Arx: A Strongly Encrypted Database System

Rishabh Poddar Tobias Boelter Raluca Ada Popa
UC Berkeley

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, existing practical proposals rely on a set of weak encryption schemes that have been shown to leak sensitive data.

In this paper, we propose Arx, the first practical and functionally rich database system that encrypts the data only with *strong* encryption schemes. Arx protects the database with the same level of security as regular AES-based encryption, which by itself is devoid of functionality. We show that Arx supports real applications such as ShareLatex and a health data cloud provider, and that its performance overhead is modest.

1 Introduction

Due to numerous and massive data breaches [28, 42], the public concern over privacy and confidentiality is likely at one of its peaks today. Unfortunately, protecting the data is not as easy as encrypting it because encryption precludes useful computation on this data.

In recent years, encrypted databases [45, 13, 54] have emerged as a promising direction that provides both confidentiality and functionality by running queries on encrypted data. CryptDB [45] has shown that such a direction can be practical and opened up a rich line of work [13, 54]. The demand for such systems is demonstrated by the adoption in industry in only a few years, despite being a radical technology. A few examples are Microsoft's Always Encrypted Service [47] currently deployed as part of SQL Server 2016, Skyhigh Networks [49], CipherCloud [48], Google's Encrypted Big Query [46], SAP's SEEED [29], Lincoln Labs [33], as well as startups such as IQCrypt [3]. Most of these services are *NoSQL databases* of various kinds showing that a certain class of encrypted computation suffices for many systems.

While existing systems provide strong security and are sufficient for a class of applications, there are many other applications for which they either leak significant private data or cannot support. The reason is that they rely on weak encryption schemes to process equality and order operations, such as deterministic and order-preserving (OPE) encryption [17, 18, 44]. These schemes tell the attacker equality relations (including the frequency count of every value), and the order of different values. Previous work [36, 39] has shown that an attacker can glean significant private data from these encryption schemes.

It would be ideal to have a database encrypted with the strong standard of encryption today, such as when using regular AES encryption, but somehow still be able to compute on it. This level of security is called *IND-CPA* (indistinguishability under chosen plaintext attack). Indeed, fully homomorphic encryption (FHE) [25] provides such security and allows full functionality, but it is still prohibitively impractical [26]. Searchable-encryption-based databases [20, 23], while more practical, are significantly limited in functionality (not supporting basic order queries and aggregates) and are inefficient for write operations, as we elaborate in §12.

In this paper, we propose Arx, the first practical and functionally rich database system that encrypts the data only with *strong* encryption schemes. Arx protects each data item with IND-CPA security, the same level of security as FHE or when encrypting each data item with regular encryption into an unfunctional database. No decryption key ever reaches the server and Arx does not rely on trusted hardware at the server.

In fact, paradoxically, Arx uses almost exclusively AES, an encryption scheme that cannot compute. Even range queries and summation are implemented using AES. This not only enables strong security guarantees, but helps performance as well due to hardware implementations of AES. To achieve this, Arx introduces a set of new mechanisms, all of them having one insight in common:
Instead of embedding the computation into special encryption schemes as in FHE and CryptDB, Arx embeds the computation into data structures, which it builds on top of traditional encryption schemes.

Arx introduces two new database indices, Arx-RANGE for range and order-by-limit queries, and Arx-EQ for equality queries. While Arx-RANGE can be used for equality queries as well, Arx-EQ is substantially faster. Both indices are data structures built on top of AES.

To provide rich computation with strong security, Arx-RANGE uses a cryptographic tool for *one-time obfuscation* at each node in an index tree. This allows running an obfuscated program at each index node that implements the desired comparisons without leaking the data it compares. Inspired by the theoretical literature on running generic programs in the cloud [24], Arx-RANGE enables the server to traverse such an encrypted index by itself. While such generic schemes are prohibitively slow, we build an index that is practical by designing for our concrete setting. For security, each such index node may only be used once, so Arx-RANGE essentially *destroys itself for the sake of security* while computing. Nevertheless, only a logarithmic number of index nodes are destroyed for each query, and Arx provides an efficient repair procedure. Arx also uses Arx-RANGE to speed up aggregations by transforming an aggregate into a lookup of a logarithmic number of nodes.

Arx-EQ works by embedding a counter into each repeating value. This ensures that the encryption of two equal values is different and the server does not learn frequency information. At the same time, this enables building a regular database index over the encryptions. When searching for a value v, the client can provide a small token to the server, which the server can expand into many search tokens for all the occurrences of v.

Because of the new indices, index and query planning becomes challenging in Arx. The application's developer specifies a set of regular indices and, as a result, expects a certain asymptotic performance. However, there is no direct mapping between regular indices and Arx indices because Arx's indices pose new constraints. The main constraints are: the same index is not used for both $=$ and $\geq$ operations, an equality index on (a,b) cannot be used to compute equality on a alone, and Arx can compute range queries only via Arx-RANGE. With these constraints 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 it is amenable to adoption. As [43] discusses, two lessons 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. It employs two proxies between the application and the DB server, which speak Arx language internally, but export the API of the DB server externally. The server proxy, in particular, converts encrypted processing into regular queries to the DB server.

We implemented Arx on top of MongoDB, a popular NoSQL database, and plan to open source our implementation. We show that Arx supports a wide range of real applications, such as ShareLaTeX [9], the Chino health data platform [2, 10], NodeBB forum [6], Leanote [4], and three others. We show that Arx adds modest performance overheads. For example, Arx decreases throughput for ShareLatex by 11% and YCSB's throughput by 3%–9%. We are collaborating with a health data cloud provider, Chino [2], for an initial deployment of Arx. Chino provides a MongoDB-like interface to medical web applications (which run on premises of hospitals). This serves the European medical project UNCAP [10], which involves 11 hospitals in 6 countries. The leaders of both UNCAP and Chino are interested in using Arx to secure their database, and are collaborating with us.

2 Overview

2.1 Model and Threat Model

Arx considers the model of an application that stores sensitive data at a database (DB) server, as in Fig. 1. The DB server can be hosted on a private or public cloud.

Arx targets attackers to the database server. Hence, our threat model assumes that the attacker does not control or see the data on the client-side (users, the application, and Arx's client proxy), and may only access the server-side (Arx's server proxy and the database servers).

Arx considers powerful server-side attackers. Such attackers include curious employees of a cloud provider, hackers breaking into the database servers, or subpoenas. The attacker can see *all* the information at the server: the entire contents of the database, any data or keys stored in memory, and any network messages received by the server. Hence, if the decryption key is in main memory, the attacker can reach it and decrypt the database. Arx does not rely on any trusted hardware at the server. The attacker is *passive*, that is, the attacker does not modify or delete the contents of the database or query results. In this paper, we do not focus on *actively* malicious attackers; the techniques to expand protection to such attackers are orthogonal and already exist in the literature [57, 32, 38].

This scenario corresponds to a set of use cases, both in the private and the public cloud. Microsoft, which uses such technology in its SQL Server, lists these use cases [47]. For example, Chino [2] is a database cloud provider for medical data. Application servers from the UNCAP project running at hospitals store patient information at Chino. Using Arx protects the confidentiality of the patient data against attackers and employees of Chino. In the private cloud setting, Arx helps remove the DB server from the trust perimeter.

2.2 Architecture

Fig. 1 shows Arx's architecture. Arx introduces two components between the application and the DB server: a trusted client proxy and a server proxy. The client proxy exports the same API as the DB server to the application so the application does not need to change. To avoid changing the DB server, the server proxy interacts with the DB server by calling its unmodified API (e.g. issuing queries); in other words, the server proxy behaves as a regular client of the DB server. Arx cannot use user-defined functions instead of the server proxy because the proxy must interact with the DB server multiple times

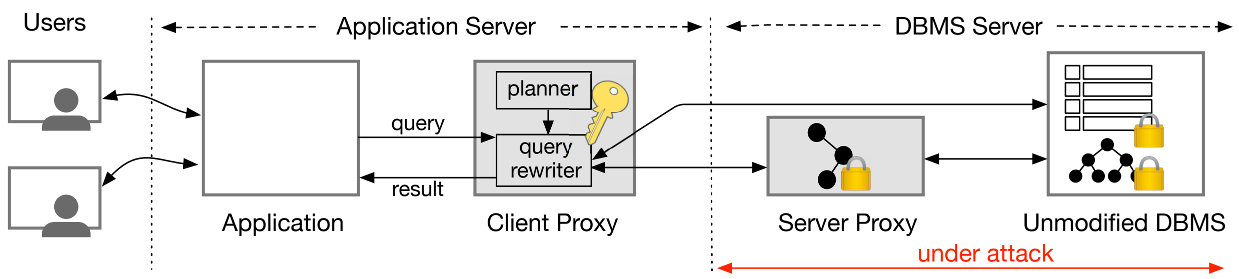


Figure 1: Arx's architecture: A trusted client proxy deployed at the application server, and an untrusted server proxy deployed at the DBMS server. The client proxy intercepts queries and encrypts sensitive information. The server proxy maintains indices over the encrypted data, and executes incoming queries. Shaded boxes depict system components introduced by Arx, and unshaded boxes represent existing components. Locks indicate that sensitive data at the component always remains strongly encrypted.

and run DB queries as part of one invocation, which is not possible from a UDF in many databases, including MongoDB.

The client proxy stores the master key. It rewrites queries, encrypts sensitive data, and forwards the encrypted queries to the server proxy for execution along with helper cryptographic tokens. It forwards any queries not containing sensitive fields directly to the DB server. The server proxy helps the server perform encrypted processing that requires changes to its regular processing.

The client proxy is *lightweight*: it does not store the database and does much less work than the server. The client proxy stores metadata (schema information) and a small optional cache. In almost all cases, the client proxy processes only the results of queries (e.g., to decrypt them). The server runs the expensive part of DB queries, which is filtering and aggregating many documents into a small result set. The client proxy rarely does more work than this: as discussed in §4, there is a corner case when the client proxy processes the results of a partial set of filters instead of all the filters.

2.3 Developer API

For concreteness, 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 our implementation is on top of MongoDB, we tried to keep Arx's design applicable to other databases.

Following the example of Microsoft's SQL Server [47] and Google's Encrypted BigQuery [46], Arx requires the developer to declare what operations will run on sensitive fields because this results in a more efficient system.

To use Arx, an application developer must specify:

1. which fields are sensitive and should be encrypted,
2. the operations that run on sensitive fields.

The second can also be automatically inferred given a complete query trace.

For the first, the developer uses the API: f *collection.sensitive* = { $field_1$: $info_1$, ..., $field_n$: $info_n$ }, specifying the fields in a collection that are sensitive. "info" is optional but if provided it increases performance of Arx. "info" should specify "unique" if the values in the field are unique. Primary keys are inferred by Arx automatically to be unique. Fields such as SSN or driver's license are unique. Moreover, specifying the maximum length in bits for these fields helps Arx choose a more effective encryption scheme. In a database that has a schema such as MySQL (which is not the case with MongoDB), Arx can infer this information from the schema.

For the second, Arx needs to know the query patterns that will run on the database. Concretely, Arx needs to know what operations run on what fields, but Arx does not need to know what constants will be queried for. For example, for the query `SELECT ID FROM T WHERE age = 10`, Arx needs to know that there will be a projection on `ID` and an equality on `age`. The developer can either specify these operations to Arx or can provide a query trace from a run of this application and Arx will automatically identify them.

2.4 Example

Let's consider a running example. We have a collection `patients` containing medical information about various patients, such as `ID`, `age`, `diagnosis`, and `days_in_hospital`. The developer specified that all fields in the collection are sensitive that the application filters by "`age` $\geq$". As a result, the Arx planner decides to maintain an Arx-RANGE index on `age` at the server.

Whenever the application inserts a document into the collection, the client proxy encrypts all values in the document to be inserted using standard encryption, as well as gives cryptographic tokens to the proxy server needed to insert the encrypted `age` value into Arx-RANGE.

When the application issues a query of the form `SELECT ID FROM patients WHERE age` $\geq$ 80, the following happens. The client proxy provides a cryptographic token to the server proxy for the value 80. The server proxy traverses the Arx-RANGE index at the DB server using the token to locate the leaf node in the index for the first value ≥ 80, and retrieves all values to the right of this leaf. It returns these values to the client proxy, who decrypts them, obtains the decrypted IDs and returns them to the application. The server proxy also informs the client proxy of the tree nodes that got destructed and receives replenishment.

2.5 Functionality

In this section, we describe the classes of read and write queries that can be supported by Arx over strongly encrypted data. Arx supports similar computation to previous approaches such as CryptDB [45].

Read queries. The read queries Arx supports are summarized by

```
SELECT [AGG doc] F(doc) FROM collection
        WHERE clause [ORDER BY f_i] [LIMIT ℓ]
```

which we now explain. Each expression in brackets is optional and *doc* denotes a document. [AGG doc] refers to aggregations over documents that can take the general form $\sum Func(doc)$. Various aggregations fit here. $\sum$ can be any associative operator and $Func$ any efficiently computable function. Examples include sum, count, sum of squares, min, max. More aggregations can be computed with minimal postprocessing at the client proxy by combining a few aggregations, such as average or standard deviation.

$F(doc)$ refers to a projection that selects various fields of the document, such as in "SELECT diagnosis, timestamp", or to other lightweight postprocessing such as formatting (e.g., convert to date format).

$$clause = \left[\wedge_{i=1}^{e} \mathtt{EQ}(f_i, v_i)\right] \wedge \left[\wedge_{i=1}^{r} \mathtt{RANGE}(f_i, v_i)\right]$$

where $\mathtt{EQ}(f_i, v_i)$ denotes equality-based operations over a field f_i that compare its value with v_i such as $=, \neq$ or array operations such as $\in, \notin$, and $\mathtt{RANGE}(f_i, v_i)$ denotes range operations over some field f_i comparing its value with v_i such as $\geq, >, \leq$. Not all such possible clauses are supported by Arx, and we discuss some constraints in §9.

Write queries. We support standard write queries such as `INSERT`, `DELETE`, and `UPDATE`. `UPDATE` works both by setting values or by incrementing values. For `DELETE` and `UPDATE` queries, similar constraints as above apply on their `WHERE` clause.

2.6 Security

Arx aims to hide the database contents as well as the data in queries. An attacker often knows the code of the application, which means the attacker knows metadata such as the database schema, what fields have indices, what operations run on the data as part of different queries, so Arx does not attempt to hide these. As with standard encryption, Arx does not hide the size of the database (the number of collections, documents, items per field, or the size of the items). Padding is a standard procedure for hiding these at a performance cost.

To better explain our security guarantees, we split attackers into two kinds: snapshot and persistent attackers. The snapshot attacker steals a snapshot of the database (tables and indices included). The persistent attacker manages to install a logger that records over time the accesses and cryptographic tokens from the client and then sends it to the attacker.

The snapshot attacker is by far the most common attacker we encounter today. For this attacker, Arx provides strong security guarantees: it provides an IND-CPA-like security to the database, which reveals nothing beyond sizing information. IND-CPA [27], indistinguishability under chosen plaintext attack, is a strong and standard level of security.

Theorem 1 (Informal)**.** *The contents of the database (collections and indices) are protected with IND-CPA-like security, and the decryption key is never sent to the server.*

In Appendix A, we define and prove formally the security guarantees of Arx, including this theorem.

For the persistent attacker, Arx provides stronger security than the existing practical and functionally-rich approaches. This attacker is less common than the snapshot attacker and is much easier to detect and stop than a snapshot attacker. The longer the attacker stays in the system, the more likely it gets caught. Compared to a snapshot attacker, a persistent attacker gets to see side channel information: timing attacks (e.g., the time when a query arrives indicates the user is online) or attacks based on access patterns (which positions in the database or index are accessed and how frequently, but not their contents). This attacker still does not see database contents which are encrypted with IND-CPA security. We note that the snapshot attacker is the typical attacker addressed by existing encryption-based mechanisms today (such as file system encryption): indeed, encryption cannot protect against access patterns by itself. While the access pattern information is typically much less sensitive than the contents of the database, it can still help the attacker glean some information. Still, since Arx keeps the database encrypted with IND-CPA security, it leaks less than prior solutions even for such an attacker. Arx does not prevent the leakage of such side channel information. Oblivious RAM [53] is an active area of research in security that hides access patterns, and Garbled RAM [24] enables combining Oblivious RAM with computation like in Arx. Despite significant progress on these fronts in recent years, there still does not exist a cost-effective solution. In Appendix A, we define the security guarantees with respect to the persistent attacker.

Arx is not tied to AES. Arx's design applies to *any* secure block cipher, not only AES. We implement Arx with AES because AES is the current safe standard and it has fast hardware implementations. Should AES be deemed insecure in the future, any secure block cipher can take its place in our design.

3 Encryption Building Blocks

Besides Arx's indexes, Arx uses three encryption schemes. These schemes already exist in the literature so we do not elaborate on them. It suffices to say that they provide IND-CPA (or equivalent) security, maintaining the strong security guarantees Arx aims for. Moreover, they are almost all based on AES, except for AGG defined below. Nevertheless, AGG is used relatively rarely because Arx performs aggregations over a range or equality of indexed sensitive fields using Arx-EQ and Arx-RANGE.

BASE is standard encryption, implemented with AES in counter mode and random initialization vector. We denote by Enc, encryption with BASE.

EQ enables equality checks. It consists of three algorithms: the client uses $\mathsf{EQEnc}_k(v) \to \mathsf{ct}$ to encrypt a value v and $\mathsf{EQToken}_k(w) \to \mathsf{tok}$ to produce a token tok used to search for w, and the server uses $\mathsf{EQEnc}(\mathsf{ct}, \mathsf{tok})$ to search, which returns true if $v = w$. To implement EQ, we use a searchable encryption scheme similar to the schemes in [51, 20].

In this scheme, $\mathsf{EQEnc}_k(v) = (\mathsf{IV}, \mathsf{AES}_{\mathsf{KDF}_k(w)}(\mathsf{IV}))$, where IV is a random value and KDF is a key derivation algorithm based on AES. To search for a word w, the token is $\mathsf{tok} = \mathsf{KDF}_k(w)$. To identify if the token matches an encryption, the server proxy combines tok with IV and checks to see if it equals the ciphertext. Note that one cannot build an index on this encryption directly because it is randomized. Hence, Arx uses this scheme only for non-indexed fields (i.e., for linear scans). When the developer desires an index on this field, Arx uses our new Arx-EQ 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 implements EQ using deterministic encryption. Deterministic encryption does *not weaken* security in this case, because when values are unique, it is as secure as IND-CPA security. Such a scheme is very fast: to check for equality, the server simply uses the equality operator, as if the data were not encrypted. Moreover, databases can build indexes on this field as before so this case is an optimization for Arx-EQ too.

AGG enables addition. Arx uses the Paillier [40] cryptosystem, which has the property that $\mathsf{AGGEnc}_k(x) \cdot \mathsf{AGGEnc}_x(y) = \mathsf{AGGEnc}_k(x+y)$ in a certain algebraic group. One can extend AGG with multiplication using the ElGamal cryptosystem if needed.

4 Arx-RANGE and Order-based Queries

We now present our index for performing order operations. In this paper, we use cryptographic tools as black boxes so one does not need to understand them.

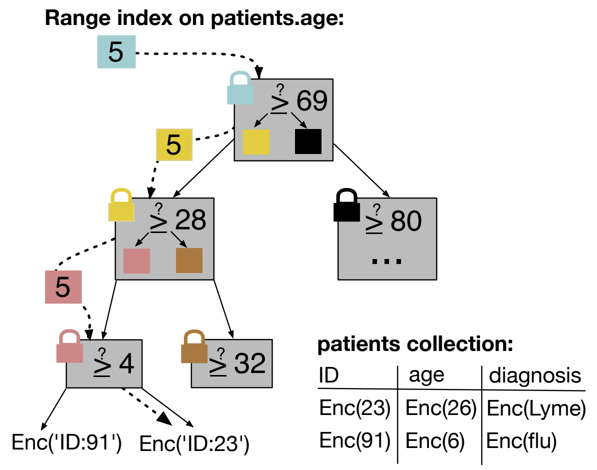
4.1 Strawman

We begin by presenting a helpful but inefficient strawman. This strawman corresponds to the protocols in mOPE [44] and the startup ZeroDB [22, 11]. For simplicity, consider the index to be a binary search tree (instead of a regular B+ tree). To obtain the desired security, each node in the tree is encrypted using regular encryption. 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 client with the root node, the client decrypts it into a value v, compares v to a, and tells the server whether to go to the left or to the right child. The server then provides the relevant node to the client, which again tells the server which way to go. This procedure repeats until the server reaches a leaf node. However, this procedure is too slow because each level in the tree requires a roundtrip. Some web applications issue tens of queries for one user click. The ZeroDB developers confirmed that this is a main issue they are struggling with.

4.2 Non-interactive index traversal

Arx-RANGE enables the server to traverse the tree by itself while maintaining security. Say the server receives $\mathsf{Enc}_k(a)$ and must locate the leaf node corresponding to a. For this goal, the server must be able to compare $\mathsf{Enc}_k(a)$ with the encrypted value at a node, say $\mathsf{Enc}_k(v)$. The idea is to store **an obfuscated program at each tree node** that performs the comparison. The obfuscation hides a and v from the attacker.

Garbled circuits help implement such obfuscation. Using a garbling scheme [55], the client can *garble* a program P using a key k and create an obfuscated program ObfP, also called a garbled circuit. The algorithm GarbleEnc produces an encoding for a value a using the key k denoted e_a. The server can run ObfP on e_a and obtains $P(a)$. The security of garbled circuits guarantees that the server learns *nothing* about a or the data hardcoded in P other than the output $P(a)$. 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



ID	age	diagnosis
Enc(23)	Enc(26)	Enc(Lyme)
Enc(91)	Enc(6)	Enc(flu)

Figure 2: An example of Arx-RANGE.

using the same key k to the server, the security guarantees no longer hold.

Inspired from the theoretical literature on garbled RAM [24], we place a garbled circuit at each node in the tree and **chain the garbled circuits**. Each garbled circuit outputs an encoding of the same input that can be used with the relevant child garbled circuit.

Let N be a node in the index with value v, and let L and R be the left and right nodes. The client generates a key for every node: k_N, k_L, and k_R. The garbled circuit at N is a garbling with key k_N of the comparison program:

```
if $a \leq v$ then
    $e_a \leftarrow \mathsf{GarbleEnc}(k_L, a)$; output $e_a$ and 'left'
else
    $e_a \leftarrow \mathsf{GarbleEnc}(k_R, a)$; output $e_a$ and 'right'
end if
```

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, $\mathsf{GarbleEnc}(5)$ 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 garbled circuit on the new encoding. The server proceeds similarly until it reaches the desired leaf node.

4.3 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 clients needs to supply new garbled circuits to replace the circuits consummed. Fortunately, only a logarithmic number of garbled circuits got consumed. Consider that a node N and its left child L were consumed. However, for each node N, the client needs two pieces of information from the server: the value encoded in N, v and the key for the right child R. Instead of sending N's garbled circuit to the client, the server sends an encryption of v, $\mathsf{Enc}(v)$ (stored separated in the index), and the ID of the garbled circuit at R, which was used to derive the key. The key for a garbled circuit is not small (1KB for a 32-bit comparison), so this procedure saves bandwidth.

4.4 A secure database index

We need to take two more steps to obtain a secure index.

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 [12] instead of a regular search tree. This data structure has the property that its shape is the same independent of the insertion or deletion order. Of course, it is not guaranteed that the system maintains no history-dependent information. For example, in our implementation, we have no control of where in memory the language runtime decides to place certain data which depends on history. Nevertheless, using a history-independent data structure is a first step in this regard.

Second, we store at each node in the tree the encrypted primary key of the document to which this value belongs. This enables locating the documents of interest. If the primary key were not encrypted, the server would learn the order of the relevant fields based on their order in the tree.

Running queries using the index. Consider the query `SELECT * FROM patients WHERE age ≥ 1 AND age ≤ 5`. Each node in the index tree has two garbled circuits to allow for a range. The client proxy provides $\mathsf{GarbleEnc}$ tokens to the server for the values 1 and 5 so that the server locates the leftmost and rightmost leaves in the interval $[1,5]$. The server fetches the encrypted primary keys from all the nodes in between which form the range of interest. The server sends this information to the client proxy which decrypts them, randomizes their order, and then selects the documents based on this primary key from the server. The purpose of the randomization is to hide from the server the order of the documents matching the range.

For aggregates over a range, the server will not send the entire range, as we describe in §6. The server answers `ORDERBY LIMIT` L queries by simply taking the leftmost or rightmost L nodes. Inserting and deletion of values in the index happens similarly to the index traversal. For monotonic inserts, 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 a tree traversal. As a performance optimization, order-by queries are not performed using Arx-RANGE. Since they do not have a limit, they do not do any filtering, so the client proxy can simply sort the result set itself.

4.5 Concurrency

Arx-RANGE provides limited concurrency because each index node needs to be repaired before it can be used again. To provide 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 within each such tree 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. For example, for less than 40KB of storage on the client proxy (which corresponds to about 12 levels of the tree because the tree is not entirely full), there will be about 1024 nodes in the first level of the tree, so 1024 queries can proceed in parallel. Hence, for a little client storage, the degree of concurrency can be significant. Queries to the same subtree still need to be sequential. A common case are monotonic inserts, but for these we have an optimization, as follows.

Monotonic inserts refer to inserts in increasing or decreasing order. In this case, the client proxy maintains the latest value inserted in an Arx-RANGE. If the value to be inserted is larger (for increasing values) than this, the client proxy knows where in the tree the new value must be inserted which avoids the tree traversal and the repair of the corresponding path.

Of course, queries to another Arx-RANGE index or to other parts of the DB can proceed in parallel.

4.6 Garbled circuit design

Garbled circuits are constructed entirely from AES. They take as input a program written using boolean gates. One of the main drawbacks of garbled circuits is that converting even a simple program to such a circuit often results in large circuits, and hence bad performance.

We put considerable effort into making our garbled circuits short and fast; in interest of space, we mention only briefly the steps we took. We used the short circuit for comparison from [35], which represents comparison of n-bit numbers in n gates. We employ transition tables between two garbled circuits, to avoid encoding the key for a child circuit inside the garbled circuit. Since this key is large, this reduces the size of the garbled circuit by a factor of 128. We use the half-gates technique [56] to further halve the size of garbled circuit. Since all garbled circuits have the same topology but different ciphertexts, we decouple the topology from the ciphertext it contains. The server hardcodes the topology and the client transmits only ciphertexts. We are planning to release our garbling library which can be used to compute any function, not only the one of interest here.

4.7 Security

The index provides the security desired: if an attacker steals the database, the index provides IND-CPA security. The index leaks nothing about the data other than the number of elements in the database. Note that the index does not even leak the order of the values. The reason is that the mapping between a node and a row in a collection is encrypted at the nodes. The access patterns during search visible to the server are the same as in the strawman: the server sees which path in the tree was taken, but not the query or the values at the nodes. We formalize and prove rigorously the security of our index scheme in [16].

5 Arx-EQ and Equality Queries

The Arx-EQ index enables filtering based on equality expressions such as in: `SELECT` [...] `WHERE age = 80`. As explained in §12, Arx-EQ builds on insights from the searchable encryption literature [19]. We begin by presenting a base protocol that we improve in stages.

5.1 Base protocol

Consider an index on the field `age`. Arx-EQ 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: Arx-EQ 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 counter, 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 example, for `age`, this map has about 100 entries.

Encrypt and insert. Consider that the application performs an insert for a document where `age` has value v. The client proxy first increments $\mathsf{counter}[v]$. Then, the proxy encrypts v into:

$$\mathsf{Enc}(v) = H(\mathsf{EQunique}(v), \mathsf{counter}[v]), \qquad (1)$$

where H is a cryptographic hash (modeled as a random oracle). This encryption provides IND-CPA security because $\mathsf{EQunique}(v)$ is a deterministic encryption scheme which becomes randomized when combined with a unique salt per value v: $\mathsf{counter}[v]$. This encryption is not decryptable, but as discussed in §8.2, Arx encrypts v with BASE as well. The document with the encryption of v is then inserted in 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 `80`. The search token for a value v is the list of encryptions from Eq. (1) for every counter from 1 to $\mathsf{counter}[v]$: $H(\mathsf{EQunique}(v), 1), \ldots, H(\mathsf{EQunique}(v), \mathsf{counter}[v])$.

Search. The server proxy uses the search token to and construct a query of the form: `SELECT [..] WHERE` `age` $= H(\mathsf{EQunique}(v), 1)$ `OR ... OR age` $= H(\mathsf{EQunique}(v), \mathsf{counter}[v])$ (with the clauses in a random order). The DB server uses the regular index built on `age` for each clause in this query. The results correspond to the search results.

Note that, if the application inserts more ages equal to v after the search, the server cannot use the old search token to learn if the new values are equal to v because the new values 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 $(\log \mathsf{counter}[v])$ instead of $\mathsf{counter}[v]$.

Instead of encrypting a value v as in Eq. (1), the client proxy hashes according to the tree in Fig. 3. It starts with $\mathsf{EQunique}_k(v)$ 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 $0, 1, 2, 3, ..\mathsf{counter}[v]$.

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.

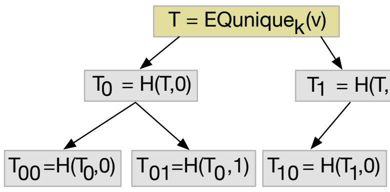


Figure 3: Search token tree.

New search token. To search for a value v with counter $\mathsf{counter}[v]$, the proxy computes the *covering set* for leaf nodes $0, \ldots, \mathsf{counter}[v] - 1$. The covering set is the set internal tree nodes whose subtrees cover exactly the leaf nodes $0, \ldots, \mathsf{counter}[v] - 1$. For the example in Fig. 3, $\mathsf{counter}[v] = 3$ and the covering set of the three leaves is node T_0 and node T_{10}. The search token are the numbers in the covering set. The covering set can be easily deduced from the binary representation of $\mathsf{counter}[v] - 1$.

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

5.3 Updates

We have already discussed insert. To delete a document, Arx simply deletes this 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 the accuracy of the search, but as more counters go missing, it affects throughput because the DB server wastes cycles looking for values with no matches. Hence, when a search query for a value v indicates more than a threshold of missing counters, Arx-EQ runs a cleanup procedure.

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

In this case, the client proxy does as much work as the number of matches for v. If the search query filters only on `age`, the proxy does as much work as the result set. If the query had additional filters outside of `age`, the proxy does more work than the result set, which is not ideal. This case might be rare if deletes are not common. Nevertheless, avoiding this case is an interesting future work.

5.4 The counter map

We now discuss the implications of storing the counter map at the server or at the client proxy. While the counter map can be stored encrypted at the server and still provide our strong guarantees against a snapshot attacker, we recommend storing it at the client for increased security against the persistent attacker.

Counter map at server. The counter map can be stored encrypted at the server. An entry of the sort $v \rightarrow \mathsf{ct}$ becomes $\mathsf{EQunique}_{k_1^*}(v) \rightarrow \mathsf{EQunique}_{k_2^*}(\mathsf{ct})$, 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 $\mathsf{EQunique}_{k_1^*}(v)$ 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. The security of this scheme satisfies Arx's goal in §2.6: a stolen database remains IND-CPA encrypted and nothing leaks about it other than sizes.

Counter map at client. However, we recommend keeping the counter map at the client proxy for added security. This approach provides higher security against a persistent attacker, who observes access patterns over time beyond stealing a snapshot of the database. For every newly inserted value, the attacker sees which entry of the counter map is accessed and which document is inserted in the database. In this way, the attacker can compute the number of times each entry appears in the database and which documents it corresponds to. Even though the encryption hides the value of the entry, if an attacker manages to watch for a sufficiently long time, sensitive frequency information can leak. Storing the counter map at the client hides entirely such correlations. For each insert query, the only access pattern is inserting that document.

Moreover, there are many fields for which the counter map is very small (e.g., gender, age, letter grades). Furthermore, when all values are unique (the maximum size for a counter map), Arx-EQ defaults to the regular index built over EQunique encryptions, not needing any counter map. 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.

5.5 Array-based operations

Arx-EQ can also be used to handle array-based equality operations such as push, pull, set, unset, $\in$, and $\notin$. We discuss such operations in greater detail in § 8.4.

5.6 Security

We formalize and prove the security of Arx-EQ in Appendix A, and show that it maintains the guarantees outlined in §2.6.

6 Aggregation queries using Arx-AGG

We now explain Arx's aggregation over the encrypted indices. It is based on AES and faster than homomorphic encryption like Paillier [40].

Many aggregations happen over a range query such as computing the average days in hospital for people in a certain age group. Arx computes an average by selecting sum and count and dividing them in the client proxy. Hence, let's focus on the query: `select sum(days_in_hospital) from patients where 70` $\leq$ `age` $\leq$ `80`.

The idea behind aggregations in Arx is inspired from the authenticated data structure literature [37]. This work targets integrity guarantees (and not confidentiality), but interestingly, we use it for computation on encrypted data. Consider the Arx-RANGE 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 `days_in_hospital` corresponding to the leaves under N. The root node thus contains the sum of all values. This value is stored encrypted with BASE.

When the server needs to compute the sum over an arbitrary range, such as $[70,80]$, the server locates the edges of the range as before, and then it identifies a *perfectly covering set*. Note that this set of nodes is *logarithmic* in size. The server returns the encrypted aggregate of all children and the encrypted value of the node itself for each node in the covering set to the client proxy, which decrypts them and sums them up.

In the case of (1) inserting/deleting a document or (2) modifying a field having an aggregate, the partial sums on the path from N to the root need to be updated, where N is the node in the tree corresponding to the changed document. In the second case, the client also needs to repair the path in the tree, so the partial sum update happens essentially for free.

This aggregation strategy supports any aggregation function of the form $\sum F(doc)$ where F is an arbitrary function whose input is a document, as explained in §2.5. For aggregates over fields with an EQ index, we have a similar strategy to the aggregates over a range, but we do not describe it here due to space constraints. For all other cases, we use Paillier. However, the number of such cases is reduced significantly.

7 Joins using Arx-JOIN

We now describe how Arx supports a class of join operations, namely, foreign-key joins. Arx extends Arx-EQ or Arx-RANGE for this purpose. This assumes that the join contains:

```
FROM T1 JOIN T2 ON T1.fkey = T2.ID
        WHERE T1.field index-op,
```

where `T1` and `T2` are the two tables being joined, `fkey` is the foreign key pointing to the primary key `ID` in `T2`, and `ind-op` is either an equality or range operation on an index, Arx-EQ or Arx-RANGE.

Arx-EQ-based joins. Consider an example with table `T2` having a primary key ID and table `T2` having a field `age` with Arx-EQ and `diagnosis`, which is a foreign key pointing to `T2.ID`.

The primary key in the secondary collection `T2.ID` is encrypted with EQunique as before. Consider inserting a document with age 10 and diagnosis 'flu' in `T1`, and let's discuss how the client proxy encrypts this pair. Since foreign keys are not unique, `T1.diagnosis` is encrypted with BASE. Additionally, to perform the join, the client proxy computes an *encrypted pointer* for `T1.diagnosis`. When decrypted, this pointer will point to the appropriate encrypted `T2.ID`. Instead of using one key for Arx-EQ, the client proxy now uses two keys k_1 and k_2. It generates a token for each key as before: t_1 and t_2. The client proxy includes t_1 in the document as before, and uses t_2 to encrypt the diagnosis 'flu' as in: $J = \mathsf{BASE}_{t_2}(\mathsf{EQunique}(\text{'flu'}))$. J will help with the join. Hence, upon insert, the pair (10, 'flu') becomes ($\mathsf{BASE}(10)$, t_1, $\mathsf{BASE}(\text{'flu'})$, J). Note that the client does not add t_2 to the document: this prevents an attacker from decrypting the join pointer and performing joins that were not requested.

Now consider the join query: `SELECT [...] FROM T1 JOIN T2 ON T1.diagnosis = T2.ID WHERE T1.age = 10`. To execute this query, the server proxy computes t_1 and t_2 for the age of 10, as usual with Arx-EQ. It locates the documents of interest using t_1, and then uses t_2 to decrypt J and obtain EQunique('flu'). This value is a primary key in `T2`, and the server simply does a lookup in `T2`.

Arx-RANGE-based joins. Arx employs a different strategy in case the `WHERE` clause of the join query requires an Arx-RANGE index for execution, e.g. `WHERE T1.age > 10`. In such a scenario, Arx-JOIN tokens for `T1.age` cannot be computed as described above.

Instead, the foreign key values encrypted with BASE are directly added to the nodes of the Arx-RANGE index over `T1.age`, which already contain the encrypted primary keys of documents in `T1` (as described in § 4.4). While traversing the index in order to resolve the `WHERE` clause, the server fetches the encrypted foreign keys as well from the nodes of interest, and sends them to the client proxy for decryption as with regular Arx-RANGE. The client decrypts the encrypted foreign keys, re-encrypts them with EQunique, shuffles them, and returns them to the server. The server then uses these values to locate the corresponding documents in `T2`, and performs the join. Note that this strategy does not bring any extra round trips between the proxies.

8 Arx's Planner

Arx's planner takes as input a set of query patterns, Arx-specific annotations, and a list of regular indices, and produces a data encryption plan, a list of Arx-style indices to build, and a query plan for each query pattern.

8.1 Index planning

Before deciding what index to build, note that Arx-RANGE and Arx-EQ 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` $\geq$ `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 developer expects: if she specified an index over certain fields, then the time to execute queries on those fields should be logarithmic and should 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 between 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 Arx-RANGE index for equality operations because of its higher cost and different security.
- Unlike a regular index, a compound Arx-EQ index on (a,b) cannot be used to compute equality on a alone because Arx-EQ performs a complete match.
- A range or order by limit on a sensitive field can be computed only via an Arx-RANGE index, so it can no longer be computed after applying a separate index.

All these are further complicated by the fact that the developer can declare compound indices on a mixture of fields, both sensitive and not. Similarly, queries can have both sensitive and regular fields in a `where` clause.

As a consequence of our performance goal and these constraints, interestingly, there are cases when Arx builds an Arx-RANGE index on a composition of a non-sensitive and a sensitive field. Consider, for example, that the developer built an index on a, a nonsensitive field, and wants to perform a query containing `WHERE` $a =$ `AND` $s \geq$, where s is sensitive. The developer expects the DB to filter documents by a rapidly based on the index, and then, to filter the result by "$s \geq$".

If we follow the straightforward solution of building an Arx-RANGE index on s alone, the resulting asymptotics are different. The DB will filter by s and then, it will scan the results and filter them by a, rendering the index on a useless. The reason the developer specified an index on a might be that performance is better if the server filters on "$a =$" first; hence, the new query plan could significantly affect the performance of this query especially if the Arx-RANGE index returns a large number of matches. To deliver the expected performance, Arx builds a composite Arx-RANGE index on (a,s). Note that this is beneficial for security too because the server will not learn which documents match one filter but not the other filter: the server learns only which documents matched the entire where clause in an all-or-nothing way.

Despite all these constraints, our index planning algorithm is quite simple. It also applies to queries that have multiple query plans using different indices, in which case it maintains the asymptotics of every query plan. The index planner 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.
Example: For clarity, we use three query patterns as examples. Their `where` clauses are: W_1: "$a =$ `and` $b =$", W_2: "$x =$ `and` $y \geq$ `and` $z =$". The indices specified by the developer 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 §9, 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 automatically 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. If b and y are sensitive (a,x,z can be either way), for S above, the indices Arx builds are: Arx-EQ (a,b) and Arx-RANGE (x,y).

One can see why our planner maintains the asymptotic performance of the developer's index specification: it ensures that each expression that was sped up by an index remains sped up. Moreover, it creates extra indices only to meet the encryption constraints. In §11, we show that the number of extra indices Arx builds is modest and does not blowup in real applications.

8.2 Data layout

After deciding which indices to build, laying out the data 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 Arx-RANGE or Arx-EQ. 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 index aggregation strategy described in §6. If not, encrypt the respective fields with AGG if the aggregate requires the computation of a sum.
- For every sensitive field projected by at least one query, encrypt it with BASE. The reason is that EQ and our indices are not decryptable.
- For every query pattern, if the where clause W_i performs an equality on a field, and this field is not part of every element of S_i, encrypt the field with EQ (which could be in addition to a BASE encryption). The reason is that at least one query plan will need to filter this field by equality outside of an index.
- For all sensitive fields that do not fall in any of the above, simply encrypt them with BASE.

8.3 Query planning and execution

Arx knows which indices are relevant to each query pattern from index planning. Sometimes a query has multiple options for which index to use. For example, if there is an index on `age` and one on `diagnosis`, a query containing `WHERE age` ≥ 10 and `diagnosis='flu'` can use the filter on `age` or the one on `diagnosis`. Different databases take different strategies in this case. In MongoDB, for example, the two plans are ran in parallel, and the first plan that produces the result completes the query. In databases like MySQL, the database chooses the best plan based on statistics about the data. One can add statistics-based planning to Arx by maintaining the statistics at the client proxy. Our indexing plan above enables both to work, but our implementation naturally follows the first. Given a query, for each index option it has, Arx runs in parallel the query plan for that index. For each query plan, Arx first uses Arx-EQ or Arx-RANGE, and then does any remaining filtering based on EQ or unencrypted fields.

1) Index filtering at the server. Arx first uses the index to filter the results, as described in §5 for Arx-EQ or in §4 for Arx-RANGE.

2) Additional filtering at the server. If the query has clauses not captured by the index, these are either equality on sensitive fields or unrestricted operations on nonsensitive fields. For the first, the server proxy filters the results using EQ with a search token from the client proxy. For the second, the server proxy issues a regular query to the DB server.

3) Postprocessing at the client proxy. For MongoDB, postprocessing happens rarely and only to finalize certain aggregates. For example, Arx cannot compute average at the server, but it can compute sum and count. The client proxy then divides the sum and count to obtain the average.

It is worth mentioning that other databases like MySQL offer a set of postprocessing functions (denoted F(doc) in §2.5) such as formatting: `FORMAT` (timestamp, "MM/DD/YY"). These are cheap to compute and run on the result of filtering, so they can be easily performed at the client proxy.

8.4 Queries over arrays

In MongoDB, certain fields in a document might have fields whose values are arrays, e.g. `names = [ 'Alice', 'Bob' ]`. To support read and write queries over such fields, it might not be enough to build an index over the field as a whole. For example, the query `SELECT * 'Alice' ∈ names` returns all documents where the array contains the value `'Alice'`. Such a query cannot be handled by an Arx-EQ index over `names` as a whole. Thus, read operations on such fields are handled by *indexing the individual elements* in the array field across documents, as opposed to indexing the array as a whole. This also enables the execution of write operations on such fields that push new values into the array or pop values from it.

Alternatively, some queries might perform operations on elements of the array at a specific position. For example, the query `SELECT * WHERE names.0 = 'Alice'` returns all documents in which the first element of `names` is `'Alice'`. To handle such queries, separate indices are built *per array position*. This means that instead of maintaining a single index for `names`, a separate index is maintained for `names.0`.

Finally, consider the query `SELECT * WHERE names ∈ ['Alice','Bob']`, where the field `names` itself is *not* an array field. This query returns all documents where `names` is either `'Alice'` or `Bob`. Arx executes this query using an Arx-EQ index over `names`. Arx first transforms the query into the following query instead: `SELECT * WHERE names = 'Alice' ∨ names = 'Bob'`. The transformed query is then executed using the Arx-EQ index over `names`.

9 Discussion and Limitations

Arx does not support operations beyond those listed in §2.5. Furthermore, Arx poses some constraints on the `where` clause because range and order-by-limit operations can be handled only via an Arx-RANGE index:

1. The query may contain `RANGE` operations over at most one sensitive field. While $f_1 \geq 3$ `and` $f_1 \leq 5$, is supported, $f_1 \geq 3$ `and` $f_1 \leq 5$ `and` $f_2 \leq 10$ is not supported, but additional `RANGE` operations over non-sensitive and not indexes fields are supported.
2. If the query contains a `LIMIT` along with `RANGE` operations over a sensitive field, then it may contain an `ORDERBY` operation over the sensitive field *alone*.

Further, the Arx-JOIN scheme described in § 7 is limited to joins over foreign keys. Joins over values that are *not* foreign keys are more complicated because a document in the queried collection `T1` may join with multiple documents in the secondary collection `T2`, and the scheme would need to account for dynamic inserts and updates in `T2`. Handling such joins is part of future work. The implementation at certain points needs to ensure that data is securely deleted at the server, otherwise our security guarantees against a snapshot attacker can not be achieved. Also, the implementation needs to be completely history-oblivious. Both points require very careful low-level implementation and are impossible to achieve with Java.

Query logging. Some application administrators may want the database to log queries for debugging purposes. Maintaining a query log can help a snapshot attacker become as powerful as a persistent attacker: a snapshot of the log provides history of accesses. This situation can be fixed in two simple ways: the client proxy can instead do the logging, or the client proxy encrypts the query and any tokens with BASE so the server logs encrypted information.

10 Implementation

While the design of Arx is decoupled from any particular database system as described in §2.2, we implemented our prototype for MongoDB 3.0, one of the most popular NoSQL data stores. Arx consists of two transparent proxies between the application and the database server based on the Netty I/O framework [5]. Arx does not require modifications to applications or the database.

Arx's implementation consists of ∼11,500 lines of Java, along with ∼600 lines of C/C++ code. Additionally, we implemented a C++ library for garbling, Arx-Garble, for Arx-RANGE. We incorporated a set of recent advances in garbling, which increase performance. The library consists of ∼1200 lines of C++ code, and it can be used for general functions, not only for Arx-RANGE. We plan to release the source code of both Arx and Arx-Garble soon.

Finally, while Arx supports join operations by design using the Arx-JOIN scheme, joins are not supported in many NoSQL databases including MongoDB (until recently). Most NoSQL applications implement any needed joins in the application. Thus, we did not implement joins in our ptototype because we did not need it in the MongoDB applications we found.

11 Evaluation

In this section, we show that Arx supports real applications with a small developer effort and a modest performance overhead.

11.1 Functionality

To understand if Arx supports real applications, we evaluate Arx on seven existing applications built on top of MongoDB. All these applications contain sensitive user information. Table 4 summarizes our results. Some fields are clearly sensitive (heart rate, private messages) and we marked them as such, but other fields were less clearly so, such as timestamps. We conservatively

Application	Protected fields		NS	# LoA	No. of Arx-EQ	No. of Arx-RANGE	Total no. of indices	
	No.	Examples					Vanilla	With Arx
ShareLaTeX [9]	50	document lines, edits	0/1	62	4	3	12	15
Uncap (medical) [10]	17	heart rate, medical tests	–	18	0	2	2	2
NodeBB (forum) [6]	13	posts, comments	–	17	9	3	12	16
Pencilblue (CMS) [7]	54	articles, comments	0/2	60	19	18	70	73
Leanote (notes) [4]	111	notes, books, tags	2/2	129	23	13	69	83
Budget manager [1]	21	expenditure, ledgers	–	25	3	0	5	5
Redux (chat) [8]	14	messages, groups	–	17	2	0	3	3

Figure 4: Examples of applications supported by Arx: the number of fields we deemed as sensitive and annotated them using Arx, a few examples of what these fields are, NS – the number of queries not supported(the right number) with the same number excluding timestamps (the left number), the number of lines of annotations (LoA) the developer had to specify to protect these fields, how many Arx-specific indices it uses, and the number of indices the database builds in the vanilla application and with Arx (which includes indices on nonsensitive fields). Since Arx-AGG is built on Arx-EQ and Arx-RANGE, we do not count it separately.

marked fields as sensitive. Regarding queries not supported, half were due to timestamps which are less sensitive. The limitation was the number of range/order operations Arx allows in the query, as explained in §9. For Leanote, the two queries were performing regular expression on the sensitive fields, which Arx cannot support. Nevertheless, the tables shows that Arx can protect almost all sensitive fields in these applications.

In terms of the developer's effort, there is *no change* to application code. The LoA mostly equals the number of fields annotated, the difference being mostly in formatting. The table also shows that, while Arx's index planner increases the number of indices by 14%, this number does not blow up. The main reason is that the number of fields that both are sensitive and have an order query on them are small.

11.2 Performance setup

To evaluate the performance of Arx, we used the following setup. Arx's server proxy was collocated with MongoDB 3.0.11 on 4 logical cores of a machine with two 2.3GHz Intel E5-2670 Haswell-EP 12-core processors and 256GB of RAM. Arx's client proxy was deployed on 4 logical cores of an identical machine. A separate machine with four 2.0GHz Intel E7540 Nehalem 6-core processors and 256GB of RAM was used to run the clients. In throughput experiments, we ran the clients on all 48 logical cores of the machine to measure the server at maximum capacity. All three machines were connected over a 1Gb Ethernet network.

We start the evaluation with low level microbenchmarks and work our way to end-to-end experiments.

11.3 Encryption schemes microbenchmarks

The cryptographic schemes used by Arx are efficient, as shown in Fig. 5. The results were averaged over multiple iterations.

Scheme	Enc.	Dec.	Token	Operation
BASE	0.327	0.13	–	–
EQ	4.998	–	2.353	Match: 2.368
EQunique	0.012	0.047	–	Equality: ∼0
AGG	16,254	15,116	–	Sum: 8

Figure 5: Microbenchmarks of cryptographic schemes used by Arx in μs

11.4 Performance of Arx-EQ

We evaluate the performance of Arx-EQ (without the unique optimization) using relevant queries issued by ShareLaTeX. These queries filter by one field using Arx-EQ, allowing us to focus on Arx-EQ. The field is the document ID in the history collection which is a log of all changes to any documents. We first loaded the database with 100K documents representative of a ShareLaTeX workload.

Fig. 6 compares the read throughput of Arx-EQ with a regular MongoDB index, when varying the number of duplicates per value of the indexed field. The Arx-EQ scheme expands a query from a single equality clause into a disjunction of 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. At the same time, 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, Arx-EQ incurs a performance penalty of 55% in the worst case, of which ∼8% is due to Arx's proxy. Further, when all fields are unique, the added latency due to Arx-EQ is small—1.13ms as opposed to 0.94ms for MongoDB, as shown in Fig. 9. As the number of duplicates increases, the latency of both MongoDB and Arx increase in similar proportions—at 100 duplicates, the latency for Arx is 42.1ms, while that of MongoDB is 18.8ms.

Fig. 7 compares the write throughput of Arx-EQ with

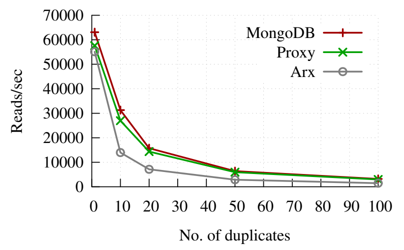


Figure 6: Arx-EQ read throughput with increasing no. of duplicates.

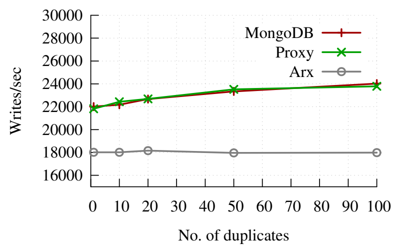


Figure 7: Arx-EQ write throughput with increasing no. of duplicates.

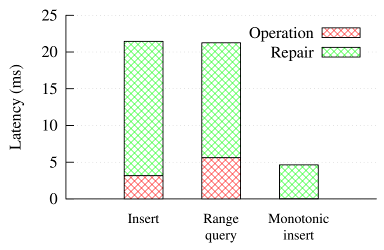


Figure 8: Arx-RANGE latency of reads and writes

Dup.	Read latency (ms)			Write latency (ms)		
	Mongo	Proxy	Arx	Mongo	Proxy	Arx
1	0.94	1.04	1.13	2.69	2.72	3.30
10	1.91	2.23	4.29	2.69	2.66	3.34
20	3.81	4.19	8.49	2.62	2.65	3.28
50	9.40	10.09	20.86	2.55	2.53	3.33
100	18.80	20.23	42.10	2.50	2.51	3.35

Figure 9: Arx-EQ latency of reads and writes with increasing no. of duplicates.

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 Arx-EQ 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, which is a constant-time operation, the write throughput of Arx-EQ 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, as shown in Fig. 9, and remains stable for Arx-EQ 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.

11.5 Performance of Arx-RANGE

Our garbled circuits are implemented in AES, which takes advantage of existing hardware implementations. For a 32-bit value, the garbled circuit is 3088 bytes long, the time to garble is 19786 cycles and the time to evaluate is 7842 cycles. For a 128-bit value, the circuit is 12304 bytes, the time to garble is 70109 cycles (0.03ms) and the time to evaluate is 29099 cycles.

Each circuit has the size of two ciphertexts per gate and additionally needs to store a 128 bit unique random id. An n bit const-comparison circuit has exactly n gates. No additional metadata is needed, hence the size for a n bit comparison circuit is $n \cdot 2 \cdot 128 + 128$ bit. For a 32 bit comparator this amounts to 1040 byte. One transition table has the size $n \cdot 2 \cdot 128$ bit. Hence the size of a complete node is $n \cdot 6 \cdot 128 + 128$ bit, which again for a bitlength of 32 bit results in 3088 byte.

The cost of evaluating a 32 bit comparison circuit is dominated by the 64 AES evaluations needed. Theoretically of these 64 evaluations, 32 come for free as they are independent and hence can exploit instruction level parallelism. A single AES instruction has a latency of 7 cycles on modern CPUs, hence the complete evaluation of the circuit can theoretically be as fast as 224 cycles, at least for the AES part.

The overhead stems from the fact that we currently use the gcrypt library to evaluate AES and we have not yet done any low-level optimizations.

We now evaluate the latency introduced by Arx-RANGE. We pre-inserted 1M values into the index, and assumed a length of 128 bits for the index keys, 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 different operations on the index. Fig. 8 illustrates the latency of each operation, divided into two parts: (1) the time taken to perform the operation, and (2) the time taken to repair the index. The generation of fresh garbled circuits in order to repair the index remains the primary contributor towards latency.

Range queries cost more than writes because the former traverse two paths in the index (for bounded queries), while the latter traverse a single path. The latency for a range query is about 6 ms. We note that using the strawman in §4.1, one incurs a roundtrip overhead for each node in the tree, which is roughly as long our entire range query. The figure also highlights the performance improvement when the index can be optimized for monotonic inserts, which was common in the applications we evaluated.

11.6 Performance of Arx-AGG

Computing an aggregate over a range with Paillier as in CryptDB [45] takes significantly longer than with Arx. In Arx, this cost is essentially zero because traversing the index for a range query automatically computes the cover set. In CryptDB, one has to do a homomorphic multiplication for every value in the range. For example, aggregating over a range size of 10,000 values, Arx takes about 0 ms for the aggregate and CryptDB takes 80ms. With the cost of the range, Arx is 13 times faster.

11.7 Comparison to CryptDB

We cannot compare to CryptDB directly because CryptDB is implemented on MySQL. On one hand, CryptDB's order queries via order-preserving encryption are faster than Arx's, but such encryption scheme is significantly less secure. On the other hand, Arx's aggregate over a range is faster than CryptDB's for the same security, as evaluated in §11.6.

11.8 End-to-end evaluation

In this section, we evaluate Arx on ShareLaTeX and YCSB.

11.8.1 Evaluation on ShareLaTeX

We evaluate the overhead of Arx using ShareLaTeX [9], 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. The application was collocated with Arx's client proxy.

ShareLaTeX maintains multiple collections in MongoDB corresponding to users, projects, documents, document history, chat messages, etc. We were conservative in estimating the sensitivity of fields in the different collections. Specifically, all information generated from user activity (such as a user's personal information, project contents, chat messages, edit history, etc.) was deemed sensitive, along with associated metadata including timestamps and version numbers. The remaining fields comprised application-specific metadata (e.g. font size, compiler type, etc.) and were assumed to be non-sensitive.

Before every experiment, we pre-loaded the database with 100K projects, 200K users, and each collection with 100K records. Subsequently, using Selenium, multiple clients open up browsers in parallel and collaborate on projects, continuously editing documents and exchanging messages via chat. Fig. 10 shows the maximum throughput of ShareLaTeX in various configurations for both a vanilla deployment against regular MongoDB, and for an Arx deployment. The client proxy is either collocated with the ShareLaTeX application sharing the same four cores, or deployed on extra and separate cores. While the decline in application throughput is significant when the client proxy and ShareLaTeX are collocated, the performance improves drastically when two separate cores are allocated to Arx's client proxy, in which case 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 Netty [5] 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 client threads and 100% CPU load at the application, the reduction in performance owing to Arx is 11%, of which 8% is due to the Arx's proxy, and the remaining 3% due to Arx's encryption and indexing schemes.

Finally, the latency introduced by Arx is modest in comparison to the latency of the application. Under conditions of low stress with 16 clients, performance remains bottlenecked at the application, and the latency added by Arx is negligible, which increases from an average of 257ms per operation to 258ms. At peak throughput with 40 clients, the latency of vanilla ShareLaTeX is 343ms, which grows by 12% to 385ms in the presence of Arx, having marginal impact on user experience.

An important takeaway from these experiments is that 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. Second, even though Arx-RANGE is not cheap, Arx-RANGE is just one operation out of a set of multiple operations Arx runs, with the other operations being faster and overall more common, such as: Arx-EQ and the building blocks in §3.

11.8.2 YCSB Benchmark

Since Arx is a NoSQL database, we also evaluate its overhead on the YCSB benchmark [21] running against the client proxy. We first loaded the database with 1M documents. We annotated all fields as sensitive including the primary key. Hence, the primary key has an Arx-EQ index and the rest of the fields are encrypted with BASE.

Fig. 12 shows the average performance of Arx versus vanilla MongoDB, across different workloads with varying proportions of reads and writes. In the figure, "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 ranges from 3%-9%, based on the workload. Increase in latency due to Arx is also unremarkable—for example, average read latency increases from 2.31ms to 2.43ms under peak throughput, while average update latency increases from 2.36ms to 2.47ms, in the 50% read-50% update workload. Arx's performance on YCSB is high because YCSB conforms to Arx-EQ's optimized special case when fields are unique. In general, this shows that indexing primary keys is fast with Arx.

11.9 Storage

Arx increases the amount of data stored in the database for the following reasons: (1) ciphertexts are larger than

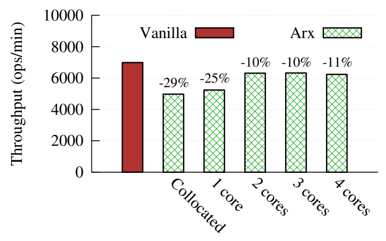


Figure 10: ShareLaTeX performance with Arx's client proxy on varying cores

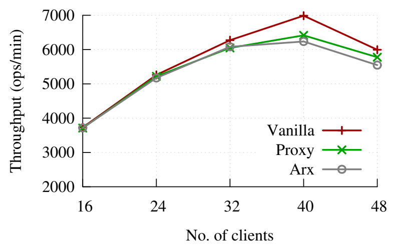


Figure 11: ShareLaTeX performance with increasing no. of client threads

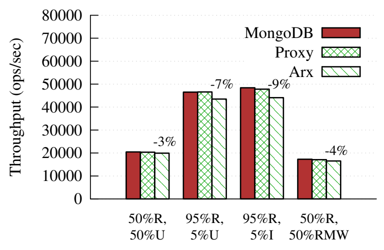


Figure 12: YCSB throughput for different workloads.

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 Arx-EQ indexing. Further, Arx-RANGE indices are larger than regular B+Trees, because each node in the index tree stores garbled circuits. Vanilla ShareLaTeX with 100K documents per collection required 0.52GB of storage in MongoDB, with an additional 45.4 MB for indices. With Arx, the data storage increased by $\sim$1.4$\times$ to 0.72GB for the reasons described above. The application required three compound Arx-RANGE indices, which together occupied 8.4GB of memory at the server proxy while indices maintained by the database occupied 45.8MB. This resulted in a net increase of $\sim$16$\times$ at the DB server, for storing the encrypted data along with Arx's indices. However, there remains substantial scope for optimizing the size of the indices in our implementation. For example, a serialized dump of the three Arx-RANGE indices occupied 3.1GB of memory. Moreover, our choice of workload did not include large data items such as images or videos, which typically would not require indexing by Arx-RANGE. In such cases, the storage overhead imposed by Arx would proportionately decrease.

Finally, the application required two Arx-EQ indices for which counter maps were maintained at the client proxy, which in turn occupied 8.6MB of memory, illustrating that Arx-EQ imposes modest storage overhead at the application server. Moreover, the values inserted into the counter maps were distinct; in case of duplicates, the memory requirements would be proportionately lower.

12 Related Work

Encrypted databases. Early approaches [30] used heuristics instead of encryption schemes with provable security. Practical encrypted databases such as [45, 13, 48, 54], in some cases, either don't support useful functions or use weak encryption schemes that reveal sensitive information [36, 39]. Searchable-encryption based databases [20, 23] are more secure than these approaches (and comparable in security to Arx), but are too restricted in functionality. They return a superset of the results for a range query and do not support `order by limit` queries, a common query used for pagination. Moreover, they do not support aggregates over a range because the range identifies a superset of the relevant documents, yielding an incorrect aggregate. Similarly, concurrent work [31, 34] support equality-based queries and are too restricted in functionality. They do not support range, order-by-limit, or aggregates over range queries, and the former does not support updates and inserts, which are crucial for many applications. Other encrypted databases [41] rely on certain trust assumptions at the server: the server is split in two parts that do not collude with each other. Approaches using trusted hardware promise full functionality [15, 50, 14], but they rely on trusted hardware at the server.

Work related to Arx-EQ. Arx-EQ falls in the general category of searchable-encryption schemes [52, 19] and builds on insights from this literature. While there are many schemes proposed in this space [19], none of them meet the following desired security and performance from a database index. First, a searchable encryption scheme should not only encrypt the database with IND-CPA security, but also, when inserting a field, the access pattern should not inform the attacker of what other fields it equals, and an old search token should not allow searching on newly inserted data. Second, inserts, updates and deletes should be efficient and should not cause reads to become slow. Arx-EQ meets all these goals. Perhaps the closest work to Arx-EQ is [20]. This scheme uses revocation lists for delete operations, which adds significant complexity and overhead, as well as leaks more than our goal in Arx: an old search token can be used to search new data, and the revocation lists leak various metadata.

Work related to Arx-RANGE. We provide an extensive survey of the literature related to Arx-RANGE in [16]. The main highlight is order-preserving encryption [17, 18, 44], which is efficient, but leaks a significant amount of information [36, 39].

13 Conclusion.

Arx is the first practical and functionally rich database system that encrypts the database with strong encryption schemes and computes on the encrypted database. We showed that Arx supports a rich set of applications and incurs a modest performance overhead.

Acknowledgements

The authors would like to thank Riyaz Faizullabhoy for discussing early versions of Arx as part of a class project.

References

[1] Budget Manager. https://github.com/kdelemme/budget-manager/.

[2] Chino.io: Security and Privacy for Health Data in the EU. https://chino.io/.

[3] iQrypt: Encrypt and query your database. http://iqrypt.com/.

[4] Leanote. https://leanote.com/.

[5] Netty Project. http://netty.io/.

[6] NodeBB. https://nodebb.org/.

[7] PencilBlue. https://pencilblue.org/.

[8] Redux chat. https://github.com/raineroviir/react-redux-socketio-chat/.

[9] ShareLatex. https://www.sharelatex.com/.

[10] UNCAP: Ubiquitous iNteropable Care for Ageing People. http://www.uncap.eu/.

[11] ZeroDB. http://zerodb.io/.

[12] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, 1989.

[13] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A secure coprocessor for database applications. In *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications (FPL)*, Porto, Portugal, 2013.

[14] S. Bajaj and R. Sion. TrustedDB: A trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, Athens, Greece, 2011.

[15] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[16] T. Boelter, R. Poddar, and R. A. Popa. A secure one-roundtrip index for range queries. Cryptology ePrint Archive, Report 2016/568, 2016. http://eprint.iacr.org/.

[17] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, Cologne, Germany, 2009.

[18] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proceedings of the 31st International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, 2011.

[19] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 2014.

[20] 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 *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, Cascais, Portugal, 2011.

[22] M. Egorov and M. Wilkison. ZeroDB white paper. *CoRR*, abs/1602.07168, 2016. http://arxiv.org/abs/1602.07168.

[23] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*, Vienna, Austria, 2015.

[24] S. Garg, S. Lu, and R. Ostrovsky. Black-box garbled RAM. In *Proceedings of the 56th Annual Symposium on Foundations of Computer Science (FOCS)*, 2015.

[25] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC)*, Bethesda, MD, 2012.

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

[27] O. Goldreich. *The Foundations of Cryptography*. Cambridge University Press, 2001.

[28] T. Greene. Biggest data breaches of 2015, 2015. http://www.networkworld.com/article/3011103/security/biggest-data-breaches-of-2015.html.

[29] P. Grofig, M. Haerterich, I. Hang, F. K. and Mathias Kohler, A. Schaad, A. Schroepfer, and W. Tighzert. Experiences and observations on the industrial implementation of a system to sear ch over outsourced encrypted data. In *Lecture Notes in Informatics, Sicherheit*, 2014.

[30] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, WI, 2002.

[31] S. Kamara and T. Moataz. Sql on structurally-encrypted databases. Cryptology ePrint Archive, Report 2016/453, 2016. http://eprint.iacr.org/.

[32] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.

[33] 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. *CoRR*, 2014.

[34] 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/.

[35] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Proceedings of the 8th International Conference on Cryptology and Network Security (CANS)*, 2009.

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

[37] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security*, 13(4), 2010.

[38] R. Merkle. *Secrecy, authentication and public key systems / A certified digital signature*. PhD thesis, Stanford University, 1979.

[39] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, 2015.

[40] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, Prague, Czech Republic, 1999.

[41] 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 *Proceedings of the 35th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2014.

[42] F. Pennic. Anthem suffers the largest healthcare data breach to date, 2015.

[43] R. A. Popa. *Building Practical Systems that Compute on Encrypted Data*. PhD thesis, MIT, 2014.

[44] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, 2013.

[45] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[46] Google. Encrypted BigQuery client. https://github.com/google/encrypted-bigquery-client.

[47] Microsoft SQL Server 2016. Always encrypted database engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[48] CipherCloud. Cloud data protection solution. http://www.ciphercloud.com.

[49] Cloud Threat Intelligence. Skyhigh cloud security labs, Skyhigh Networks. https://www.skyhighnetworks.com/.

[50] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[51] J. Sherry, C. Lan, R. P. Ada, and S. Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *Proceedings of the 26th ACM Special Interest Group on Data Communication Conference (SIGCOMM)*, 2015.

[52] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (IEEE S&P)*, Oakland, CA, 2000.

[53] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2012.

[54] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, Riva del Garda, Italy, 2013.

[55] A. C. Yao. How to generate and exchange secrets (extended abstract). In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*, 1986.

[56] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2015.

[57] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, 2015.

Appendices

A Formal treatment of security

In this appendix, we provide security definitions and proofs for the security of Arx. In this paper, we focus on the system, so we include most of the formal cryptographic treatment in a separate report: [16]. This report focuses on the cryptographic treatment of Arx-RANGE as well as presents other contributions of this scheme. It also introduces notation, definitions and proofs we need here, so for conciseness, we will simply refer to them when needed.

A.1 Persistent attacker

We begin with the persistent attacker because the snapshot attacker is an instantiation of the snapshot attacker, and the guarantee for the snapshot attacker will follow.

As is typical when defining security of such protocols as Arx, we define a leakage function that the attacker sees (e.g., size of the database, number of values in the result), and prove that the attacker does not learn anything else. We prove non-adaptive semantic security with respect to these leakage functions. Non-adaptive security suffices in this case because, in our threat model, an attacker does not have the ability to issue queries and users, who issue queries, do not see the encrypted database. However, one can extend the proofs to adaptive security with some (less efficient) tweaks to our schemes.

A few preliminaries are needed to understand our notation and notions. Please refer to [16].

We now define the leakage profile of Arx in the face of a persistent attacker.

Definition 1. *The schema and sizing leakage is, given a database DB,*

- *the schema: the name of collections, the number of documents in each collection and which fields they contain (but not the content of the fields), unique or size information per field declared by the developer, indices built by the developer,*
- *size: the size of each field, and*
- *the query patterns (list of operations per field, but no constants).*

The leakage for query Q, in general, is the entire query except for the constants in the query, the time when the query is issued, which documents and fields are accessed (removed, inserted, read, updated), but not their content.

For EQ,

- *upon search on field f for constant c: whether the search token equals any previous search token for field f, the items that match the constant c from field f (including the items that were deleted and observed by the attacker),*

- *upon insert: nothing besides above with respect to old search tokens observed by attacker, and*
- *upon update: as above with respect to old search tokens observed by attacker.*
  *For Arx-*RANGE *and Arx-*AGG*, as specified in [16].*
  *For Arx-*EQ*:*
- *upon search on field f for constant c: the items that match the constant c from field f and the counter of each (indicating insert order), including counters and matches for elements that were deleted since the last cleanup, such items were in deleted documents,*
- *upon insert, delete, update: nothing additional, and*
- *upon cleanup: same as search.*

*For Arx-*JOIN*: the same information as Arx-*EQ *or Arx-*RANGE*, depending on which Arx-*JOIN *was built on, as well as leakage as in* EQ *for the foreign key, where each match identifies a primary key.*

Theorem 2. *The Arx protocol is non-adaptively semantically secure (see definition in [16]) with leakage profile defined in Def. 1, under the assumptions that: AES is a pseudorandom permutation and the Paillier assumption [40]. In some places, for efficiency, we additionally introduce the random oracle assumption.*

The security definition captures the honest-but-curious server as defined by the protocol so we assume the server does not maintain history-dependent data: however, we note again as in §4.4, that the real server might maintain some history-dependent information which is hard to control (e.g., the language runtime's placement of data in memory).

Proof Sketch. We provide a proof sketch leaving a detailed proof for the next version of this report. Nevertheless, the proof for Arx-RANGE, the biggest part of this component is fully presented in [16], and the remainder is mostly mechanical.

We can treat each scheme per field in Arx separately because the client uses a separate key for each one. Hence, let's discuss each scheme in part.

The building blocks, BASE, EQ and AGG, already exist and are proven secure [40, 51, 20]; their security guarantees satisfy the leakage in Def. 1.

The more complex scheme is Arx-RANGE, for which we prove adherence to the desired security guarantees in [16].

The proofs for Arx-EQ and Arx-JOIN are straightforward and will be included in the next version of this report.

□

A.2 Snapshot attacker

Proof of Theorem 1. The guarantee for the snapshot attacker follows from Theorem 2. The per-query leakage is removed from the leakage function: recall that the snapshot attacker 's information is a snapshot of the database.

As promised, the remaining leakage is only the schema and sizing information, and it contains no data content. In fact, one can simulate efficiently the content of the database (collections and indices) only from schema and sizing information.

Definition 2. *The leakage is, given a database DB,*
- *the schema: the name of collections, the number of documents in each collection and which fields they contain (but not the content of the fields), unique or size information per field declared by the developer, indices built by the developer,*
- *size: the size of each field, and*
- *the query patterns (list of operations per field, but no constants).*

The fact that the decryption key is never sent to the server is by design. □