36] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why Your Encrypted Database Is Not Secure. In *Proc. HotOS*, 2017.
- [37] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. Cryptology ePrint Archive, Report 2016/895, 2016. <http://eprint.iacr.org/2016/895>.
- [38] W. He, D. Akhawe, S. Jain, E. Shi, and D. X. Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proc. CCS*, 2014.
- [39] iQrypt: Encrypt and query your database. <http://iqrypt.com/>.
- [40] H. Kadhém, 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. Info. & Sys.*, 2010.

- [41] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Databases. Cryptology ePrint Archive, Report 2016/453, 2016. <http://eprint.iacr.org/2016/453>.
- [42] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proc. CCS*, 2012.
- [43] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *Proc. IEEE S&P*, 2016.
- [44] G. Kellaris, G. Kollios, K. Nissim, and A. O’Neill. Generic Attacks on Secure Outsourced Databases. In *Proc. CCS*, 2016.
- [45] J. Kepner, V. Gadepally, P. Michaleas, N. Schear, M. Varia, A. Yerukhimovich, and R. K. Cunningham. Computing on Masked Data: A High Performance Method for Improving Big Data Veracity. *arXiv:1406.5751*, 2014.
- [46] M. Kim, H. T. Lee, S. Ling, S. Q. Ren, B. H. M. Tan, and H. Wang. Better Security for Queries on Encrypted Databases. Cryptology ePrint Archive, Report 2016/470, 2016. <http://eprint.iacr.org/2016/470>.
- [47] K. Kurosawa. Garbled Searchable Symmetric Encryption. In *Proc. FC*, 2014.
- [48] M.-S. Lacharité, B. Minaud, and K. G. Paterson. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage, 2018.
- [49] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proc. SIGCOMM*, 2015.
- [50] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield-A System’s Approach to Data Privacy on Public Cloud. In *Proc. USENIX Security*, 2014.
- [51] Leanote. <https://leanote.com/>.
- [52] 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. Info. & Sys.*, 2009.
- [53] K. Lewi and D. J. Wu. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *Proc. CCS*, 2016.
- [54] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated Index Structures for Aggregation Queries. *ACM Trans. Info. & Sys. Sec.*, 2010.
- [55] D. Liu and S. Wang. Programmable Order-Preserving Secure Index for Encrypted Database Query. In *Proc. CLOUD*, 2012.
- [56] D. Liu and S. Wang. Nonlinear order preserving index for encrypted database query in service cloud environments. *Concurrency and Computation: Practice and Experience*, 2013.
- [57] R. Merkle. *Secrecy, authentication and public key systems / A certified digital signature*. PhD thesis, Stanford University, 1979.
- [58] Microsoft SQL Server. Always Encrypted Database Engine. <https://goo.gl/51LwQ9>.
- [59] M. Naor and V. Teague. Anti-persistence: History Independent Data Structures. In *Proc. STOC*, 2001.
- [60] M. Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Report 2015/668, 2015. <http://eprint.iacr.org/2015/668>.
- [61] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proc. CCS*, 2015.
- [62] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic Searchable Encryption via Blind Storage. In *Proc. IEEE S&P*, 2014.
- [63] NodeBB. <https://nodebb.org/>.
- [64] W. Ogata, K. Koiwa, A. Kanaoka, and S. Matsuo. Toward Practical Searchable Symmetric Encryption. In *Proc. IWSec*, 2013.
- [65] G. Özsoyoglu, D. A. Singer, and S. S. Chung. Anti-Tamper Databases: Querying Encrypted Databases. In *Proc. DBSec*, 2003.
- [66] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT*, 1999.
- [67] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *Proc. OSDI*, 2016.
- [68] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind Seer: A scalable private DBMS. In *Proc. IEEE S&P*, 2014.
- [69] PencilBlue. <https://goo.gl/SS4biS>.
- [70] R. A. Popa. *Building Practical Systems that Compute on Encrypted Data*. PhD thesis, MIT, 2014.
- [71] R. A. Popa, F. H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proc. IEEE S&P*, 2013.
- [72] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. SOSP*, 2011.
- [73] F. Y. Rashid. Salesforce.com Acquires SaaS Encryption Provider Navajo Systems, 2011. <https://goo.gl/MKiF2b>.
- [74] J. Reardon, D. Basin, and S. Capkun. SoK: Secure Data Deletion. In *Proc. IEEE S&P*, 2013.
- [75] Redux. <https://goo.gl/AWZy6z>.
- [76] Selenium. <http://www.seleniumhq.org/>.
- [77] ShareLaTeX. <https://www.sharelatex.com/>.
- [78] Skyhigh. Skyhigh Security Cloud. <https://www.skyhighnetworks.com/>.
- [79] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *Proc. IEEE S&P*, 2000.
- [80] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *Proc. NDSS*, 2014.
- [81] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proc. CCS*, 2013.
- [82] TPC-C Transaction Processing Benchmark. <http://www.tpc.org/tpcc/>.
- [83] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical Queries over Encrypted Data. *PVLDB*, 6(5):289–300, 2013.
- [84] UNCAP: Ubiquitous iNteroperable Care for Ageing People. <http://www.uncap.eu/>.



- [85] L. Xiao, I.-L. Yen, and D. T. Huynh. Extending Order Preserving Encryption for Multi-User Systems. Cryptology ePrint Archive, Report 2012/192, 2012. <http://eprint.iacr.org/2012/192>.
- [86] A. C. Yao. How to Generate and Exchange Secrets (Extended Abstract). In *Proc. FOCS*, 1986.
- [87] D. Yum, D. Kim, J. Kim, P. Lee, and S. Hong. Order-Preserving Encryption for Non-uniformly Distributed Plaintexts. In *Proc. WISA*, 2011.
- [88] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proc. CCS*, 2015.