Arx: An Encrypted Database using Semantically Secure Encryption

(Extended version)*

Rishabh Poddar
UC Berkeley
rishabhp@berkeley.edu

Tobias Boelter
UC Berkeley
t.b@berkeley.edu

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

ABSTRACT

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

1. INTRODUCTION

Due to numerous data breaches [7, 26], the public concern over privacy and confidentiality is likely at one of its peaks today. In recent years, encrypted databases [5,71,76,87] (EDBs) have emerged as a promising direction towards achieving both confidentiality and functionality: queries run on encrypted data. CryptDB [76] demonstrated that such an approach can be practical and can support a rich set of queries; it then spurred a rich line of work including Cipherbase [5] and Monomi [87]. The demand for such systems is demonstrated by the adoption in industry such as in Microsoft's SQL Server [61], Google's Encrypted Big Query [35], and SAP's SEEED [36] amongst others [21, 41, 47, 82]. Most of these services are *NoSQL databases* of various kinds showing that a certain class of encrypted computation suffices for many applications.

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

Leakage from data is leakage from an encrypted database, *e.g.*, *relations among data items*. In order to execute queries efficiently, the EDBs above use a set of encryption schemes some of which are *property-preserving* by design (denoted PPE schemes), *e.g.*, order-preserving encryption (OPE) [9, 10, 75] or deterministic encryption (DET). OPE and DET are designed to reveal the *order* and the *equality relation* between data items, respectively, to enable fast order and equality operations. However, while these PPE schemes confer protection in some specific settings, a series of recent attacks [27, 39, 64] have shown that given certain auxiliary information, an attacker can extract significant sensitive information from the order and equality relations revealed by these schemes. These works demonstrate *offline* attacks in which the attacker obtains a PPE-encrypted database and analyzes it offline. For example, such attackers include hackers stealing a snapshot of the database, or government subpoenas.

Leakage from queries refers to what an (*online*) attacker can see during query execution. This includes all observable state in memory, along with which parts of the database are touched (called access patterns), including *which (encrypted) rows* are returned and *how many*, which could be exploited in certain settings [16, 37, 46, 51]. Unfortunately, hiding the leakage due to queries is very expensive as it requires oblivious protocols (*e.g.*, ORAM [85]) to hide access patterns, along with aggressive padding [63] to hide the result size. For instance, Naveed [63] shows that in some cases it is more efficient to stream the database to the client and answer queries locally than to run such a system on a server.

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

We propose Arx, a practical and functionally rich database system that takes an important step in this direction by *always* keeping the data encrypted with semantically secure encryption schemes. Semantic security implies that no information about the data is leaked (other than its size and layout), preventing the aforementioned offline attacks on a stolen database. This model is particularly suitable for protecting data against subpoenas, in which case there is only leakage from data, and no leakage from queries.

For an online attacker, Arx incurs pay-as-you-go information leakage: the attacker no longer learns the frequency count or order relations for *every* value in the database, but *only for data involved in the queries it can observe*. In the worst case (*e.g.*, if the attacker observes many queries over time), this leakage could add up to the leakage of a PPE-based EDB, but in practice it may be significantly more secure for short-lived online attackers. As prior work points out [11, 19, 56], this model fits the "well-intentioned cloud provider" which uses effective intrusion-detection systems to prevent attackers from observing and logging queries over time, but fears "steal-and-run" attacks. For example, Microsoft's Always Encrypted [61] advocates this model.

Unfortunately, there is little work on such EDBs, with most work focusing on PPE-based EDBs. The closest to our goal is the line of work by Cash *et al.* [17, 18] and Faber *et al.* [29], which builds on searchable encryption. As a result, these schemes are significantly limited in functionality—they

*A shorter version of this work was published in *Proceedings of the VLDB Endowment,* Vol. 12, No. 11, 2019.

do not support common queries such as order-by-limit, aggregates over ranges, or joins—and are also inefficient for write operations (*e.g.*, updates, deletes). Furthermore, for certain *online* attackers, these systems have some extra leakage not present in PPEs, as we elaborate in §12. To replace PPE-based EDBs, we need a solution that is always at least as secure as PPE-based EDBs.

Overall, by exclusively using semantically secure encryption, Arx prevents the offline attacks above [27, 39, 64] from which PPE-based EDBs suffer. For online attackers, Arx is *always* either more or as secure as PPE-based EDBs.

1.1 Techniques and contributions

A simple attempt to protect against offline attacks could be to keep the data encrypted at rest in the database and only decrypt it when it is in use. However, such an approach directly leaks the secret key even to a short-lived attacker who succeeds in taking a well-timed snapshot of memory. Worse, the key itself would be directly vulnerable to a subpoena, akin to not encrypting the database at all. A better alternative might be to consider a hybrid design that uses a PPE-based EDB, but employs a second layer of encryption for data at rest on the disk. However, an attacker who similarly obtains the decryption key gets access to the PPE-encrypted data, rendering the second layer of encryption useless and leaking more information than our goal in Arx.

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

To enable range queries, **ArxRange** builds a tree over the relevant keywords, and stores at each node in the tree a *garbled circuit* for comparing the query against the keyword in the node [33, 90]. Our tree is history-independent [4] to reduce structural leakage. The main challenge with ArxRange is to avoid interaction (*e.g.*, as needed in BlindSeer [72]) at every node on a tree path. To address this challenge, Arx draws inspiration from the theoretical literature on Garbled RAM [30]. Arx chains the garbled circuits on a tree in such a way that, when traversing the tree, a garbled circuit produces input labels for the child circuit to be traversed next. Thereby, the whole tree can be traversed in a single round of interaction. For security, each such index node may only be used once, so ArxRange essentially *destroys itself for the sake of security*. Nevertheless, only a logarithmic number of nodes are destroyed per query, and Arx provides an efficient repair procedure.

ArxEq builds a regular database index over encrypted values by embedding a counter into repeated values. This ensures that the encryption of two equal values is different and the server does not learn frequency information. To search for a value v, the client provides a small token to the server, which the server expands into many search tokens for all occurrences of v. ArxEq provides forward privacy [13], preventing old tokens from being used to search new data.

Building on top of these two indices, Arx speeds up aggregations by transforming them into tree lookups via **ArxAgg**. and supports foreign-key joins with **ArxJoin**.

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

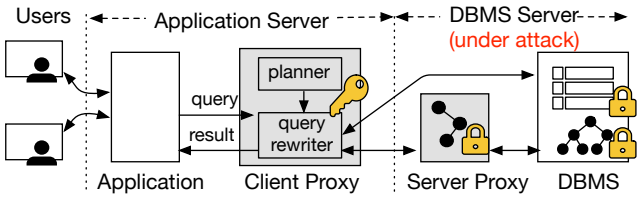

Figure 1: Arx's architecture: Shaded boxes depict components introduced by Arx. Locks indicate that sensitive data at the component is encrypted.

Finally, we designed **Arx's architecture** so that it is amenable to adoption. Two lessons [74] greatly facilitated the adoption of the CryptDB system: do not change the DB server and do not change applications. Arx's architecture, presented in Fig. 1, accomplishes these goals. The difference over the CryptDB architecture [76] is that it has a server-side proxy, a frontend for the DB server. The server proxy converts encrypted processing into regular queries to the DB, allowing the DB server to remain unchanged.

We **implement and evaluate Arx** on top of MongoDB, a popular NoSQL database. We show that Arx supports a wide range of real applications such as ShareLaTeX [81], the Chino health data platform [20], NodeBB forum [67], and Leanote [54] amongst others. In particular, Chino is a cloud-based platform that serves the European medical project UNCAP [88]. Chino provides a MongoDB-like interface to medical web applications (running on hospital premises) but currently operates on *plaintext* data. The project's leaders confirmed that Arx's model fits Chino's setup perfectly. Finally, we also show that Arx's overheads are modest: it impacts the performance of ShareLaTeX by 11% and the YCSB [23] benchmark by 3–9%.

2. OVERVIEW

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

2.1 Architecture

Arx considers the model of an application that stores sensitive data at a database (DB) server. The DB server can be hosted on a private or public cloud. Fig. 1 shows Arx's architecture. The application and the database system remain unmodified. Instead, Arx introduces two components between the application and the DB server: a trusted client proxy and an untrusted server proxy. The client proxy exports the same API as the DB server to the application so the application does not need to be modified. The server proxy interacts with the DB server by invoking its unmodified

API (*e.g.*, issuing queries); in other words, the server proxy behaves as a regular client of the DB server. Unlike CryptDB, Arx cannot use user-defined functions instead of the server proxy because the proxy must interact with the DB server multiple times per client query.

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

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

2.2 Threat Model

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

Arx considers *passive* (honest-but-curious) server attackers: the attackers examine server-side data to glean sensitive information, but follow the protocol as specified, and do not modify the database or query results. The active attacker is interesting future work, that can potentially leverage complementary techniques [45, 57, 60, 93]. Further, in the Arx model, an attacker cannot inject any *new* queries as she does not have access to the client application or to the secret keys at the client proxy, but only to the server.

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

2.3 Security guarantees

Arx has different guarantees for the two attackers.

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

For this attacker, Arx provides strong security guarantees revealing nothing about the data beyond the schema and size information (the number of collections, documents, items per field, size of items, which fields have indices and for what operations). The contents of the database (collections and indices) are protected with *semantically secure* encryption, and the decryption key is never sent to the server. In particular, Arx prevents the offline attacks of [27, 39, 64] from which PPE-based EDBs suffer. In PPE-based EDBs, the attacker readily sees the order or the frequency of all the values in the database for PPE-encrypted fields. This is significantly less secure than the semantically secure schemes in Arx, in which the attacker does not see such relations.

Online attacker. The online attacker additionally watches *queries* and their execution. Arx hides the parameters in the queries, but not the operations performed. In particular, Arx does not hide metadata (*e.g.*, query identifiers and timestamps) or access patterns during execution (*e.g.*, which positions in the database or index are accessed/returned and how many). Prior work has shown that, if an attacker can observe such information from many queries and if certain assumptions and conditions hold, the attacker can reconstruct data [16, 37, 46, 51]. Since each query in Arx reveals only a limited amount of metadata, the sooner an attacker is detected (*e.g.*, the fewer queries they observe), the less information they are able to glean.

For this attacker, Arx aims to always be more or as secure as PPE-based EDBs. Indeed, for all operations, Arx's leakage is always upper-bounded by the leakage in PPE-based EDBs. This is non trivial: for example, a prior EDB aiming for semantic security [29] is not always more secure than PPE-based EDBs, as we explain in §12.

Security definition. To quantify the leakage to online attackers, we provide a security definition for Arx and its protocols that is parameterized by a *leakage profile* $\mathcal{L}$, which is a function of the database and the sequence of the queries issued by the client. Our security definition is fairly standard, and similar to prior work [18, 29].

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

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

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

where λ is the security parameter, z is auxiliary information, and $\mathsf{negl}(\lambda)$ a negligible function in λ.

We formalize the leakage profile of Arx and its protocols in §9, and provide proofs of security with respect to Definition 1 for non-adaptive adversaries (who select the query sequence beforehand) in Appendix B. Informally, the leakage for ArxEq includes the list of queries that search for the same keyword; for ArxRange, the leakage includes the ranks of the bounds in the range query.

2.4 Admin API

We describe the API exposed by Arx to application admins. The admin can take an existing application and enhance it

with Arx annotations. Arx's planner, located at the client proxy, uses this API to decide the data encryption plan, the list of Arx indices to build, and a query execution plan for each query pattern.

Following the example of Microsoft's SQL Server [61] and Google's Encrypted BigQuery [35], Arx requires the admin to declare what operations will run on the database fields. By default, Arx considers *all* the fields in the database to be sensitive, unless specified otherwise. To use Arx, the admin specifies the following information during system setup:

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

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

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

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

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

2.5 Functionality

We now describe the classes of read and write queries that Arx can execute over encrypted data. As we show in §10, this functionality suffices for a wide range of applications.

Read queries. Arx supports queries of the form:

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

doc denotes a document and [agg *doc*] aggregations over documents, which take the form $\sum Func(doc)$. $\sum$ can be any associative operator and *Func* an arbitrary, efficiently-computable function. Examples include sum, count, sum of squares, min, and max. More aggregations can be computed with minimal postprocessing at the client proxy by combining a few aggregations, such as average or standard deviation.

The predicate *clause* is $\left[\wedge_i \mathsf{op}(f_i)\right]$ where $\mathsf{op}(f_i)$ denotes equality/range operations over a field f_i such as $=, \geq$ and $<$.

In addition to these queries, Arx supports a common form of joins—namely, foreign-key joins—which we describe in §7.

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

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

3. ENCRYPTION BUILDING BLOCKS

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

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

EQ enables equality checks using a searchable encryption scheme similar to existing work [17, 52]. The EQ scheme we use is as follows. $\mathsf{EQEnc}_k(v) = (\mathsf{IV}, \mathsf{AES}_{\mathsf{KDF}_k(v)}(\mathsf{IV}))$, where IV is a random value and KDF is a key derivation algorithm based on AES. To search for a word w, $\mathsf{EQToken}_k(w)$ computes the token as $\mathsf{tok} = \mathsf{KDF}_k(w)$. To identify if the token matches an encryption, the server proxy combines tok with IV and checks to see if it equals the ciphertext: $\mathsf{EQMatch}((\mathsf{IV}, x), \mathsf{tok}) = (\mathsf{AES}_{\mathsf{tok}}(\mathsf{IV}) \stackrel{?}{=} x)$. 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 ArxEq index.

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

AGG enables addition using the Paillier scheme [70].

4. ArxRange & ORDER-BASED QUERIES

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

4.1 Strawman

We begin by presenting a helpful but inefficient strawman, that corresponds to the protocols in mOPE [75] and the startup ZeroDB [28]. For simplicity, consider the index to be a binary tree (instead of a regular B+ tree). To obtain the desired security, each node in the tree is encrypted using a standard encryption scheme. Because such encryption is not functional, the server needs the help of the client to traverse the index. To locate a value a in the index, the server and the client interact: the server provides the root node to the client, the client decrypts it into a value v, compares v to a, and tells the server whether to go left or right. The server then provides the relevant child to the client, and the procedure

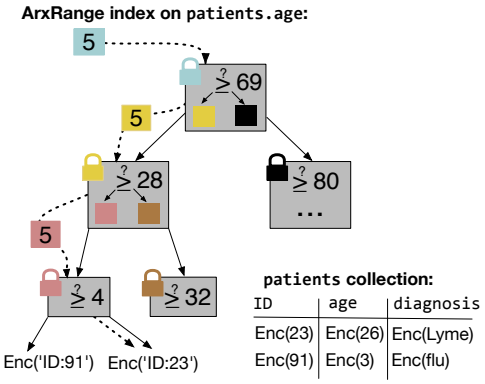

Figure 2: ArxRange example. Enc is encryption with BASE.

repeats until it reaches a leaf. As a result, each level in the tree requires a roundtrip, making the process inefficient.

4.2 Non-interactive index traversal

ArxRange enables the server to traverse the tree by itself. Say the server receives $\mathsf{BASE}_k(a)$ and must locate the leaf node corresponding to a. To achieve this goal, the server must be able to compare $\mathsf{BASE}_k(a)$ with the encrypted value at a node, say $\mathsf{BASE}_k(v)$. Inspired from the theoretical literature on garbled RAM [30, 31], we store **a garbled circuit at each tree node** that performs the comparison, while hiding a and v from the attacker.

A garbling scheme is a set of algorithms (Garble, Encode, Eval) [33, 90]. Using a garbling scheme, the client can invoke the algorithm Garble on a boolean circuit f to obtain a *garbled* version F of the circuit, along with some secret encoding information e. Given an input a, the client can run $\mathsf{Encode}(e, a)$ to produce an encoding e_a corresponding to the input. Then, the server can run $\mathsf{Eval}(F, e_a)$ and obtain the output $y = f(a)$. The security of garbled circuits guarantees that the server learns *nothing* about a or the data hardcoded in f other than the output $f(a)$ (and the size of a and f). This guarantee holds as long as the garbled circuit is used *only once*. That is, if the client provides two encodings e_a and e_b using the same encoding information e to the server, the security guarantees no longer hold. Hence, our client provides at most one input encoding for each garbled circuit.

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

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

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

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

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

4.3 The database index

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

First, the shape of the index should not leak information about the order in which the data was inserted. Hence, we use a history-independent treap [4, 62] instead of a regular search tree. This data structure has the property that its shape is independent of the insertion or deletion order.

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

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

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

Updating the index. For inserts and deletes, the server traverses the index to the appropriate position, performs the

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

Some update or delete queries may first perform a filter on a field using a different index, but also requiring deletes from an ArxRange index as a result. To support this case, we maintain an encrypted *backward* pointer from the document to the corresponding node in the tree. The backward pointers enable the identification of these nodes *without* having to traverse the ArxRange index. Decryption of these pointers requires a single round of interaction with the client proxy.

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

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

4.4 Optimizations

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

5. ArxEq & EQUALITY QUERIES

The ArxEq index enables equality queries and builds on insights from the searchable encryption literature [12], as explained in §12. We aim for ArxEq to be *forward private*, a property shown to increase security significantly in this context [13]: the server cannot use an old search token on newly inserted data. We begin by presenting a base protocol that we improve in stages.

5.1 Base protocol

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

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

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

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

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

where H is a cryptographic hash (modeled as a random oracle). This encryption provides semantic security because $\mathsf{EQunique}(v)$ is a deterministic encryption scheme which becomes randomized when combined with a unique salt per value v: $\mathsf{ctr}[v]$. This encryption is not decryptable, but as discussed in §8.2, Arx encrypts v with BASE as well. The document is then inserted into the database.

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

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

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

5.2 Reducing the work of the client proxy

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

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

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

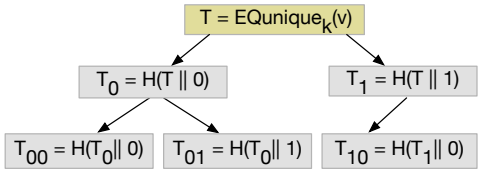

Figure 3: Search token tree.

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

set constitute the search token. The covering set can be easily deduced from the binary representation of $\mathsf{ctr}[v] - 1$.

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

5.3 Updates

We have already discussed inserts. For deletes, Arx simply deletes the document. An update is a delete followed by an insert. As a result, encrypted values for some counters will not return matches during search. This does not affect accuracy, but as more counters go missing, it affects throughput because the DB server wastes cycles looking for values with no matches. It also provides a small security leakage because a future search leaks how many items were deleted. As a result, ArxEq runs a cleanup procedure after each deletion. As a performance optimization, one can run a cleanup procedure when a search query for a value v indicates more than a threshold of missing counters, relaxing security slightly.

Cleanup. The server proxy tells the client proxy how many matches were found for a search, say ct. The client proxy updates $\mathsf{ctr}[v]$ 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 tokens to the server, which replaces the fields found matching with these.

5.4 Counter map

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

Counter map at server. The counter map can be stored encrypted at the server. An entry of the sort $v \to \mathsf{ct}$ becomes $\mathsf{EQunique}_{k_1^*}(v) \to \mathsf{EQunique}_{k_2^*}(\mathsf{ct})$, 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. This scheme satisfies Arx's security goal in §2.3: a stolen database remains encrypted with semantic security and leaks nothing except size information.

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

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

6. ArxAgg & AGGREGATION QUERIES

We now explain Arx's aggregation over the encrypted indices. It is based on AES and is faster than homomorphic encryption schemes like Paillier [70]. Many aggregations happen over a range query, such as computing the average days in hospital for people in a certain age group. Arx computes the average by computing sum and count at the server, and then dividing them at the client proxy. Hence, let's focus on the query: `select sum(daysAdmitted) from patients where` $70 \leq$ `age` ≤ 80.

The idea behind aggregations in Arx is inspired from literature on authenticated data structures [57]. This work targets integrity guarantees (not confidentiality), but interestingly, we use it for computations on encrypted data. Consider the ArxRange index in Fig. 2 built on `age`. At every node N in the tree, we add the *partial aggregate* corresponding to the subtree of N. For the query above, N contains a partial sum of `daysAdmitted` corresponding to the leaves under N. The root node thus contains the sum of all values. This value is stored encrypted with BASE.

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

In the case of (i) inserting/deleting a document, or (ii) 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 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 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 ArxEq 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 AGG . However, the number of such cases is reduced significantly.

7. JOINS USING ArxJoin

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

```
select [...] from C1 join C2
       on C1.fkey = C2.ID
       where clause(C1) [and eq(C2)]
```

where `C1` and `C2` are the two collections being joined, `fkey` is the foreign key in `C1` pointing to the primary key `ID` in `C2`, and *clause* is a predicate that can be evaluated using an ArxEq or ArxRange index. The query may additionally filter the joined documents in `C2` using equality operations, denoted by eq(`C2`).

ArxEq-based joins. Consider an example with collection `C2` having a primary key `ID`, and collection `C1` having a field `age`

with ArxEq, and `diagnosis` which is a foreign key pointing to `C2.ID`.

The primary key in the secondary collection `C2.ID` is encrypted with EQunique as before. Consider inserting a document with age 10 and diagnosis 'flu' in `C1`, and let's discuss how the client proxy encrypts this pair. Since foreign keys are not unique, `C1.diagnosis` is encrypted with BASE. Additionally, to perform the join, the client proxy computes an *encrypted pointer* for `C1.diagnosis`. When decrypted, this pointer will point to the appropriate encrypted `C2.ID`. Instead of using one key for ArxEq, 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 (BASE(10), t_1, BASE('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 C1 join C2 on C1.diagnosis = C2.ID where C1.age = 10`. To execute this query, the server proxy computes t_1 and t_2 for the age of 10, as usual with ArxEq. 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 `C2`, and the server simply does a lookup in `C2`.

The `where` clause of the query may additionally filter documents in `C2` using an equality predicate, *e.g.*, `where age = 10 and C2.symptom = 'fever'`. To filter the joined documents by `symptom`, Arx employs the EQ protocol for equality checks as described in §3. Note that this additional filtering cannot make use of an index; hence, it is restricted to equality predicates and may not contain range operations.

ArxRange-based joins. Arx employs a different strategy in case the `where` clause of the join query requires an ArxRange index for execution, *e.g.*, `where C1.age > 10`. In such a scenario, ArxJoin's tokens for `C1.age` cannot be computed as described above.

Instead, the foreign key values encrypted with BASE are directly added to the nodes of the ArxRange index over `C1.age`, which already contain the encrypted primary keys of documents in `C1` (as described in §4.3). 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 ArxRange. 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 `C2`, and performs the join. Note that this strategy does not bring any extra round trips between the proxies.

Updates. The semantics of updates remain unchanged in the presence of ArxJoin. Updates to the foreign key `C1.fkey` simply update the underlying index, ArxEq or ArxRange. Updates to `C2.ID` are also straightforward, and do not affect the pointers in `C1`. This is because `ID` is a primary key in `C2` and its values are unique.

8. ARX'S PLANNER

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

8.1 Index planning

Before deciding what index to build, note that ArxRange and ArxEq support *compound* indices, which are indices on multiple fields. For example, an index on (`diagnosis`, `age`) enables a quick search for `diagnosis = 'flu' and age` $\geq$ `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 admin expects: if she specified an index over certain fields, then the time to execute queries on those fields should be logarithmic and not require a linear scan. At the same time, we would like to build few indices to avoid the overhead of maintaining and storing them. Deciding what indices to build automatically is challenging because (1) there is no direct mapping from regular indices to Arx's indices, and (2) Arx's indices introduce various constraints, such as:

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

All these are further complicated by the fact that the application admin can explicitly specify certain fields to be nonsensitive (as described in §2.4), and simultaneously declare compound indices on a mixture of fields, both sensitive and not. Similarly, queries can have both sensitive as well as nonsensitive fields in a `where` clause.

As a consequence of our performance goal and these constraints, interestingly, there are cases when Arx builds an ArxRange index on a composition of a nonsensitive and a sensitive field. Consider, for example, that the admin 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 admin 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 ArxRange 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 admin 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 ArxRange index returns a large number of matches. To deliver the expected performance, Arx builds a composite ArxRange 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 runs in two stages: per-query processing and global analysis. Only the `where` clauses (including order-by-limit operations) matter here. The first stage of the planner treats sensitive and nonsensitive fields equally. For clarity, we use two query patterns as examples. Their `where` clauses are: W_1: "$a =$ `and` $b =$", W_2: "$x =$ `and` $y \geq$ `and` $z =$". The indices specified by the admin are on x and (a, b).

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

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

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

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

One can see why our planner maintains the asymptotic performance of the admin's index specification: each expression that was sped up by an index remains sped up. In §10, we show that the number of extra indices Arx builds is modest and does not blow up in practice.

8.2 Data layout

Next, laying out the encryption plan is straightforward:

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

9. SECURITY ANALYSIS

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

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

Protocol (e)	Operation	$\mathcal{L}_e(\mathsf{DB}, \mathsf{q})$
EQ (§3)	`where` $field = w$	$\mathsf{sp}(w), \mathsf{Hist}(w)$
	`insert`	$\mathsf{sp}(w)$
	`delete`	–
ArxEq (§5)	`where` $field = w$	$\mathsf{sp}(w), \mathsf{Hist}(w)$
	`cleanup` (§5.3)	$\mathsf{sp}(w), \mathsf{Hist}(w)$
	`insert, delete`	–
ArxRange (§4)	`where` $a \leq field \leq b$	$\mathsf{rk}(a-1), \mathsf{rk}(b)$
	`orderby limit` ℓ	ℓ
	`insert, delete` v	$\mathsf{rk}(v)$
ArxJoin (§7)	the same information as ArxEq or ArxRange, depending on which ArxJoin was built on, as well as leakage as in EQ for the foreign key, where each match identifies a primary key.	

Figure 4: Query leakage in Arx's protocols.

9.1 Preliminaries

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

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

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

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

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

$$\mathbb{S}(\mathsf{T}_i) = (i, \{\mathsf{f}_j, \|\mathsf{f}_j\|, \mathsf{plan}(\mathsf{f}_j), \mathsf{shape}(I)\ \forall I \in I(\mathsf{f}_j)\}_{j=1}^m).$$

Here, $\mathsf{plan}(\mathsf{f}_j) = \{\mathsf{e} \in \mathbb{E}\}$ is the set of protocols maintained on field f_j. The schema of a database DB is then given by $\mathbb{S}(\mathsf{DB}) = \bigcup_i \mathbb{S}(\mathsf{T}_i)$.

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

As an example, for the query `select * from patients where` $1 \leq$ `age` < 5, we have $\mathsf{T} =$ `patients`, $\mathsf{qtype} =$ `read`, $\mathsf{pred} = (\mathsf{age}, <)$, $\mathsf{params} = (1, 5)$, and $\mathsf{D} = \phi$.

Note that our definition of a query consists only of a single predicate for simplicity of exposition. We model queries with multiple predicates as a list of single-predicate queries; as a result, our leakage definitions are an upper bound on the actual query leakage.

For insert queries, we further model the insert to include additional operations for fields that have one or more indices maintained on them, one operation per index. For example, consider the following insert query: `insert into patients (name, age) values ('alice', 30)`, and let the `age` field contain both ArxEq and ArxRange indices. In this case, we first insert the document into the database (which creates an identifier for the document). Next, for each field in the document, we insert the corresponding value into the indices maintained on the field: *i.e.*, for the field `age`, we (i) perform an insert into its ArxEq index; and (ii) perform another insert into its ArxRange index. We model delete operations similarly.

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

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

$$\mathbb{P}(\mathsf{T}_i) = \{[(\mathsf{f}_j, \mathsf{op}_j)]_j \mid \mathsf{f}_j \in \mathsf{F}_i\} \quad ,$$

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

9.2 Leakage definitions

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

Definition 2 (Offline leakage of a database). *The leakage of a database* DB *is:*

$$\mathcal{L}_{\mathsf{off}}(\mathsf{DB}) = (\mathbb{S}(\mathsf{DB}), \mathbb{P}(\mathsf{DB}), \{\forall \mathsf{T}_i \ \|\mathsf{T}_i\|\}),$$

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

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

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

Given collection T with a field f that has the EQ or ArxEq protocol maintained on it, we define the *search pattern* of a keyword w (following Bost [13]) as:

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

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

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

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

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

Definition 3 (Online leakage of queries). *Let* $([\mathsf{id}_i]_i, [\mathsf{e}_j]_j) \leftarrow \mathsf{DB}(\mathsf{q})$. *Then, the leakage of a query* q *over database* DB *is:*

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

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

We note that the leakage of ArxEq, as captured in Fig. 4, is similar to that of Sophos [13] and Diana [14].

10. EVALUATION

We now show that Arx supports real applications with a modest overhead.

Implementation. While the design of Arx is decoupled from any particular DBMS, we implemented our prototype for MongoDB 3.0. Arx's implementation consists of ∼11.5K lines of Java, and ∼1800 lines of C/C++ code. We used the Netty I/O framework [66] to implement Arx's proxies. We also disable query logs and query caches to reduce the chance that an offline attacker gets information an online attacker would see, as discussed in Grubbs *et al.* [38].

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

10.1 Functionality

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

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

Application	Examples of fields	Unsupported queries		Number of indices		Total indices	
		Total	Excl. timestamps	ArxEq	ArxRange	Vanilla	Arx
ShareLaTeX [81]	document lines, edits	1	–	12	4	12	16
Uncap (medical) [88]	heart rate, tests	–	–	0	2	2	2
NodeBB (forum) [67]	posts, comments	2	2	13	4	12	17
Pencilblue (CMS) [73]	articles, comments	3	–	46	27	70	73
Leanote (notes) [54]	notes, books, tags	5	5	64	28	69	92
Budget manager [15]	expenditure, ledgers	–	–	5	0	5	5
Redux (chat) [79]	messages, groups	–	–	3	0	3	3

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

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 6: Microbenchmarks of cryptographic schemes used by Arx in μs.

Height	Token	Cover	Expansion
8	3.7	9.5	131.9
10	4.6	14.5	542.3
12	5.5	20.5	2164.9

Figure 7: Microbenchmarks of ArxEq operations in μs .

if the unsupported data field is nonsensitive and mark it as such, but this should be done with care. The table also shows that though Arx's planner increases the number of indices by 20%, this number does not blow up. The main reason is that the number of fields with order queries that are not indexed by the application is small.

10.2 Microbenchmarks

Encryption building blocks. The cryptographic schemes used by Arx are efficient, as shown in Fig. 6. The reported results are the median of a million iterations.

ArxEq microbenchmarks. The ArxEq protocol encrypts a value v as $(\mathsf{BASE}(v), t)$, where t is a token for the value computed using the search token tree as described in §5. The time to compute t is directly proportional to the height of the tree, involving a hash computation at each level. We evaluate the time taken to compute t for different tree heights, and report the results as the median of a 100K iterations in Fig. 7. The results show that ArxEq encryption is efficient.

To search for v, the client proxy computes the covering set of all tokens and sends it to the server. The computation depends on the number of existing tokens for v, which ranges from 1 to 2^h where h is the height of the tree. We compute the cover for a randomly selected number of tokens, and report the median time over 100K iterations. The server proxy searches for v by expanding the covering set into all possible tokens. Fig. 7 shows that the operations are efficient and the client proxy does little work compared to the server.

ArxRange microbenchmarks. 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 19.8K cycles and the time to evaluate is 7.8K cycles. For a 128-bit value, the circuit is 12.3KB in size, the time to garble is 70.1K cycles (0.03ms) and the time to evaluate is 29.1K cycles.

10.3 Performance of ArxEq

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

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

Fig. 9 compares the write throughput of ArxEq with increasing number of duplicates. The write performance of a regular B+Tree index slowly improves with increased duplication, as a result of a corresponding decrease in the height of the tree. In contrast, writes to an ArxEq index are independent of the number of duplicates by virtue of security: each value looks different. Further, since each individual insert requires the computation of a single token (a constant-time operation), the write throughput of ArxEq remains stable in this experiment. As a result, the net overhead grows from 18% (when fields are unique) to 25% when there are 100 duplicates per value. Latency follows a similar trend (see Fig. 11) and remains stable for ArxEq at ∼3.3ms. For a regular MongoDB index, the latency slowly improves from 2.7ms to 2.5ms as the number of duplicates grows to 100.

YCSB Benchmark. Since Arx is a NoSQL database, we also evaluate its overhead using the YCSB benchmark [23]. YCSB conforms to ArxEq's optimized case when fields are unique. In this experiment, we loaded the database with 1M documents. Arx considers all fields to be sensitive by default, including the primary key. Hence, the primary key has an ArxEq index and the rest of the fields are encrypted with BASE. Fig. 10 shows the average performance of Arx versus vanilla MongoDB, across four different workloads with varying proportions of reads and writes, as specified.

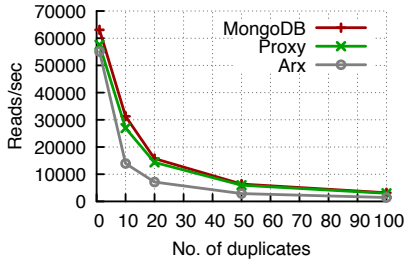

Figure 8: ArxEq read throughput with increasing no. of duplicates.

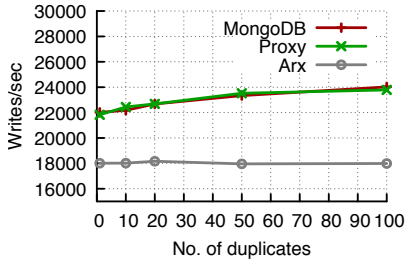

Figure 9: ArxEq write throughput with increasing no. of duplicates.

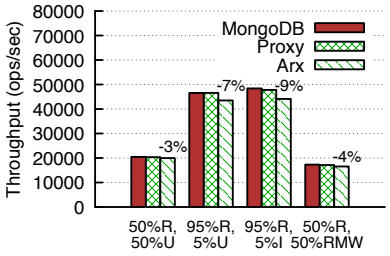

Figure 10: YCSB throughput for different workloads.

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 11: ArxEq latency of reads and writes with increasing no. of duplicates.

"R" refers to proportion of reads, "U" to updates, "I" to inserts, and "RMW" to read-modify-write. The reduction in throughput is higher for read-heavy workloads as a result of the added latency due to sequential decryption of the result sets. Overall, the overhead of Arx is 3%-9% across workloads, showing that indexing primary keys is fast with Arx. Increase in latency due to Arx is also unremarkable—the 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%R - 50%U workload.

10.4 Performance of ArxRange

We now evaluate the latency introduced by ArxRange. We pre-inserted 1M values into the index, and assumed a length of 128 bits for the index keys, which is sufficient for composite keys. We cached the top 1000 nodes of the index at the client proxy, which amounted to a mere 88KB of memory. We subsequently evaluated the performance of read and write operations on the index. Fig. 12 illustrates the latency of each operation, divided into three parts: (1) the time taken to traverse the index, (2) the time taken to decrypt the retrieved document IDs (for reads)—this incurs a network roundtrip as described in §4.3; (3) the time taken to retrieve the corresponding results (for reads) or insert the document (for writes), and (4) the time taken to repair the index. The generation of fresh garbled circuits in order to repair the index contributes the most towards latency.

Overall, range queries cost more than writes because the former require a network roundtrip in order to decrypt the retrieved IDs before fetching the corresponding documents. The cost of traversing a path in the index is ∼3ms. We note that the strawman in §4.1 incurs a roundtrip overhead for each node in the path, while our protocol incurs only a single roundtrip cost for decrypting the IDs in the leaves of the index. Fig. 12 also highlights the improvement when the index can be optimized for monotonic inserts, which was common in the applications we evaluated. We also note that though the overall latency of ArxRange is high, the results of a range query can be returned to the client *before* performing the repair operation (see §4.4). Thus, in low load scenarios, the effective latency of a range query drops to ∼15ms.

We next measure the throughput of ArxRange in Fig. 15. Without client-side caching of nodes, the throughput of the index is very limited, since each operation requires the circuit at the root of the index to be replenished, forcing the operations to be sequential. However, when the top few levels of the tree are cached at the client, multiple queries to different parts of the index can proceed in parallel, and the throughput increases by more than an order of magnitude (at which point the client proxy in our testbed gets saturated).

10.5 Performance of ArxAgg

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

10.6 End-to-end evaluation on ShareLaTeX

We now evaluate the end-to-end overhead of Arx using ShareLaTeX [81], a popular web application for real-time collaboration on LaTeX projects, that uses MongoDB for persistent storage. We chose ShareLaTeX because it uses both of Arx's indices, it has sensitive data (documents, chats) and is a popular application. ShareLaTeX maintains multiple collections in MongoDB corresponding to users, projects, documents, chat messages, etc. We considered all the fields in the application to be sensitive, which is the default in Arx. The application was run on four cores of the client server.

Before every experiment, we pre-loaded the database with 100K projects, 200K users, and other collections with 100K records each. Subsequently, using Selenium (a tool for automating browsers [80]), multiple users launch browsers in parallel and collaborate on projects in pairs—(i) editing documents, and (ii) exchanging messages via chat. We ran the user processes on a separate machine with 48 cores. Fig. 13 shows the throughput of ShareLaTeX in a vanilla deployment with regular MongoDB, compared to its performance with Arx in various configurations. The client proxy is either collocated with the ShareLaTeX application sharing the same four cores, or deployed on extra and separate cores. The application's throughput declines by 29% when the client proxy and ShareLaTeX are collocated; however, when two separate cores are allocated to Arx's client proxy, the reduction in throughput stabilizes at a reasonable 10%.

Fig. 14 compares the performance of Arx with increasing load at the application server, when four separate cores are allocated to Arx's client proxy. It also shows the performance of MongoDB with the proxy without the Arx hooks. Note that each client thread issues many requests as fast as it

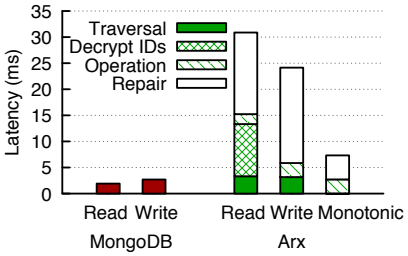

Figure 12: ArxRange latency of reads and writes.

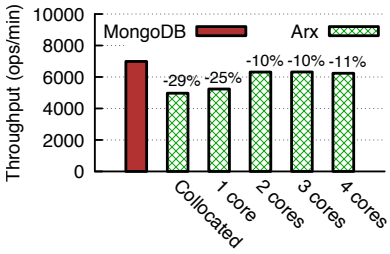

Figure 13: ShareLaTeX performance with Arx's client proxy on varying cores

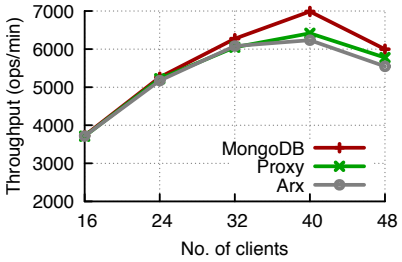

Figure 14: ShareLaTeX performance with increasing no. of client threads

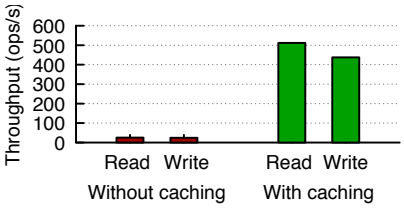

Figure 15: ArxRange throughput, with and without caching.

can, bringing a load equivalent to many real users. At peak throughput with 40 clients and 100% CPU load at the application, the reduction in performance due to Arx is 11%; 8% is due to Arx's proxy, and the remaining 3% due to its encryption and indexing schemes.

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

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

10.7 Storage

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

Finally, the application required two ArxEq indices for which counter maps were maintained at the client proxy, which in turn occupied 8.6MB of memory, illustrating that ArxEq 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.

10.8 Comparison with CryptDB

Arx supports fewer queries than CryptDB, but we find their functionalities are nevertheless comparable. For example, CryptDB supports all the queries in the TPC-C benchmark [86], while Arx supports 30 out of 31 queries. Arx also enables a rich class of applications as shown above, though it does not support group-by operations (for security issues), arbitrary conjunctions of range filters, and more generic joins, supported by CryptDB.

As regards performance, on one hand, CryptDB's order and equality queries via PPE schemes are faster than Arx's—with a reported overhead of ∼1ms [76], as opposed to a few milliseconds in Arx—but also significantly less secure. On the other hand, Arx's aggregate over a range is an order of magnitude faster for the same security, because CryptDB uses Paillier [70] to compute aggregates which requires a homomorphic multiplication per value in the range. For a range of 10,000 values, aggregates take 80ms in CryptDB compared to ∼3ms in Arx. Overall, Arx is a heavier solution due to the significant extra security, but remains at par with CryptDB in terms of overall impact on target applications: both systems report an overhead on the order of 10%.

11. LIMITATIONS AND FUTURE WORK

ArxRange extensions. Our current ArxRange index is a binary tree. An interesting extension is to implement the index for data structures with higher fanout such as B-trees, *e.g.*, by (i) storing at each node in the tree multiple garbled circuits; and (ii) using a history-independent B-treap data structure [34], instead of a binary treap.

History-independence. One needs to be careful that when logically implementing a history-independent data structure (as in ArxRange), the physical implementation of it is history-independent as well. For example, in our treap data structure, we ultimately require file system support for implementing *secure deletion* [6, 78]. This is because, when a node is logically deleted, the file system needs to ensure that instead of merely unlinking the data structure in memory, all copies of the data (caches, in-memory and disk) are in fact physically removed so as to become irrecoverable to an attacker. Implementing secure deletion is complementary to our work.

Transactions. Arx currently does not support transactional semantics. While our techniques can be extended to

transactional systems as well, it has significant practical challenges. For instance, our ArxRange index requires updates to multiple nodes in the tree per query along with interaction with the client, making support for transactions complicated. However, doing so is interesting future work.

12. RELATED WORK

We compare Arx with state-of-the-art EDBs, and discuss protocols related to its building blocks, ArxEq and ArxRange. We do not discuss PPE-based EDBs [5,71,76,87] further as we have already compared Arx against them extensively in §1 and §2.3. Seabed [71] hides frequency in some cases, but still uses PPE.

EDBs using semantically-secure encryption. This category is the most relevant to Arx, but unfortunately, there is little work done in this space. First, the line of work in [17,29] is based on searchable encryption, but is too restricted in functionality. It does not support joins, order-by-limit queries (commonly used for pagination, more common than range queries in TPC-C [86]), or aggregates over a range (because the range identifies a superset of the relevant documents for security, yielding an incorrect aggregate). Regarding security, while being significantly more secure than PPE-based EDBs for offline attackers, for online attackers they could leak more than PPE-based EDBs because their range queries leak the number of values matching sub-ranges as well as some prefix matching information—leakage that is not implied by order. Arx addresses all these aspects. Other recent works [43,48] also support equality-based queries but do not support range, order-by-limit, or aggregates over range queries; the former doesn't support inserts or updates either.

Second, BlindSeer [72] is another EDB providing semantic security. BlindSeer provides stronger security than Arx and even hides the client query from the server through two-party computation. Its primary drawbacks with respect to Arx are performance and functionality. BlindSeer requires a large number of interactions between the client and server. For example, for a range query, the client and server need to interact for every data item in the range (and a few times more) because tree traversal is interactive. If the range contains many values, this query is slow. In Arx, there is no interaction in this case. BlindSeer also does not handle inserts easily, nor does it support deletes, updates, aggregates over ranges or order-by-limit queries.

Finally, Obladi [24] targets much stronger guarantees than Arx by combining ACID semantics with ORAM, but consequently, is also orders of magnitude slower.

Work related to ArxEq. ArxEq falls in the general category of searchable-encryption schemes and builds on insights from this literature. While there are many schemes proposed in this space [12,13,17,25,29,40,44,50,53,65,68,83,84], none of them meets the following desired security and performance from a database index. Besides semantic security, when inserting a value, the access pattern should not leak what other values it equals, and an old search token should not allow searching on newly inserted data (forward privacy), both crucial in reducing leakage [13]. Second, inserts, updates and deletes should be efficient and should not cause reads to become slow. ArxEq meets all these goals. Perhaps the closest prior work to ArxEq is [17]. This scheme uses revocation lists for delete operations, which adds significant complexity and overhead, as well as leaks more than our goal in Arx: it lacks forward privacy and the revocation lists leak various metadata. Sophos [13] also provides forward privacy, but uses expensive public key cryptography instead of symmetric key. Diana [14] is similar to ArxEq.

Work related to ArxRange. There has been a significant amount of work on OPE schemes in both industry and research communities [1–3,9,10,22,42,55,58,59,69,75,77,89,91]. OPE schemes are efficient but have significant leakage [64]. Order-revealing encryption (ORE) provides semantic security [11,19,56]. The most relevant of these is the construction by Lewi and Wu [56] which is more efficient than ArxRange because it does not need replenishment, but also less secure because it leaks the position where two plaintexts differ. Thus, it is not strictly more secure than OPE.

Acknowledgments

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

13. REFERENCES

[1] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database Management as a Service: Challenges and Opportunities. In *Proc. ICDE*, 2009.

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

[3] G. W. Ang, J. H. Woelfel, and T. P. Woloszyn. System and Method of Sort-Order Preserving Tokenization. US Patent Application 13/450,809, 2012.

[4] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. FOCS*, 1989.

[5] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A secure coprocessor for database applications. In *Proc. FPL*, 2013.

[6] S. Bajaj and R. Sion. HIFS: History Independence for File Systems. In *Proc. CCS*, 2013.

[7] S. Bearak. 2018 Data Breaches – The Worst So Far, 2018. https://www.identityforce.com/blog/2018-data-breaches.

[8] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of Garbled Circuits. In *Proc. CCS*, 2012.

[9] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-Preserving Symmetric Encryption. In *Proc. EUROCRYPT*, 2009.

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

[11] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *Proc. EUROCRYPT*, 2014.

[12] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A Survey of Provably Secure Searchable Encryption. *ACM Computing Surveys (CSUR)*, 2014.

[13] R. Bost. Sophos - Forward Secure Searchable Encryption. In *Proc. CCS*, 2016.

[14] R. Bost, B. Minaud, and O. Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *Proc. CCS*, 2017.

[15] Budget Manager. https://goo.gl/chFmct.

[16] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks against Searchable Encryption. In *Proc. CCS*, 2015.

[17] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Proc. NDSS*, 2014.

[18] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Proc. CRYPTO*, 2013.

[19] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. Practical Order-Revealing Encryption with Limited Leakage. In *Proc. IACR-FSE*, 2016.

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

[21] CipherCloud. CASB+ Platform. http://www.ciphercloud.com.

[22] CipherCloud. Tokenization. http://www.ciphercloud.com/tokenization.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SOCC*, 2011.

[24] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proc. OSDI*, 2018.

[25] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. CCS*, 2006.

[26] H. Daitch. 2017 Data Breaches, 2017. https://www.identityforce.com/blog/2017-data-breaches.

[27] F. B. Durak, T. M. DuBuisson, and D. Cash. What Else is Revealed by Order-Revealing Encryption? In *Proc. CCS*, 2016.

[28] M. Egorov and M. Wilkison. ZeroDB white paper. *arXiv:1602.07168*, 2016.

[29] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Proc. ESORICS*, 2015.

[30] S. Garg, S. Lu, and R. Ostrovsky. Black-Box Garbled RAM. In *Proc. FOCS*, 2015.

[31] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption. Cryptology ePrint Archive, Report 2015/1010, 2015. http://eprint.iacr.org/2015/1010.

[32] O. Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[33] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proc. STOC*, 1987.

[34] D. Golovin. B-Treaps: A Uniquely Represented Alternative to B-Trees. In *Proc. ICALP*, 2009.

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

[36] P. Grofig, M. Haerterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schroepfer, and W. Tighzert. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Sicherheit*, 2014.

[37] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *Proc. IEEE S&P*, 2019.

[38] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why Your Encrypted Database Is Not Secure. In *Proc. HotOS*, 2017.

[39] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. Cryptology ePrint Archive, Report 2016/895, 2016. http://eprint.iacr.org/2016/895.

[40] W. He, D. Akhawe, S. Jain, E. Shi, and D. X. Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proc. CCS*, 2014.

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

[42] H. Kadhem, T. Amagasa, and H. Kitagawa. MV-OPES: Multivalued-Order Preserving Encryption Scheme: A Novel Scheme for Encrypting Integer Value to Many Different Values. *IEICE Trans. Info. & Sys.*, 2010.

[43] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Databases. Cryptology ePrint Archive, Report 2016/453, 2016. http://eprint.iacr.org/2016/453.

[44] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proc. CCS*, 2012.

[45] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *Proc. IEEE S&P*, 2016.

[46] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic Attacks on Secure Outsourced Databases. In *Proc. CCS*, 2016.

[47] J. Kepner, V. Gadepally, P. Michaleas, N. Schear, M. Varia, A. Yerukhimovich, and R. K. Cunningham. Computing on Masked Data: A High Performance Method for Improving Big Data Veracity. *arXiv:1406.5751*, 2014.

[48] M. Kim, H. T. Lee, S. Ling, S. Q. Ren, B. H. M. Tan, and H. Wang. Better Security for Queries on Encrypted Databases. Cryptology ePrint Archive, Report 2016/470, 2016. http://eprint.iacr.org/2016/470.

[49] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *Proc. CANS*, 2009.

[50] K. Kurosawa. Garbled Searchable Symmetric Encryption. In *Proc. FC*, 2014.

[51] M.-S. Lacharité, B. Minaud, and K. G. Paterson. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage, 2018.

[52] C. Lan, J. Sherry, , R. A. Popa, S. Ratnasamy, and

Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proc. SIGCOMM*, 2015.
[53] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield-A System's Approach to Data Privacy on Public Cloud. In *Proc. USENIX Security*, 2014.
[54] Leanote. https://leanote.com/.
[55] S. Lee, T.-J. Park, D. Lee, T. Nam, and S. Kim. Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases. *IEICE Trans. Info. & Sys.*, 2009.
[56] K. Lewi and D. J. Wu. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *Proc. CCS*, 2016.
[57] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated Index Structures for Aggregation Queries. *ACM Trans. Info. & Sys. Sec.*, 2010.
[58] D. Liu and S. Wang. Programmable Order-Preserving Secure Index for Encrypted Database Query. In *Proc. CLOUD*, 2012.
[59] D. Liu and S. Wang. Nonlinear order preserving index for encrypted database query in service cloud environments. *Concurrency and Computation: Practice and Experience*, 2013.
[60] R. Merkle. *Secrecy, authentication and public key systems / A certified digital signature*. PhD thesis, Stanford University, 1979.
[61] Microsoft SQL Server. Always Encrypted Database Engine. https://goo.gl/51LwQ9.
[62] M. Naor and V. Teague. Anti-persistence: History Independent Data Structures. In *Proc. STOC*, 2001.
[63] M. Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Report 2015/668, 2015. http://eprint.iacr.org/2015/668.
[64] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proc. CCS*, 2015.
[65] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic Searchable Encryption via Blind Storage. In *Proc. IEEE S&P*, 2014.
[66] Netty Project, 2017. http://netty.io/.
[67] NodeBB. https://nodebb.org/.
[68] W. Ogata, K. Koiwa, A. Kanaoka, and S. Matsuo. Toward Practical Searchable Symmetric Encryption. In *Proc. IWSec*, 2013.
[69] G. Özsoyoglu, D. A. Singer, and S. S. Chung. Anti-Tamper Databases: Querying Encrypted Databases. In *Proc. DBSec*, 2003.
[70] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT*, 1999.
[71] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *Proc. OSDI*, 2016.
[72] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind Seer: A scalable private DBMS. In *Proc. IEEE S&P*, 2014.
[73] PencilBlue. https://goo.gl/SS4biS.
[74] R. A. Popa. *Building Practical Systems that Compute on Encrypted Data*. PhD thesis, MIT, 2014.
[75] R. A. Popa, F. H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proc. IEEE S&P*, 2013.
[76] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. SOSP*, 2011.
[77] F. Y. Rashid. Salesforce.com Acquires SaaS Encryption Provider Navajo Systems, 2011. https://goo.gl/MKiF2b.
[78] J. Reardon, D. Basin, and S. Capkun. SoK: Secure Data Deletion. In *Proc. IEEE S&P*, 2013.
[79] Redux. https://goo.gl/AWZy6z.
[80] Selenium. http://www.seleniumhq.org/.
[81] ShareLaTeX. https://www.sharelatex.com/.
[82] Skyhigh. Skyhigh Security Cloud. https://www.skyhighnetworks.com/.
[83] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *Proc. IEEE S&P*, 2000.
[84] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *Proc. NDSS*, 2014.
[85] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proc. CCS*, 2013.
[86] TPC-C Transation Processing Benchmark. http://www.tpc.org/tpcc/.
[87] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical Queries over Encrypted Data. *PVLDB*, 6(5):289–300, 2013.
[88] UNCAP: Ubiquitous iNteropable Care for Ageing People. http://www.uncap.eu/.
[89] L. Xiao, I.-L. Yen, and D. T. Huynh. Extending Order Preserving Encryption for Multi-User Systems. Cryptology ePrint Archive, Report 2012/192, 2012. http://eprint.iacr.org/2012/192.
[90] A. C. Yao. How to Generate and Exchange Secrets (Extended Abstract). In *Proc. FOCS*, 1986.
[91] D. Yum, D. Kim, J. Kim, P. Lee, and S. Hong. Order-Preserving Encryption for Non-uniformly Distributed Plaintexts. In *Proc. WISA*, 2011.
[92] S. Zahur, M. Rosulek, and D. Evans. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates. In *Proc. EUROCRYPT*, 2015.
[93] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proc. CCS*, 2015.

APPENDIX

Appendix A discusses some optimizations incorporated by the ArxRange index. In Appendix B, we provide formal proofs of security for Arx.

A. ArxRange OPTIMIZATIONS

In this section, we discuss some optimizations incorporated by the ArxRange index.

Optimizing garbled circuit chaining. For performance we do not compute the encoding function inside the garbled circuit. Instead, we chain the garbled circuits together by augmenting each garbled circuit with a **transition table**. The transition table aids in translating an input label I_i for the current circuit to an input label for the correct child circuit corresponding to the same bit value. Note that the server should not be able to infer the underlying bit value that the label corresponds to but nevertheless should be able to translate it to the correct label for the next circuit.

The garbled circuit at each node first performs the comparison $a < v$, and outputs a key K_0 or K_1 based on the result of the comparison. This key is the label of the output wire in the instantiation of the scheme.

For each bit i of the input, the transition table stores four ciphertexts. Let $I^0[i], I^1[i]$ denote the i-th input labels in the encoding e_a for the current circuit; let $O_0^0[i], O_0^1[i]$ be the corresponding labels for the left child, and $O_1^0[i], O_1^1[i]$ for the right child. The table stores the following four ciphertexts:

$$E(K_0, I^0[i], O_0^0[i]),$$
$$E(K_0, I^1[i], O_0^1[i]),$$
$$E(K_1, I^0[i], O_1^0[i]),$$
$$E(K_1, I^1[i], O_1^1[i]).$$

Here E denotes a double-key-cipher implemented as $E(A, B, X) = H(A||B) \oplus X$, where H is a random oracle. Hence, without having both A and B, it is impossible to learn any information about X. We note that there are many other instantiations of such double-key-ciphers in the literature, with different security guarantees under different assumptions but for simplicity we just resort to a random oracle in this construction.

The values in the transition table are not stored in a fixed order. Instead, the point-and-permute technique [8] is employed, which means that if the least significant bit (LSB) of $I^0[i]$ is 0, the table entries are stored in the order as written and otherwise switched. This way the evaluator knows which ciphertext is the correct one without learning what bit value corresponds to the label. A formal discussion follows in Appendix B.1.1.

Concurrency. ArxRange 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 up to 1024 queries can proceed in parallel. Queries to the same subtree still need to be sequential. A common case for such queries are monotonic inserts. Fortunately, for these, the optimization described in §4.3 avoids the tree traversal and the repair. Of course, queries to another ArxRange index or to other parts of the database can proceed in parallel.

Garbled circuit design. One of the main drawbacks of garbled circuits is that converting even a simple program to a circuit often results in large circuits, and hence bad performance. We put considerable effort into making our garbled circuits short and fast. First, we used the short circuit for comparison from [49], which represents comparison of n-bit numbers in n gates. Second, we employ transition tables between two garbled circuits, to avoid embedding the encoding information for a child circuit inside the garbled circuit. Since the encoding information is large, this optimization reduces the size of the garbled circuit by a factor of 128. Third, we use the half-gates technique [92] to further halve the size of the garbled circuit. Fourth, 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.

As a result of our optimizations, a garbled 32-bit comparison circuit is only 1040 bytes in our implementation, which is as small as two Paillier ciphertexts. Evaluating it takes only 64 fixed-key AES invocations of which 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.

B. FORMAL SECURITY ANALYSIS

In this section, we first formalize the construction of relevant data structures in Arx. We then prove the security of the individual encryption schemes in our system, followed by proofs of security for the overall system and its protocols.

B.1 Preliminaries

We start with formally defining the primitives used by the ArxRange index.

Notation. We denote the set of all binary strings of length n as $\{0,1\}^n$. A boolean circuit $f : \{0,1\}^n \to \{0,1\}^m$ computes a function that takes a string of length n as input, and outputs a string of length m. Given a list X, we write $X[i]$ to refer to the i-th element in X.

B.1.1 Branching Garbled Circuit Chains

ArxRange makes use of a branching garbled circuit chain, which we formally define in this section. Abstractly, this new cryptographic primitive allows one to build a network of garbled circuits where each node branches into other nodes and the output of the garbled circuit inside every node determines which path to take.

We first recall the definition of a garbling scheme [8].

Definition 4 (Garbling Scheme). *A* garbling scheme *is a tuple of algorithms* $\mathcal{G} =$ (Garble, Encode, Eval)

- $(F, e, d) \leftarrow \mathsf{Garble}(1^\lambda, f)$
  On input the security parameter λ *and a boolean circuit* $f : \{0,1\}^n \to \{0,1\}^m$, Garble *outputs* (F, e) *where* F *is a garbled circuit,* e *is encoding information, and* d *is decoding information.*
- $X \leftarrow \mathsf{Encode}(e, x)$
  On input encoding information e *and an input* $x \in \{0,1\}^n$, Encode *outputs a garbled input* X.
- $Y \leftarrow \mathsf{Eval}(F, X)$
  On input (F, X) *as above,* Eval *outputs a garbled output* Y.
- $y \leftarrow \mathsf{Decode}(d, Y)$
  On input decoding information d *and a garbled output* Y, Decode *outputs a plain output* y.

Our garbling scheme is *projective* [8]; in other words, $e = [X_1^0, X_1^1, \ldots, X_n^0, X_n^1]$ encodes a list of *tokens*, one pair for

each bit in the input $x \in \{0,1\}^n$. $\mathsf{Encode}(e,\cdot)$ uses the bits of $x = x_1 \cdots x_n$ to output a list $X = [X_1^{x_1}, \ldots, X_n^{x_n}]$.

In most settings, the circuit f is public. Circuit privacy can always be achieved through a universal circuit but this incurs a significant performance penalty. In ArxRange, it is publicly known that the circuits are comparison circuits, but the constants those circuits compare against should remain secret. To quantify the amount of information that can be leaked without compromising security, we define the XOR-topology of a circuit.

Definition 5 (XOR-Topology). *The XOR-topology $\Phi_{\text{xor}}(f)$ of a circuit f is a function that maps the circuit f to a circuit where every non-XOR gate is replaced with an AND gate.*

Following Bellare *et al.* [8], we define the security of a garbling scheme in terms of a leakage function Φ.

Definition 6 (Garbled Circuit Security).

- ***GC Circuit Privacy*** *(*$\mathsf{gc\text{-}circuit\text{-}prv.sim}_\Phi$*):* *Intuitively, the garbled circuit F should not reveal any more information than $\Phi(f)$. Concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f))$ and whose output is indistinguishable from F generated the usual way.*
- ***GC Evaluation Privacy*** *(*$\mathsf{gc\text{-}eval\text{-}prv.sim}_\Phi$*):* *Intuitively, the collection (F, X) should not reveal any more information about x than $f(x)$. Concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f), f(x))$ and whose output is indistinguishable from (F, X) generated the usual way.*

We continue with the definition of a branching garbled circuit chain:

Definition 7 (BGCC). *A* branching garbled circuit chain *is a tuple of algorithms* $\mathcal{G} = (\mathsf{Generate}, \mathsf{Encode}, \mathsf{Eval})$

- $(F, e) \leftarrow \mathsf{Generate}(1^\lambda, f, e_0, e_1)$
  On input the security parameter λ in unary, a boolean circuit $f : \{0,1\}^n \to \{0,1\}$, encoding information e_0, e_1, Generate *outputs (F, e) where F is a branch-chained garbled circuit, and e is encoding information.*
- $X \leftarrow \mathsf{Encode}(e, x)$
  On input encoding information e and an input x suitable for f, Encode *outputs a garbled input X.*
- $(b, X_b) \leftarrow \mathsf{Eval}(F, X)$
  On input (F, X) as above, Eval *outputs a bit $b = f(x)$ and garbled inputs $X_b = \mathsf{Encode}(e_{f(x)}, x)$.*

Similar to Definition 6, we define the security of a BGCC scheme in terms of a leakage function Φ.

Definition 8 (BGCC Security).

- ***BGCC Circuit Privacy*** *(*$\mathsf{bgcc\text{-}circuit\text{-}prv.sim}_\Phi$*):* *Intuitively F should not reveal any more information than $\Phi(f)$. Concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f))$ and whose output is indistinguishable from F generated the usual way.*
- ***BGCC Evaluation Privacy*** *(*$\mathsf{bgcc\text{-}eval\text{-}prv.sim}_\Phi$*):* *Intuitively, the collection (F, X) should not reveal any more information about x than $(f(x), X_{f(x)})$. Concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f), f(x), X_{f(x)})$ and whose output is indistinguishable from (F, X) generated the usual way.*

BGCC Construction. Following the high-level description in Appendix A we provide a formal description of our BGCC construction. Our branching garbled circuit chain construction is based on a linear garbling scheme. Concretely we use the half-gate garbled circuit scheme [92] for efficiency, which satisfies Definition 6.

Construction 1 (BGCC). *Let* $\mathcal{G}^* = (\mathsf{Garble}^*, \mathsf{Encode}^*, \mathsf{Eval}^*, \mathsf{Decode}^*)$ *be the half-gate garbling scheme. Let $f : \{0,1\}^n \to \{0,1\}$ be a boolean circuit with n inputs and 1 output, and H a random oracle.*

$\mathsf{Generate}(1^\lambda, f, e_0, e_1)$:
1. $(F^*, e^*, d^*) \leftarrow \mathsf{Garble}^*(1^\lambda, f)$
2. Let K_0 (respectively K_1) be the 0-label (respectively, 1-label) for the output wire in F^*.
3. Compute list of input labels for the current circuit
   $I^0 \leftarrow \mathsf{Encode}^*(e^*, 0^n)$
   $I^1 \leftarrow \mathsf{Encode}^*(e^*, 1^n)$
4. Compute list of input labels for the left child
   $O_0^0 \leftarrow \mathsf{Encode}^*(e_0, 0^n)$
   $O_0^1 \leftarrow \mathsf{Encode}^*(e_0, 1^n)$
5. Compute list of input labels for the right child
   $O_1^0 \leftarrow \mathsf{Encode}^*(e_1, 0^n)$
   $O_1^1 \leftarrow \mathsf{Encode}^*(e_1, 1^n)$
6. $T_0^0 \leftarrow [\,]$, $T_1^0 \leftarrow [\,]$, $T_0^1 \leftarrow [\,]$, $T_1^1 \leftarrow [\,]$ (empty lists)
7. for $i = 1, \ldots, n$:
   (a) $b = \mathrm{LSB}(I^0[i])$, $\quad \bar{b} = 1 - b$
   (b) $T_0^0[i] \leftarrow H(K_0 \,\|\, I^b[i]) \oplus O_0^b[i]$
   $T_1^0[i] \leftarrow H(K_1 \,\|\, I^b[i]) \oplus O_1^b[i]$
   (c) $T_0^1[i] \leftarrow H(K_0 \,\|\, I^{\bar{b}}[i]) \oplus O_0^{\bar{b}}[i]$
   $T_1^1[i] \leftarrow H(K_1 \,\|\, I^{\bar{b}}[i]) \oplus O_1^{\bar{b}}[i]$
8. $F \leftarrow (F^*, d^*, \{T_0^0, T_1^0, T_0^1, T_1^1\})$
9. Output (F, e^*)

$\mathsf{Encode}(e, x)$:
1. Output $\mathsf{Encode}^*(e, x)$

$\mathsf{Eval}(F, X)$:
1. Parse the input as $F = (F^*, d^*, \{T_0^0, T_1^0, T_0^1, T_1^1\})$.
2. $b \leftarrow \mathsf{Decode}^*(d^*, \mathsf{Eval}^*(F^*, X))$
3. Let K_b be the label on the output wire of F^*
4. $X_b \leftarrow [\,]$ (an empty list)
5. for $i = 1, \ldots, n$:
   (a) $d = \mathrm{LSB}(X[i])$
   (b) $X_b[i] \leftarrow T_b^d[i] \oplus H(K_b \,\|\, X[i])$
6. Output (b, X_b)

Theorem 1. *The BGCC construction 1 satisfies the syntax and correctness guarantees as stated in definition 7.*

Proof. The correctness instantly follows from the correctness of the garbling scheme. □

Theorem 2. *The BGCC construction 1 satisfies the BGCC security definitions* $\mathsf{bgcc\text{-}circuit\text{-}prv.sim}_{\Phi_{\text{xor}}}$ *and* $\mathsf{bgcc\text{-}eval\text{-}prv.sim}_{\Phi_{\text{xor}}}$.

Proof. We first prove that the underlying half-gate garbling scheme satisfies $\mathsf{gc\text{-}circuit\text{-}prv.sim}_\Phi$ and $\mathsf{gc\text{-}eval\text{-}prv.sim}_\Phi$ for $\Phi = \Phi_{\text{xor}}$.

Evaluation privacy follows from theorem 1 in [92]. Though the theorem includes the whole circuit topology in the leakage, an examination the proof reveals that in fact only the XOR-Topology is leaked. Specifically, in the proof the simulator only makes its decisions based on whether a particular gate

is an XOR gate or not and does not depend on the concrete type of the gate if it is not an XOR gate.

Circuit-privacy of the half-gate scheme can be easily seen by having a closer look at the garbling algorithm of the half-gate scheme.

The random oracle H is with probability $1 - \mathsf{negl}(\lambda)$ never evaluated more than once on the same input, hence the transition table is indistinguishable from random.

Given this, the construction of our simulators is straightforward. Our simulator for $\mathsf{bgcc\text{-}circuit\text{-}prv.sim}_{\Phi_{\text{xor}}}$ invokes the simulator of the underlying garbling scheme to obtain a simulation of a garbled circuit F. The simulator adds transition tables consisting of random bits and the output is indistinguishable from a branching garbled circuit generated the usual way.

Similarly, our simulator for $\mathsf{bgcc\text{-}eval\text{-}prv.sim}_{\Phi_{\text{xor}}}$ on input $(1^\lambda, \Phi_{\text{xor}}(f), f(x), X_{f(x)})$ invokes the simulator of the underlying garbling scheme to get a simulation (F, X). The simulator adds a transition table to F that would lead to the output of $X_{f(x)}$. □

B.1.2 Treap data structure

In ArxRange we want to ensure that an attacker upon intrusion does not learn anything about the execution of past queries. A standard balanced binary search tree, *e.g.*, a red-black tree or an AVL tree, may assume different *shapes* depending on the order of insertion. Define a binary search tree as a set of nodes N. Each node $\mathsf{node} \in \mathsf{N}$ contains an identifier nid, some information x, and pointers to two children nodes $\mathsf{child} \leftarrow [\mathsf{nid}_0, \mathsf{nid}_1]$, where nid_0 is the left child and nid_1 is the right child.

Definition 9 (Shape of a Binary Search Tree). *We define the* shape *of a binary search tree* N *to be the set of 3-tuples* $\{(\mathsf{nid}^i, \mathsf{nid}^i_0, \mathsf{nid}^i_1)\}_i$ *for all* $\mathsf{nid}^i \in \mathsf{N}$.

To prevent the shape from leaking information about the order of insertion of nodes, we use a *history-independent* treap [4] data structure to implement our index. A treap is a probabilistic tree data structure in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the inorder traversal order of the nodes is the same as the sorted order of the keys. The structure of the tree is determined by the requirement that it be heap-ordered: that is, the priority number for any non-leaf node must be greater than or equal to the priority of its children. Thus, the root node is the maximum-priority node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

To insert a value into the treap, the value is first simply added to the treap at the corresponding leaf position. Next, based on the piority value of the inserted node, the treap is rebalanced via rotations, starting from the inserted leaf, all the way up to the root. The rotations are necessary to preserve the property that each node has a priority higher than its children.

To delete a value from the treap, the corresponding node is first rotated down the treap either into a leaf position, or into a position where it only has a single child node. The node is then discarded. The rotations are performed in a manner that preserve the treap's properties.

We now outline properties of this data structure relevant to our construction.

Definition 10 (Rank of an Insert). *For a sequence of insert queries* $Q = \{q_1, \ldots, q_n\}$ *where* $q_i = (v_i, i)$ *is a tuple comprising the inserted value* v_i *and its position* i *in* Q*, define the rank of a query:* $rank(q_i) = |\{(v_j, j) \in Q | v_j < v_i\}|$*, i.e., the number of queries in the sequence that insert a smaller value.*

Fact 1. *If a sequence of insert queries* $Q = \{q_1, \ldots, q_n\}$ *generates a treap with certain shape, then the sequence of inserts* $Q' = \{(rank(q_1), 1), \ldots, (rank(q_n), n)\}$ *generates a treap with the same shape, assuming the same random treap priority values for nodes in both sequences.*

B.1.3 Building blocks

BASE is implemented as AES in counter mode with a random initialization vector, and is semantically secure (per [32], Definition 5.2.8) under standard cryptographic assumptions. AGG is the Paillier encryption scheme.

EQunique is a pseudorandom function.

B.2 Overall proofs of security

We provide proofs for the security of Arx and its protocols. We start by proving the security of the different protocols in Arx. We then provide a proof for the overall security guarantees of Arx per Definition 1.

Theorem 3. *The* EQ *and* ArxEq *schemes are* $\mathcal{L}$*-semantically secure per Definition 1 under standard cryptographic assumptions, for* $\mathcal{L}$ *defined in Definition 3 and Fig. 4.*

We omit the proof for Theorem 3 because it is very similar to that of Diana [14].

We now prove the security of the ArxRange scheme. Fig. 16 provides the real world experiment for ArxRange, and Fig. 17 provides the ideal world experiment. We show that these two worlds are indistinguishable. We first provide detailed pseudocode for ArxRange in Figs. 20 and 21. The protocol executes the submitted queries by invoking the interface exposed by Client (Fig. 20), which in turn communicates with Server (Fig. 21) to execute the queries. We do not model the optimizations for concurrent query execution and monotonic inserts, described in §4.4.

Theorem 4. *The* ArxRange *scheme is* $\mathcal{L}$*-semantically secure per Definition 1 assuming standard cryptographic assumptions and the random oracle model, for* $\mathcal{L}$ *defined in Definition 3 and Fig. 4.*

Proof. We construct a simulator Sim for the ideal world execution of ArxRange in Fig. 22. Sim internally uses $\mathsf{Sim}_{\mathsf{BGCC}}$, the simulator for our BGCC construction (Construction 1), to simulate the execution of the BGCC circuits in the index. We describe the key aspects of the Sim's behavior here; Fig. 22 provides a concrete construction.

Essentially, Sim replicates the functionality of Client and Server, with the difference being that Sim internally maintains a regular treap index directly over the values it receives as input (*i.e.*, the output of the leakage function, which is the ranks of the actual values). Sim's main task is to simulate the BGCC circuits in the index during query execution, which it does as follows. Whenever it needs to traverse the index given the rank of a value in the index, Sim identifies the node corresponding to that rank along with the unique path in the index from the root to the node. It then invokes $\mathsf{Sim}_{\mathsf{BGCC}}$

$\mathcal{L}(Q)$:

1. Ranks maps each query to a rank
   Ranks $\leftarrow$ { }
2. $S \leftarrow$ [ ]
3. Compute global rank for each query
   For all queries $\mathsf{q} \in Q$:
   (a) If q is an insert query:
      i. Let $v \leftarrow$ value to be inserted
      ii. $r \leftarrow \mathsf{rk}(S, v)$
      iii. For all ranks $r' \in$ Ranks.values:
         A. Increment r' if $r' \geq r$
      iv. Ranks[q] $\leftarrow r$
      v. Append v to S
   (b) Else if q deletes by id:
      i. Let $\mathsf{q}' \leftarrow$ query that inserted id
      ii. Ranks[q] $\leftarrow$ Ranks[q']
   (c) Else if q searches or deletes by range:
      i. Let $(a, b) \leftarrow$ query range
      ii. $r_a \leftarrow \mathsf{rk}(S, a-1) + 1$
      iii. $r_b \leftarrow \mathsf{rk}(S, b)$
      iv. Ranks[q] $\leftarrow (r_a, r_b)$
4. $L \leftarrow$ [ ]
5. For all queries $\mathsf{q} \in Q$:
   (a) If q is an insert query:
      i. Let v be the value to be inserted
      ii. Replace v with Ranks[q]
   (b) Else if q searches or deletes by range:
      i. Let $(a, b) \leftarrow$ query range
      ii. Replace (a, b) with Ranks[q]
   (c) Append q to L
6. Output L

Figure 18: Leakage function for ArxRange.

Data item	Description
nid	Node identifier
ckt_0, ckt_1	Two copies of a garbled circuit
BASE(v)	Ciphertext for the value v at the node
BASE(id)	Encrypted document identifier that the node points to
child $\leftarrow [\mathsf{nid}_0, \mathsf{nid}_1]$	Identifiers of children nodes

Figure 19: Contents of a node in an ArxRange index.

Client.Setup():

1. Initialize the root node:
   root $\leftarrow \perp$
2. Initialize encoding information for both copies of the circuit at the root (see §4.3):
   $e^0_{\mathsf{root}} \leftarrow \perp$
   $e^1_{\mathsf{root}} \leftarrow \perp$

Client.Search(a, b):

1. Generate input tokens for the index:
   $\mathsf{tok}_0 \leftarrow \mathsf{Encode}(e^0_{\mathsf{root}}, a)$
   $\mathsf{tok}_1 \leftarrow \mathsf{Encode}(e^1_{\mathsf{root}}, b)$
2. (IDs, N) $\leftarrow$ Server.Search(root.nid, root.nid, $\mathsf{tok}_0, \mathsf{tok}_1$)
3. Client.Repair(N)
4. Output IDs

Client.Insert(id, v):

1. Create a new garbled node node (see Fig. 19)
2. If root $\neq \perp$
   (a) tok $\leftarrow$ Encode(e^0_{root}, v)
   (b) N $\leftarrow$
   Server.Insert(id, BASE(id), BASE(node.nid), node, root.nid, tok)
   (c) Client.Repair(N)
   Else
   (a) Server.Insert(id, BASE(id), BASE(node.nid), node, $\perp$, $\perp$)

Client.Repair(N):

1. For node $\in$ N (traverse from leaves to root)
   (a) Generate two new BGCC circuits ckt_0 and ckt_1 for node
   (b) Server.Repair(node.nid, $\mathsf{ckt}_0, \mathsf{ckt}_1$)
2. Update root, e^0_{root}, and e^1_{root}

Client.Delete(a, b):

1. Generate input tokens for the index
   $\mathsf{tok}_0 \leftarrow \mathsf{Encode}(e^0_{\mathsf{root}}, a)$
   $\mathsf{tok}_1 \leftarrow \mathsf{Encode}(e^1_{\mathsf{root}}, b)$
2. N $\leftarrow$ Server.Delete(root.nid, root.nid, $\mathsf{tok}_0, \mathsf{tok}_1$)
3. Client.Repair(N)

Client.DeleteID(id):

1. N $\leftarrow$ Server.DeleteID(id)
2. Client.Repair(N)

Client.Decrypt(C):

1. $P \leftarrow$ [ ]
2. For $c \in C$:
   (a) Decrypt c and append to P
3. Shuffle and output P

Figure 20: Pseudocode for the Client-side execution of ArxRange. The node data structure is shown in Fig. 19, and Server is as defined in Fig. 21.

Server.Setup():
1. Index maps a node identifier nid to a node
   Index ← { }
2. $\mathsf{IDs_{fwd}}$ maps a node identifier nid to an encrypted document identifier BASE(id)
   $\mathsf{IDs_{fwd}}$ ← { }
3. $\mathsf{IDs_{rev}}$ maps a document identifier id to an encrypted node identifier BASE(nid)
   $\mathsf{IDs_{rev}}$ ← { }

Server.Traverse(node, tok, i):
1. Nodes ← { }
2. While node $\neq \perp$:
   (a) Evaluate the i-th BGCC circuit at node
   (dir, tok) ← Eval(node.ckt_i, tok)
   (b) Add the evaluated node to Nodes so that it can be repaired later
   Nodes ← Nodes ∪ {node}
   (c) Retrieve the next node in the path
   If node.child[dir] $\neq \perp$
      i. node ← Index[node.child[dir]]
   Else break
3. Output (dir, node, Nodes)

Server.Search($\mathsf{nid}_l, \mathsf{nid}_r, \mathsf{tok}_l, \mathsf{tok}_r$):
1. node_l ← Index[nid_l]
2. node_r ← Index[nid_r]
3. $(\cdot, \mathsf{leaf}_l, \mathsf{N}_l)$ ← Server.Traverse($\mathsf{node}_l, \mathsf{tok}_l, 0$)
4. $(\cdot, \mathsf{leaf}_r, \mathsf{N}_r)$ ← Server.Traverse($\mathsf{node}_r, \mathsf{tok}_r, 1$)
5. Do in-order traversal of Index from leaf_l to leaf_r, and collect the intermediate nodes in S
6. Initialize empty set O
7. For node $\in S$:
   (a) Append $\mathsf{IDs_{fwd}}$[node.nid] to O
8. L ← Client.Decrypt(O)
9. Output $(L, \mathsf{N}_l \cup \mathsf{N}_r)$

Server.Repair(nid, $\mathsf{ckt}_0, \mathsf{ckt}_1$):
1. node ← Index[nid]
2. node.ckt_0 = ckt_0
   node.ckt_1 = ckt_1

Server.Insert(id, BASE(id), BASE(nid), $\mathsf{node_{new}}, \mathsf{nid}_0$, tok):
1. nid ← $\mathsf{node_{new}}$.nid
2. $\mathsf{IDs_{fwd}}$[nid] ← (BASE(id))
3. $\mathsf{IDs_{rev}}$[id] ← (BASE(nid))
4. Index[nid] ← $\mathsf{node_{new}}$
5. node ← Index[nid_0]
6. (dir, parent, N) ← Server.Traverse(node, tok, 0)
7. parent.child[dir] = nid
8. Rebalance the index
9. Output N

Server.Delete($\mathsf{nid}_l, \mathsf{nid}_r, \mathsf{tok}_l, \mathsf{tok}_r$):
1. node_l ← Index[nid_l]
2. node_r ← Index[nid_r]
3. $(\cdot, \mathsf{leaf}_l, \mathsf{N}_l)$ ← Server.Traverse($\mathsf{node}_l, \mathsf{tok}_l, 0$)
4. $(\cdot, \mathsf{leaf}_r, \mathsf{N}_r)$ ← Server.Traverse($\mathsf{node}_r, \mathsf{tok}_r, 1$)
5. Do in-order traversal of Index from leaf_l to leaf_r, and collect the intermediate nodes in S
6. Initialize empty set O
7. For node $\in S$
   (a) Append $\mathsf{IDs_{fwd}}$[node.nid] to O
   (b) Delete $\mathsf{IDs_{fwd}}$[node.nid] from the $\mathsf{IDs_{fwd}}$ map
   (c) Delete Index[node.nid] from the Index map
8. L ← Client.Decrypt(O)
9. For id $\in L$
   (a) Delete $\mathsf{IDs_{rev}}$[id] from the $\mathsf{IDs_{rev}}$ map
10. N ← $\{\mathsf{N}_l \cup \mathsf{N}_l\} \setminus \{S\}$
11. Adjust node pointers, and rebalance the index
12. Add to N all nodes with at least one different child node after rebalancing
13. Output N

Server.DeleteID(id):
1. nid ← Client.Decrypt($\mathsf{IDs_{rev}}$[id])
2. Delete $\mathsf{IDs_{fwd}}$[nid] from the $\mathsf{IDs_{fwd}}$ map
3. Delete Index[nid] from the Index map
4. Delete $\mathsf{IDs_{rev}}$[id] from the $\mathsf{IDs_{rev}}$ map
5. Adjust node pointers, and rebalance the index
6. N ← all nodes with at least one different child node after rebalancing
7. Output N

Figure 21: Pseudocode for the Server-side execution of ArxRange. The node data structure is shown in Fig. 19.

$\mathbf{Real}_{\mathcal{A}}^{\mathsf{ArxRange}}(\lambda, Q)$:
1. Transcript ← { }
2. Execute queries and produce transcript
   For all queries q $\in Q$:
   (a) If q is an insert query:
      i. Let (id, v) ← identifier and value pair to be inserted
      ii. R ← Client.Insert(id, v)
   (b) Else if q deletes by id:
      i. R ← Client.Delete(id)
   (c) Else if q searches by range:
      i. Let (a, b) ← range bounds in the query
      ii. R ← Client.Search(a, b)
   (d) Else if q deletes by range:
      i. Let (a, b) ← range bounds in the query
      ii. R ← Client.Delete(a, b)
   (e) Append R to Transcript
3. Output $\mathcal{A}$(Transcript)

Figure 16: Real world experiment for ArxRange, where Client is as defined in Fig. 20.

$\mathbf{Ideal}_{\mathcal{A},\mathsf{Sim},\mathcal{L}}^{\mathsf{ArxRange}}(\lambda, Q)$:
1. Simulate query execution and produce transcript
   (a) Transcript ← Sim($\mathcal{L}(Q)$)
2. Output $\mathcal{A}$(Transcript)

Figure 17: Ideal world experiment for ArxRange. The leakage function $\mathcal{L}$ is given in Fig. 18.

to simulate the execution of the BGCC nodes in the path. Specifically, for each node in the path, Sim checks whether its child is the left node or the right, and accordingly invokes $\mathsf{Sim_{BGCC}}$ to simulate the BGCC circuit at the node. In doing so, it retroactively updates the function invocations in the transcript that previously created the node. Note that the root circuit is also part of the path during traversal; using the tokens outputted by $\mathsf{Sim_{BGCC}}$ for the root, Sim simulates the real-world client-server interaction for the search / insert / delete operation that triggered the traversal.

We argue the indistinguishability of the real and ideal

world views, as follows. The adversary's view contains all communication between Client and Server (*i.e.*, the inputs to the API exposed by Server and the outputs), along with all server state. Private network communication includes garbled nodes, tokens, and ciphertexts, the indistinguishability of which follows from Theorem 2 and the security of the BASE encryption scheme (Appendix B.1.3). The indistinguishability of the index at the server follows from the fact that each node in the index contains a BGCC, an encryption of the key and an encryption of the payload (the value). The indistinguishability of BGCCs follows from Theorem 2, and the indistinguishability of the latter two follows from the security of the encryption scheme. Finally, the indistinguishability of the server's execution (*i.e.*, traversal of the index) follows because given indices with the same shape and queries with the same rank, the outputs of the BGCCs in the execution are the same (since the path from the root to the leaf is the same). □

We now prove the overall security of Arx. For simplicity, we only consider queries on ArxEq and ArxRange indices; we do not model ArxAgg and ArxJoin since they are straightforward extensions of the underlying ArxEq or ArxRange index. For the online attacker, we assume that the initial database is empty, and all documents are inserted into the database using a sequence of Insert operations.

Additionally, as described in §9, we model queries that have multiple predicates as a sequence of queries, such that each query in the sequence contains at most one predicate and touches at most a single index. As a result, our leakage definitions are an upper bound on the actual query leakage.

THEOREM 5. *The Arx database system is $\mathcal{L}$-semantically secure per Definition 1 for $\mathcal{L}$ as defined in Definitions 2 and 3, assuming standard cryptographic assumptions and the random oracle model.*

PROOF. We first briefly recall the definition of a database DB from §9: $\mathsf{DB} = (\{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$ where each collection $\mathsf{T}_i = (\mathsf{F}_i, \mathsf{Ind}_i, [(\mathsf{id}_j, \mathsf{D}_j)]_j))$. In addition, before the actual execution of queries, the admin supplies a list of query predicates $\mathbb{P}(\mathsf{T}_i)$ for each collection T_i. Since every individual query pertains to a specific collection, without loss of generality, we henceforth assume that the database contains a single collection T. Further, since we model queries as operating over single fields, without loss of generality we further assume that the collection contains a single field f, such that every protocol $\mathsf{e} \in \mathbb{E}$ is maintained on it. With the exception of ArxRange, each protocol adds a corresponding ciphertext to the documents.

We proceed to prove the theorem using a series of hybrids.

Hybrid H_0. This corresponds to the real world view of $\mathcal{A}_{\mathsf{on}}$.

Hybrid H_1. Same as H_0, but with the database system modified as follows. Given a list of queries Q, the system first reorders the queries—it groups together all the queries that operate on ArxEq in order of their timestamps and pushes the group to the end of the query list, and similarly for all the EQ and ArxRange queries as well. It then executes the queries in order, capturing the view per query into the transcript. Finally, it reorders the per-query items in the transcript in accordance with their original order in Q. Since the per-index query execution is independent of other indices and the database, with independent key material, this hybrid is indistinguishable from H_0.

Hybrid H_2. Same as H_1, except that for the queries predicated on the field f_{EQ}, the server obtains a response from the simulator for the EQ protocol $\mathsf{Sim}_{\mathsf{EQ}}$.

Hybrid H_3. Same as H_2, except that for the queries predicated on the field $\mathsf{f}_{\mathsf{ArxEq}}$, the server obtains a response from the simulator for the ArxEq protocol $\mathsf{Sim}_{\mathsf{ArxEq}}$.

Hybrid H_4. Same as H_3, except that for the queries predicated on the field $\mathsf{f}_{\mathsf{ArxRange}}$, the server obtains a response from the simulator for the ArxRange protocol $\mathsf{Sim}_{\mathsf{ArxRange}}$.

Hybrid H_5. For every collection T_i, we replace the ciphertexts in all documents $\mathsf{D}_j \in \mathsf{T}_i$ with random strings.

The construction of a simulator for the ideal world follows directly from H_5. The indistinguishability of the hybrids follows from Theorems 3 and 4 and the security of Arx's building blocks (Appendix B.1.3). □

Sim($\mathcal{L}(Q)$):
1. Initialize the root node and data structures: root $\leftarrow \perp$, Index $\leftarrow \{\ \}$, $\mathsf{IDs_{fwd}} \leftarrow \{\ \}$, $\mathsf{IDs_{rev}} \leftarrow \{\ \}$
2. Transcript $\leftarrow \{\ \}$
3. Simulate query execution and produce transcript For all queries $\mathsf{q} \in \mathcal{L}(Q)$:
   (a) If q is an insert query:
      i. Let id $\leftarrow$ identifier for value to be inserted
      ii. $R \leftarrow$ Sim.Insert(id, Ranks[q])
   (b) Else if q deletes by id:
      i. $R \leftarrow$ Sim.Delete(id)
   (c) Else if q searches by range:
      i. $R \leftarrow$ Sim.Search(Ranks[q])
   (d) Else if q deletes by range:
      i. $R \leftarrow$ Sim.Delete(Ranks[q])
   (e) Append R to Transcript
4. For all nodes in Index from leaves to root without ckt_i:
   (a) Generate $\mathsf{node.ckt}_i$ using $\mathsf{Sim_{BGCC}}$
   (b) Update the Insert or Repair invocation in Transcript with the simulation of $\mathsf{node.ckt}_i$ for when node was last generated or repaired
5. Output Transcript

Sim.Search(a, b):
1. $(\cdot, \mathsf{leaf}_l, \mathsf{N}_l) \leftarrow$ Sim.Traverse($\mathsf{node}_l, a, 0$)
2. $(\cdot, \mathsf{leaf}_r, \mathsf{N}_r) \leftarrow$ Sim.Traverse($\mathsf{node}_r, b, 1$)
3. Do in-order traversal of Index from leaf_l to leaf_r, and collect the intermediate nodes in S
4. Initialize empty set O
5. For node $\in S$:
   (a) Append $\mathsf{IDs_{fwd}}$[node.nid] to O
6. IDs $\leftarrow$ Sim.Decrypt(O)
7. Sim.Repair($\mathsf{N}_l \cup \mathsf{N}_r$)
8. Output IDs

Sim.Traverse(node, v, i):
1. Nodes $\leftarrow \{\ \}$
2. While node $\neq \perp$:
   (a) Evaluate node by comparing v with the value at the node
   (b) Add the node to Nodes
   Nodes $\leftarrow$ Nodes $\cup$ {node}
   (c) Retrieve the next node in the path
   If $\mathsf{node.child[dir]} \neq \perp$
      i. node $\leftarrow$ Index[node.child[dir]]
   Else break
3. For each node in Nodes from leaf to root, simulate the circuits at each node using $\mathsf{Sim_{BGCC}}$ as follows:
   (a) If node is a leaf:
      i. Choose random b
      ii. Choose random labels tok
   Else
      i. Set $b = 0$ if the left child of node is in Nodes, else $b = 1$
      ii. Set tok based on the output of $\mathsf{Sim_{BGCC}}$ for the child node
   (b) Generate $\mathsf{node.ckt}_i$ and corresponding tokens using $\mathsf{Sim_{BGCC}}$ on input b and tok
   (c) Update the Insert or Repair invocation in Transcript with the simulation of $\mathsf{node.ckt}_i$ for when node was last generated or repaired
4. Simulate the client-server interaction for the calling function (*i.e.*, Search, Insert, or Delete): Update the invocation of the calling function in Transcript with the tokens output by $\mathsf{Sim_{BGCC}}$ for the root node in step 3b
5. Output (dir, node, Nodes)

Sim.Repair(N):
1. For n $\in$ N (traverse from leaves to root)
   (a) node $\leftarrow$ Index[n.nid]
   (b) Delete the circuits ckt_0 and ckt_1 at node (these will be generated using $\mathsf{Sim_{BGCC}}$ later)
2. Update root

Sim.Delete(a, b):
1. $(\cdot, \mathsf{leaf}_l, \mathsf{N}_l) \leftarrow$ Server.Traverse($\mathsf{node}_l, a, 0$)
2. $(\cdot, \mathsf{leaf}_r, \mathsf{N}_r) \leftarrow$ Server.Traverse($\mathsf{node}_r, b, 1$)
3. Do in-order traversal of Index from leaf_l to leaf_r, and collect the intermediate nodes in S
4. Initialize empty set O
5. For node $\in S$
   (a) Append $\mathsf{IDs_{fwd}}$[node.nid] to O
   (b) Delete $\mathsf{IDs_{fwd}}$[node.nid] from the $\mathsf{IDs_{fwd}}$ map
   (c) Delete Index[node.nid] from the Index map
6. IDs $\leftarrow$ Sim.Decrypt(O)
7. For id $\in$ IDs
   (a) Delete $\mathsf{IDs_{rev}}$[id] from the $\mathsf{IDs_{rev}}$ map
8. N $\leftarrow \{\mathsf{N}_l \cup \mathsf{N}_l\} \setminus \{S\}$
9. Adjust node pointers, and rebalance the index
10. Add to N all nodes with at least one different child node after rebalancing
11. Sim.Repair(N)

Sim.DeleteID(id):
1. nid $\leftarrow$ Sim.Decrypt($\mathsf{IDs_{rev}}$[id])
2. Delete $\mathsf{IDs_{fwd}}$[nid] from the $\mathsf{IDs_{fwd}}$ map
3. Delete Index[nid] from the Index map
4. Delete $\mathsf{IDs_{rev}}$[id] from the $\mathsf{IDs_{rev}}$ map
5. Adjust node pointers, and rebalance the index
6. N $\leftarrow$ all nodes with at least one different child node after rebalancing
7. Sim.Repair(N)

Sim.Insert(id, v):
1. Create a new node node (see Fig. 19) additionally containing v, but without the circuits ckt_0 and ckt_1 (these will be generated using $\mathsf{Sim_{BGCC}}$ later)
2. nid $\leftarrow$ node.nid
3. $\mathsf{IDs_{fwd}}$[nid] $\leftarrow$ (BASE(id))
4. $\mathsf{IDs_{rev}}$[id] $\leftarrow$ (BASE(nid))
5. Index[nid] $\leftarrow$ node
6. If root $\neq \perp$
   (a) (dir, parent, N) $\leftarrow$ Sim.Traverse(root, $v, 0$)
   (b) parent.child[dir] = nid
   (c) Rebalance the index
7. Sim.Repair(N)

Sim.Decrypt(C):
1. $P \leftarrow [\ ]$
2. For $c \in C$:
   (a) Decrypt c and append to P
3. Shuffle and output P

Figure 22: Simulator for ArxRange.