Arx: A DBMS with Semantically Secure Encryption

Rishabh Poddar
Tobias Boelter
Raluca Ada Popa

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-111
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-111.html

May 21, 2017

Arx: A DBMS with Semantically Secure Encryption

Rishabh Poddar Tobias Boelter Raluca Ada Popa
UC Berkeley

ABSTRACT

In recent years, encrypted databases have emerged as a promising direction that provides data confidentiality without sacrificing functionality: queries are executed on encrypted data. However, existing practical proposals rely on a set of weak encryption schemes that have been shown to leak sensitive data. In this paper, we propose Arx, 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 and a health data cloud provider with a modest performance overhead.

KEYWORDS

Security and privacy; encrypted databases; property preserving encryption; semantic security.

1 INTRODUCTION

Due to numerous data breaches [41, 73], the public concern over privacy and confidentiality is likely at one of its peaks today. Unfortunately, protecting the data is not as easy as encrypting it because encryption precludes useful computation on this data.

In recent years, encrypted databases [19, 71, 76, 83] (EDBs) have emerged as a promising direction towards achieving both confidentiality and functionality: queries run on encrypted data. CryptDB [76] showed that such an approach can be practical and can support a rich set of queries, and was followed by a rich line of work including Cipherbase [19] and Monomi [83]. The demand for such systems is demonstrated by the adoption in industry such as in Microsoft's SQL Server [63], Google's Encrypted Big Query [40], and SAP's SEEED [42] amongst others [5, 28, 30, 51]. Most of these services are *NoSQL databases* of various kinds showing that a certain class of encrypted computation suffices for many applications.

At their core, these EDBs use a set of encryption schemes, some of which are property-preserving (denoted PPE schemes) such as order-preserving encryption (OPE) [21, 22, 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, a series of recent attacks [33, 43, 66] have shown that an attacker can extract a significant amount of sensitive information from the leakage in these schemes when the attacker has certain auxiliary information. These works demonstrate *offline* attacks in which the attacker is able to steal a snapshot of the OPE-encrypted or DET-encrypted database. We call this attacker model a *snapshot* attacker. Because OPE and DET leak the order or frequency count of values within a column, when combined with certain auxiliary information, the attacker is able to decrypt a large fraction of the sensitive column.

Preventing *any leakage* at the server is known to be very difficult to achieve in a *practical* way. It would require (1) using only semantically secure encryption schemes, and (2) oblivious protocols (e.g., ORAM [82]) to hide query access patterns, along with significant padding [65], which are heavy protocols. For the latter, Naveed [65] shows that in some cases it is more efficient to stream the database to the client and answer queries locally than to use such a system on the server. In this work, we focus on the former.

We consider the following question: how can we keep the database encrypted only with *semantically secure* encryption schemes while *still providing rich functionality and good performance?* Semantic security implies that no partial information about the data is leaked (other than size information), preventing the aforementioned inference attacks on a stolen database.

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. [26] and Faber et al. [35], which builds on searchable encryption. As a result, these are significantly limited in functionality: not supporting common queries such as order-by-limit, aggregates on ranges, and joins, as well as being inefficient for write operations (e.g. updates, deletes), as we elaborate in §12. Also, while these systems are significantly more secure for an offline attacker, for certain *online* attackers (which we call *persistent* attackers) they have an extra leakage not present in PPEs, as we explain in §12. To replace PPE-based EDBs, we need a solution that is always at least as secure as PPE-based EDBs.

In this paper, we propose Arx, a practical and functionally-rich database system that encrypts the data only with *semantically secure* encryption schemes. Arx supports a rich set of queries: equality, ranges, aggregates over ranges, order-by-limit, and a common class of joins; importantly, Arx also integrates updates and deletes seamlessly. At the same time, Arx is significantly more secure than PPE-based EDBs. First, by using semantically-secure encryption schemes, Arx protects against the recent offline attacks [33, 43, 66] from which PPE-based EDBs suffer. Second, for online attackers, Arx is *always* either more secure or as secure as PPE-based EDBs. Thus, we propose Arx as an alternative to PPE-based EDBs.

1.1 Summary of techniques and contributions

To achieve rich computation, Arx introduces two new database indices: Arx-Range for range and order-by-limit queries, and Arx-Eq for equality queries. While Arx-Range can be used for equality queries as well, Arx-Eq is substantially faster.

To enable range queries, **Arx-Range** 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.* Our tree is history-independent [18] to avoid structural leakage. The main challenge with Arx-Range 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 [36] and chains garbled circuits. Arx chains 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 Arx-Range 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.

Arx-Eq 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. Arx-Eq provides forward privacy [25], preventing old search tokens from being used to search new data. When searching for a value v, the client can provide a small token to the server, which the server can expand into many search tokens for all the occurrences of v.

Arx speeds up aggregations with **Arx-Agg** by transforming an aggregate into a tree lookup, and provides foreign-key joins with **Arx-Join**, both indices being built on Arx-Eq and Arx-Range.

Because of the restrictions of encrypted indices, index and query planning becomes challenging in Arx. The application's developer specifies a set of regular indices and thus expects a certain asymptotic performance. However, there is no direct mapping between regular indices and Arx's indices because Arx's indices pose new constraints. The main constraints are: the same index is not used for both = and ≥ operations, an equality index on (a, b) cannot be used to compute equality on a alone, and Arx can compute range queries only via Arx-Range. With these in mind, we designed **an index planning algorithm** that guarantees the expected asymptotic performance while building few additional indices.

Finally, we designed **Arx's architecture** so it is amenable to adoption. Two lessons [74] greatly facilitated the adoption of the CryptDB system: do not change the database (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 provide **an implementation and evaluation of Arx** on top of MongoDB, a popular NoSQL database. We plan to open source our implementation. We show that Arx supports a wide range of real applications, such as ShareLaTeX [12], the Chino health data platform [4, 14], NodeBB forum [8], Leanote [6], and three others. In particular, Chino [4] is a health data cloud provider that serves the European medical project UNCAP [14]. Chino provides a MongoDB-like interface to medical web applications, which run on premises of hospitals. The leaders of UNCAP and Chino have expressed interest in using Arx for their system, which currently runs on *plaintext* data, and confirmed that Arx's model fits Chino perfectly. Finally, we show that Arx adds modest performance overheads. For example, Arx decreases throughput for ShareLaTeX by 11% and for the YCSB benchmark by 3–9%.

2 OVERVIEW

In 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 our implementation is on top of MongoDB, Arx's 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 calling 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 and run DB queries as part of one invocation.

The client proxy stores the master key. It rewrites queries, encrypts sensitive data, and forwards the encrypted queries to the server proxy for execution along with helper cryptographic tokens. It forwards any queries not containing sensitive fields directly to the DB server. The client proxy is *lightweight*: it does not store the database and does much less work than the server. The client proxy stores metadata (schema information), a small amount of state, and optionally a cache. The client proxy processes only the results of queries (e.g. to decrypt them) in most cases (except for a few corner cases we discuss). The server runs the expensive part of DB queries, filtering and aggregating many documents into a small result set.

2.2 Admin API

We now describe the API exposed by Arx to an application administrator. 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 Google's Encrypted BigQuery [40] and Microsoft's SQL Server [63], 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, an application admin specifies the following information to Arx during system setup:

(1) (optionally) field-specific information: which fields are unique, their maximum size, and which fields are *not* to be encrypted;
(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. fields such as SSN or driver's license number. Primary keys are automatically inferred by Arx to be unique. *info* may also specify the maximum length in bits for the field, which helps Arx choose a more effective encryption scheme.

Arx encrypts all the fields in the database by default. However, the admin may explicitly override this behavior by specifying *info* as "nonsensitive" for a particular field. This should be only 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

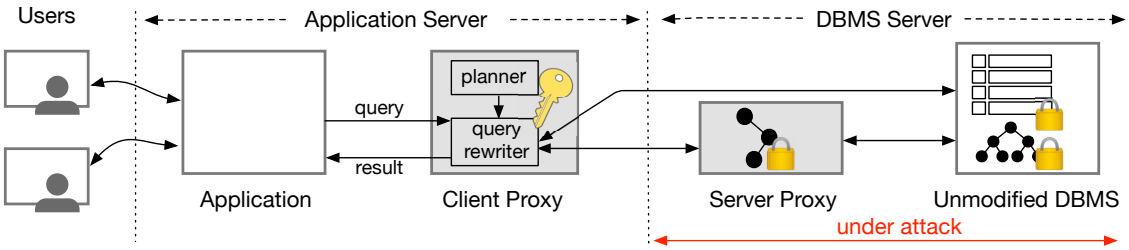

Figure 1: Arx's architecture: A trusted client proxy deployed at the application server, and an untrusted server proxy deployed at the DBMS server. The client proxy intercepts queries and encrypts sensitive information. The server proxy maintains indices over the encrypted data, and executes incoming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

field but supports everything else in the application, and 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 provide auxiliary information about other sensitive 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 what fields, but Arx does not need to know what constants will be queried for. For example, for the query `select * from T where age = 10`, Arx needs to know that there will be an equality check on age. The admin can either specify these operations to Arx or can provide a query trace from a run of this application and Arx will automatically identify them.

Third, Arx needs to know the list of regular indices built by the application. Arx requires 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.

2.3 Functionality

In this section, we describe the classes of read and write queries that can be supported by Arx over encrypted data. As we show in §11, this functionality supports a wide range of applications.

Read queries. Arx supports read 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 $=, \neq, \geq$ and $<$.

In addition to these queries, Arx supports a common form of joins—namely, foreign-key joins—which we describe in §8. It does not support other forms of join, which are part of our future work.

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

Constraints. Not all range/order queries are supported by Arx. First, the query may contain range operations over at most one encrypted field. While $f_1 \geq 3$ and $f_1 \leq 5$ is supported, $f_1 \geq 3$ and $f_1 \leq 5$ and $f_2 \leq 10$ is not supported; but additional range operations over nonsensitive and non-indexed fields are supported. Second, if the query contains a limit along with range operations over a sensitive field, then it may contain an order-by operation over the sensitive field *alone*.

3 SECURITY GUARANTEES

3.1 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 (users, the application, and Arx's client proxy)—including not issuing queries through the client proxy—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, including not modifying the database or query results. The active attacker is part of our future work, and will likely leverage existing techniques [49, 58, 62, 88]. Further, in the Arx model, an attacker cannot inject queries because he does not have access to the client proxy, but only to the server.

We consider two models of passive attackers, snapshot and persistent attackers. The *snapshot* attacker corresponds to an attacker that manages to steal *one* snapshot of the database: the snapshot is made of collections and indices. It does not contain in-memory data related to the execution of current queries (which falls under the persistent attacker). This is an *offline* attacker in the sense that the attacker steals the database and works offline for as long as it wishes to extract data out of it. The *persistent* 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.

3.2 Security guarantees

Arx provides different guarantees for the two attacker models.

Snapshot attacker. Arx's most visible contribution over PPE-based EDBs is for this attacker. Such an attacker corresponds to many

real-world instances because many attacks are steal-and-run (e.g., hackers often extract a dump or part of the database). A cloud employee or insider could steal a copy of the database for offline analysis. This attacker encompasses the recent attacks of Durak et al. [33], Grubbs et al. [43], and Naveed et al. [66].

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

Comparison to PPE-based EDBs. 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 semantic secure schemes in Arx, in which the attacker does not see such relations. Hence, Arx protects against the recent offline attacks [33, 43, 66] that succeed in extracting significant information from PPE-based EDBs.

Persistent attacker. This attacker additionally watches *queries* and their execution. Arx hides the constants in the queries, but not the operations performed. Moreover, Arx does not hide timing information (e.g. the time when a query arrives), query metadata and access patterns (e.g. which positions in the database or index are accessed/returned and how many).

All known practical approaches leak some information due to queries. We specify Arx's leakage profile due to queries in Def. 3.3. Kellaris et al. [50] showed that *any generic encrypted database*, without Oblivious RAM [81] and output padding, leaks ordering information to a persistent attacker over time. To fully remove such leakage [50], the state-of-the-art techniques are Oblivious RAM [39, 81] or similar constructs [36] plus padding, which are known to be too slow, and in fact, might be worse than streaming all data locally as shown by Naveed [65].

Arx's goal for this attacker is to always remain more secure 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 not trivial: for example, a prior EDB aiming for semantic security [35] is not always more secure than PPE-based EDBs, as explained in §12.

We note that, in practice, mounting a persistent attack for a long time is significantly more difficult than a snapshot attack. For example, cloud providers have effective tracking systems to monitor what customer data a cloud employee is accessing, and effective intrusion detection monitors that detect logging (e.g., of queries) and data exfiltration [1, 2]. Avoiding such detection for a long time is significantly more unlikely. If the attacker sees only a few queries, the attacker does not learn much more than the snapshot attacker. Moreover, some client applications do not query every value in the DB for equality queries, and similarly for range queries, so the attacker will still not be able to gather as much information as for PPE-based EDBs. For example, some applications query only a small set of meaningful ranges on some data fields (e.g. age over 18 or over 21), compute aggregates over a range (in which case Arx does not reveal which documents are in the range), or order-by-limit which might include only a part of the data. Nevertheless, pushing the envelope further for the persistent attacker is an important problem and part of our future work.

3.3 Leakage definitions

We now explain Arx's security guarantees. A database DB is comprised of multiple collections. Each collection is comprised of a set of (attribute, keyword) pairs. A database system is a pair of stateful random access machines (Client, Server). Server stores the database and Client can through interaction with Server compute *queries* out of a set of supported queries (§2.3), which may modify the database.

Query execution leakage. We now define the leakage profile of Arx: first for the DB itself (snapshot attacker), then for the execution of each query (persistent attacker). The latter leakage includes (1) the IDs of the documents returned, (2) for Arx-Range (§5), the rank of the range bounds, and (3) for EQ (§4) and Arx-Eq (§6), the match patterns including for documents not in the database any more. Our leakage in Arx-Eq is similar to Sophos [25].

Definition 3.1 (Leakage of Database). The leakage of a database itself $\mathsf{Leak}(\mathsf{DB})$ is

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

Definition 3.2 (Preliminaries). We define the following:

- The rank of an element x in a list $L = (a_i)_{i \in \mathbb{N}}$ is $\mathsf{rk}(L, x) := |\{a_i \mid a_i \leq x\}|$, and we write $\mathsf{rk}(x)$ if L is clear from the context.
- Our leakage function Leak is stateful and stores a query history Q which contains tuples (i, w) for a query with timestamp i searching for a keyword w, and tuples (i, o, d) for an update query, where d is the data and $o \in \{\texttt{insert}, \texttt{delete}\}$.
- The search pattern of a keyword w is $\mathsf{sp}(w) := \{j : (j, w) \in Q\}$.
- The history of w is $\mathsf{Hist}(w) := (\mathsf{DB}_0(w), \mathsf{UpHist}(w))$ where $\mathsf{DB}_0(w)$ lists the matching document IDs of the initial database, and $\mathsf{UpHist}(w)$ is the list of document updates matching w.

Definition 3.3 (Leakage of a query). The leakage of a query is the entire query except for the constants in the query, the timestamp, the IDs of documents that match *any* filter, and the IDs of documents being created. Additionally there is leakage depending on the schemes used to execute the query, as detailed in Fig. 2.

Security definition. We provide an adaptive indistinguishability-based security definition for the overall system, which we call IND-CQA (i.e. indistinguishability under a chosen query attack). This definition, Def. 3.4, is quite standard and similar to definitions in prior work [35]. In the indistinguishability game, the attacker chooses two databases and queries to execute on them, with the requirement that the leakage function returns the same values in these two worlds. The game consists of two phases in order to accommodate for the model of the snapshot attacker: the first phase corresponds to the execution when the attacker does not see query

Scheme	Operation	Leakage
EQ (§4)	`where` $field = c$	sp(w), Hist(w)
	`insert`	sp(w)
	`delete`	–
Arx-Eq (§6)	`where` $field = c$	sp(w), Hist(w)
	`cleanup` (§6.3)	sp(w), Hist(w)
	`insert, delete`	–
Arx-Range (§5)	`where` $a \leq field \leq b$	rk($a-1$), rk(b)
	`orderby limit` ℓ	ℓ
	`insert, delete` v	rk(v)
Arx-Join (§8)	the same information as Arx-Eq or Arx-Range, depending on which Arx-Join was built on, as well as leakage as in EQ for the foreign key, where each match identifies a primary key.	

Figure 2: Query leakage in Arx's protocols.

execution, followed by a second phase when the attacker can see this execution.

Definition 3.4 (IND-CQA). Let (Client,Server) be a database system and Adv an adversary adaptively trying to distinguish two execution traces that have the same leakage. We consider the following experiment:

$\mathbf{Ind}_{\mathsf{Client,Server,Adv,Leak}}(1^\lambda)$**:**

(1) A random bit $b \leftarrow \{0,1\}$ is chosen.

(2) Adv generates two initial databases and a list of m queries for each:

$$(\mathsf{DB}^0, (q_1^0, \ldots, q_m^0), \mathsf{DB}^1, (q_1^1, \ldots, q_m^1)) \leftarrow \mathsf{Adv}(1^\lambda).$$

We require that both databases produce the same leakage after the queries are executed on them, i.e. $\mathsf{Leak}(\mathsf{DB}^0(q_1^0, \ldots, q_m^0)) = \mathsf{Leak}(\mathsf{DB}^1(q_1^1, \ldots, q_m^1))$.

(3) The queries q_i^b are executed on DB_b.

The next phase simulates the break-in of Adv. From now on, Adv is given access to Server's random coins and the communication transcript.

(4) Adv is given a copy of the state of Server and proceeds to adaptively generate pairs of queries (q_i^0, q_i^1) which are required to produce the same leakage.

(5) Query q_i^b is executed between Client and Server.

(6) After a polynomial number of rounds, Adv outputs a bit b'. The output of the experiment is 1 if $b = b'$, else 0.

In the case that the experiment outputs 1, we say Adv *wins* the experiment. We call the database system *adaptively secure in the indistinguishability sense* with leakage Leak, if for all polynomial Adv, there exists a negligible function negl such that

$$\Pr[\mathbf{Ind}_{\mathsf{Client,Server,Adv,Leak}}(1^\lambda) = 1] \leq 1/2 + \mathsf{negl}(\lambda)$$

Theorem 3.5. *The Arx database system is secure as defined in Def. 3.4 with leakage defined in Def. 3.1–Def. 3.3, under standard cryptographic assumptions.*

The proof strategy is to compose sub-proofs for the different Arx indices and schemes, as we discuss in §B.4. In appendix §B, we provide formal definitions and proofs for the security of Arx-Range because this is the most nonstandard of our schemes. The proofs for Arx-Eq and the other schemes are quite standard (similar to [25, 35]), and we plan to include them in a full paper version.

4 ENCRYPTION BUILDING BLOCKS

In addition to Arx's indices, Arx uses three semantically-secure encryption schemes. These schemes already exist in the literature, so we do not elaborate on them.

BASE is standard encryption, AES-CTR in our case.

EQ enables equality checks. The client uses $\mathsf{EQEnc}_k(v) \rightarrow \mathsf{ct}$ to encrypt a value v and $\mathsf{EQToken}_k(w) \rightarrow \mathsf{tok}$ to produce a token tok used to search for w. To search, the server uses $\mathsf{EQMatch}(\mathsf{ct}, \mathsf{tok})$, which returns true if $v = w$. To implement EQ, we use a searchable encryption scheme similar to those of Cash et al. [26] and Sherry et al. [78]. $\mathsf{EQEnc}_k(v) = (\mathsf{IV}, \mathsf{AES}_{\mathsf{KDF}_k(v)}(\mathsf{IV}))$, where IV is a random value and KDF is a key derivation algorithm based on AES. To search for a word w, the token is $\mathsf{EQToken}_k(w) = \mathsf{KDF}_k(w)$. 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 Arx-Eq index. Arx uses this scheme only for non-indexed fields (i.e. for linear scans). When the developer desires an index on a field with equality operations, Arx uses our new Arx-EQ index instead.

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: to check for equality, the server simply uses the equality operator, as if the data were not encrypted. Moreover, databases can build indices on this field as before so this case is an optimization for Arx-Eq too.

AGG enables addition using the Paillier scheme [70].

5 ARX-RANGE AND ORDER-BASED QUERIES

We now present our index enabling order operations.

5.1 Strawman

We begin by presenting a helpful but inefficient strawman. This strawman corresponds to the protocols in mOPE [75] and the startup ZeroDB [34]. For simplicity, consider the index to be a binary search tree (instead of a regular B+ tree). To obtain the desired security, each node in the tree is encrypted using a standard semantically secure 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 client with the root node, the client decrypts it into a value v, compares v to a, and tells the server whether to go to the left or to the right child. The server then provides the relevant child to the client, which again tells the server

which way to go. This procedure repeats until the server reaches a leaf node. However, this procedure is too slow because each level in the tree requires a roundtrip. Some web applications issue tens of queries for one user click. The ZeroDB developers confirmed that this is a main issue they are struggling with.

5.2 Non-interactive index traversal

Arx-Range 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. For 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 [36, 37], we store **a garbled circuit at each tree node** that performs the comparison, while hiding a and v from the attacker.

Using a garbling scheme (Garble, Encode, Eval) [38, 85], the client can invoke $(F, e) \leftarrow \mathsf{Garble}(1^\lambda, f)$ on a boolean circuit f to obtain a *garbled* version F along with encoding secret e. The algorithm $e_a \leftarrow \mathsf{Encode}(e, a)$ uses e to produce an encoding e_a for some input a. For each bit a_i in the input, e_a contains a label I_i^0 (if a_i is 0) or I_i^1 (if a_i is 1). Given encoding e_a, the server can invoke $y \leftarrow \mathsf{Eval}(F, e_a)$ on the garbled circuit to obtain $y = f(a)$. The security of garbled circuits guarantees that the server learns *nothing* about a or the data hardcoded in f other than the output $f(a)$. 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 is different from the encoding for its children. We therefore **chain the garbled circuits** so that each circuit outputs an encoding that can be used with the relevant child circuit.

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 implementing the comparison of the input with the hardcoded value v that 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'
end if

Fig. 3 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, 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.

Repairing the index. A part of our index gets destroyed during the traversal because each garbled circuit may be used at most once.

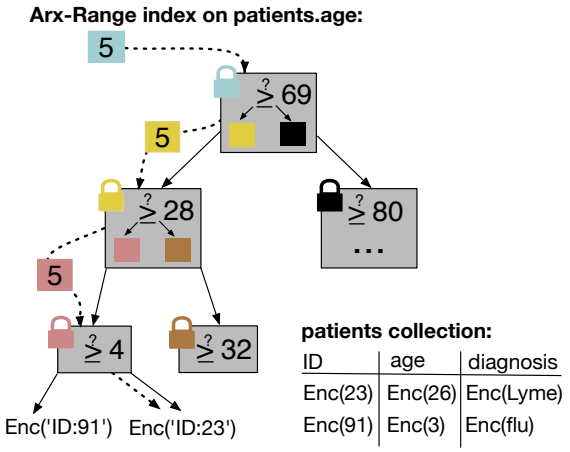

Figure 3: Arx-Range example. Enc **is encryption with** BASE.

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. Consider that a node N and its left child L were consumed. However, for each node N, the client needs two pieces of information from the server: the value 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 garbled circuit at R that together with the secret key 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).

5.3 A secure database index

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

First, the shape of the index should not leak information about the order in which the data was inserted. Hence, we use a history-independent treap [18] instead of a regular search tree. This data structure has the property that its shape is the same independent of the insertion or deletion order. One needs to be careful that when implementing a history-independent data structure the implementation of it is history-independent as well. For example, in our Java implementation, we have no control of where in memory the language runtime places certain data which depends on history. Implementing history-independent data structures [64] is orthogonal to our work.

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.

Running queries using the index. Consider the query `select * from patients where 1 < age ≤ 5`. Each node in the index tree has two garbled circuits to allow concurrent search for the lower and upper bounds. The client proxy provides tokens for the 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 the nodes in between. The server sends this information to the client proxy which decrypts them, randomizes their order, and then selects the documents based on these primary keys from the server. The randomization *hides from the server the order of the documents matching the range.*

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

Updating the index. For inserts and deletes, the server traverses the index to the appropriate position, performs the operation, and balances the index as required. For updates, the server first performs a delete followed by an insert. Some update or delete queries delete by filtering on a different field. In this case, we maintain an encrypted backwards pointer from the document in a collection to the corresponding leaf in the tree.

A common case are *monotonic inserts*: inserts in increasing or decreasing order. In such cases, 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.

5.4 Optimizations

We employ several techniques to improve the performance of Arx-Range. The highlights are: (1) we chain garbled circuits together using *transition tables* instead of computing the encoding function inside the circuit; (2) we make the index concurrent by caching the top few levels of the tree at the client proxy; and (3) we incorporate recent advances in garbling in order to make our circuits short and fast. We defer the details to §A.1.

6 ARX-EQ AND EQUALITY QUERIES

The Arx-Eq index enables equality queries and builds on insights from the searchable encryption literature [24], as explained in §12. We aim for Arx-Eq to be *forward private*, a property shown to increase security significantly in this context [25]: the server cannot use an old search token on newly inserted data. We begin by presenting a base protocol that we improve in stages.

6.1 Base protocol

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

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

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

Encrypt and insert. Suppose the application performs an insert for a document where age has value v. The client proxy first increments counter$[v]$. Then, the proxy encrypts v into:

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

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: counter$[v]$. This encryption is not decryptable, but as discussed in §9.2, Arx encrypts v with BASE as well. The document with the encryption of v is then inserted in the database.

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

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

Note that this provides forward privacy: the server cannot use an old search token to learn if the new values are equal to v because the new values would have a higher counter.

6.2 Reducing the work of the client proxy

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

Instead of encrypting a value v as in Eq. (1), the client proxy hashes according to the tree in Fig. 4. 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 $0, 1, 2, 3, \ldots, \mathsf{counter}[v]$.

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

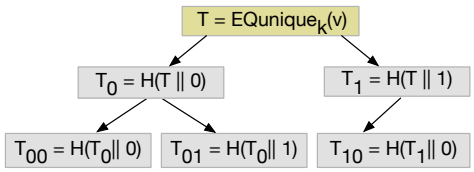

Figure 4: Search token tree.

New search token. To search for a value v with counter counter$[v]$, the client proxy computes the *covering set* for leaf nodes $0, \ldots,$ counter$[v] - 1$. The covering set is the set of internal tree nodes

whose subtrees cover exactly the leaf nodes $0, \ldots, \text{counter}[v] - 1$. For the example in Fig. 4, $\text{counter}[v] = 3$ and the covering set of the three leaves is node T_0 and node T_{10}. The search token are the nodes in the covering set. The covering set can be easily deduced from the binary representation of $\text{counter}[v] - 1$.

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

6.3 Updates

We have already discussed insert. To delete a document, Arx simply deletes this document. An update is a delete followed by an insert.

As a result, encrypted values for some counters will not return matches during search. This does not affect 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, Arx-Eq 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 $\text{counter}[v]$ with ct, chooses a new key k' for v, and generates new tokens as in Fig. 4: $T'_{00}, \ldots, T'_{\text{ct}}$ using k'. It gives these tokens to the server, which replaces the fields found matching with these.

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

6.4 Arx-Eq counter map

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

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

To avoid leaking the number of distinct fields, Arx pads the counter map to the number of documents in the relevant collection. The security of this scheme satisfies Arx's goal in §3: a stolen database remains encrypted with semantic security and leaks nothing other than size information.

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

Moreover, there are many fields for which the counter map is very small (e.g. gender, age, letter grades). Furthermore, when all values are unique (the maximum size for a counter map), Arx-Eq defaults to the regular index built over EQunique encryptions, not needing any counter map. The case when there are many distinct values with few repetitions is less ideal, and we implement an optimization for this case: to decrease the size of the counter map, Arx groups multiple entries into one entry by storing their prefixes. As a tradeoff, the client proxy has to filter out some results.

7 AGGREGATION QUERIES USING ARX-AGG

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(daysInHospital) from patients where` $70 \leq$ `age` ≤ 80.

The idea behind aggregations in Arx is inspired from literature on authenticated data structures [59]. This work targets integrity guarantees (not confidentiality), but interestingly, we use it for computations on encrypted data. Consider the Arx-Range index in Fig. 3 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 daysInHospital corresponding to the leaves under N. The root node thus contains the sum of all values. This value is stored encrypted with BASE.

When the server needs to compute the sum over an arbitrary range, such as $[70, 80]$, the server locates the edges of the range as before, and then it identifies a *perfectly covering set*. Note that this set of nodes is *logarithmic* in the size of the index. For each node in the covering 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 (1) inserting/deleting a document, or (2) modifying a field having an aggregate, the partial sums on the path from N to the root need to be updated, where N is the node corresponding to the changed document. In the second case, the client also needs to repair the path in the tree, so the partial sum update happens essentially for free.

This aggregation strategy supports any aggregation function of the form $\sum F(doc)$ where F is an arbitrary function whose input is a document, as explained in §2.3. For aggregates over fields with an Arx-Eq index, we have a similar strategy to the aggregates over a range, but we do not describe it here due to space constraints. For all other cases, we use AGG . However, the number of such cases is reduced significantly.

8 JOINS USING ARX-JOIN

We now describe how Arx supports a common class of join operations, namely, foreign-key joins. Arx extends Arx-Eq or Arx-Range 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 Arx-Eq or Arx-Range index. The query may additionally filter the joined documents in C2 using equality operations, denoted by *eq*(C2).

Arx-Eq-based joins. Consider an example with collection C2 having a primary key ID, and collection C1 having a field age with Arx-Eq, 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 Arx-Eq, the client proxy now uses two keys k_1 and k_2. It generates a token for each key as before: t_1 and t_2. The client proxy includes t_1 in the document as before, and uses t_2 to encrypt the diagnosis 'flu' as in: $J = \mathsf{BASE}_{t_2}(\mathsf{EQunique}(\text{'flu'}))$. J will help with the join. Hence, upon insert, the pair (10, 'flu') becomes (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 Arx-Eq. It locates the documents of interest using t_1, and then uses t_2 to decrypt J and obtain EQunique('flu'). This value is a primary key in 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 §4. Note that this additional filtering cannot make use of an index; hence, it is restricted to equality predicates and may not contain range operations.

Arx-Range-based joins. Arx employs a different strategy in case the where clause of the join query requires an Arx-Range index for execution, e.g. `where C1.age > 10`. In such a scenario, Arx-Join'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 Arx-Range index over C1.age, which already contain the encrypted primary keys of documents in C1 (as described in §**??**). While traversing the index in order to resolve the where clause, the server fetches the encrypted foreign keys as well from the nodes of interest, and sends them to the client proxy for decryption as with regular Arx-Range. The client decrypts the encrypted foreign keys, re-encrypts them with EQunique, shuffles them, and returns them to the server. The server then uses these values to locate the corresponding documents in 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 Arx-Join. Updates to the foreign key C1.fkey simply update the underlying index, Arx-Eq or Arx-Range. 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.

9 ARX'S PLANNER

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

9.1 Index planning

Before deciding what index to build, note that Arx-Range and Arx-Eq support *compound* indices, which are indices on multiple fields. For example, an index on (diagnosis, age) enables a quick search for `diagnosis = 'flu' and age ≥ 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 Arx-Range index for equality operations because of its higher cost and different security.
- Unlike a regular index, a compound Arx-Eq index on (a, b) cannot be used to compute equality on a alone because Arx-Eq performs a complete match.
- A range or order-by-limit on a sensitive field can be computed only via an Arx-Range index, so it can no longer be computed after applying a separate index.

All these are further complicated by the fact that the application admin can explicitly specify certain fields to be nonsensitive (as described in §2.2), 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 Arx-Range index on a composition of a nonsensitive and a sensitive field. Consider, for example, that the developer built an index on a, a nonsensitive

field, and wants to perform a query containing `where` a = `and` $s \geq$, where s is sensitive. The developer expects the DB to filter documents by a rapidly based on the index, and then, to filter the result by "$s \geq$".

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

Despite all these constraints, our index planning algorithm is quite simple. It also applies to queries that have multiple query plans using different indices, in which case it maintains the asymptotics of every query plan. The index planner runs in two stages: per-query processing and global analysis. Only the `where` clauses (including order-by-limit operations) matter here. The first stage of the planner treats sensitive and nonsensitive fields equally.

Example: For clarity, we use three query patterns as examples. Their `where` clauses are: W_1: "$a =$ `and` $b =$", W_2: "$x =$ `and` $y \geq$ `and` $z =$". The indices specified by the developer are on x and (a, b).

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

Then, if W_i contains a sensitive field with a range or order-by-limit operation, append a "$\geq$" filter on this field to each member of S_i, if the member does not already contain this. Based on the constraints in §2.3, a where clause cannot have more than one such field. Example: For W_1, $S_1 = \{(a =, b =)\}$, and for W_2, $S_2 = \{(x =, y \geq)\}$.

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

Example: If a and b are nonsensitive, and S contains both $(a =, b =)$ and $(a =, b \geq)$, then $(a =, b =)$ is removed. If all of a, b and c are sensitive and S contains both $(a =, b =, c \geq)$ and $(a =, b \geq)$, then $(a =, b \geq)$ is removed. If b and y are sensitive (a, x, z can be either way), for S above, the indices Arx builds are: Arx-Eq (a, b) and Arx-Range (x, y).

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

9.2 Data layout

Next, laying out the data encryption plan is straightforward:

- All values of a sensitive field are encrypted with the same key, but this key is different from field to field.
- For every aggregation in a query, decide if the `where` clause in this query can be supported entirely by using Arx-Range or Arx-Eq. Concretely, the `where` clause should not filter by additional fields not present in the index. If so, update the metadata of the respective index to follow our index aggregation strategy described in §7. If not, encrypt the respective fields with AGG if the aggregate requires the computation of a sum.
- For every sensitive field projected by at least one query, encrypt it with BASE, because EQ cannot decrypt.
- For every query pattern, if the `where` clause W_i performs an equality on a field that is not part of every element of S_i, encrypt the field with EQ. The reason is that at least one query plan will need to filter this field by equality without an index.

10 IMPLEMENTATION

While the design of Arx is decoupled from any particular DBMS, we implemented our prototype for MongoDB 3.0, one of the most popular NoSQL data stores. Arx's implementation consists of ~11.5K lines of Java, and ~600 lines of C/C++ code. We used the Netty I/O framework [7] to implement Arx's proxies. Additionally, we implemented a C++ library for garbled circuits, ArxGarble, in ~1200 LoC. We plan to release the source code of both Arx and ArxGarble.

11 EVALUATION

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

11.1 Functionality

To understand if Arx supports real applications, we evaluate Arx on seven existing applications built on top of MongoDB. 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. Some fields are clearly sensitive (heart rate, private messages), but other fields are less clearly so, such as timestamps. Nevertheless, Arx encrypts all fields in these applications, which is the default in Arx.

Fig. 5 summarizes our results. With regard to unsupported queries across the applications, 4 of the 11 were due to timestamps, which are less sensitive in nature (and might therefore be explicitly specified as nonsensitive by the application admin). The limitation was the number of range/order operations Arx allows in the query, as explained in §2.3. For NodeBB, the two unsupported queries performed text searches, and for Leanote, the five queries were evaluating regular expressions on fields, both of which Arx cannot support. Nevertheless, 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

Application	Examples of fields	Unsupported queries		No. of	No. of	Total indices	
		Total	Excl. timestamps	Arx-Eq	Arx-Range	Vanilla	Arx
ShareLaTeX [12]	document lines, edits	1	–	12	4	12	16
Uncap (medical) [14]	heart rate, medical tests	–	–	0	2	2	2
NodeBB (forum) [8]	posts, comments	2	2	13	4	12	17
Pencilblue (CMS) [9]	articles, comments	3	–	46	27	70	73
Leanote (notes) [6]	notes, books, tags	5	5	64	28	69	92
Budget manager [3]	expenditure, ledgers	–	–	5	0	5	5
Redux (chat) [10]	messages, groups	–	–	3	0	3	3

Figure 5: Examples of applications supported by Arx: examples of fields in these applications (Arx encrypts all fields in the 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 Arx-Agg is built on Arx-Eq and Arx-Range, 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 Arx-Eq operations in μs

consider if the unsupported data field is nonsensitive and mark it as such, but this should be done with care.

The table also shows that though Arx's index planner increases the number of indices by 20%, this number does not blow up. The main reason is that the number of fields that both have order queries and are not indexed by the application is small.

11.2 Performance evaluation setup

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

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

11.3 Encryption schemes microbenchmarks

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

11.4 Performance of Arx-Eq

The Arx-Eq 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 §6. 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 Arx-Eq 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 that the client proxy does little work in comparison to the server.

We now evaluate the overall performance of Arx-Eq (without the optimization for unique values) using relevant queries issued by ShareLaTeX. These queries filter by one field using Arx-Eq, allowing us to focus on Arx-Eq. We first loaded the database with 100K documents representative of a ShareLaTeX workload.

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

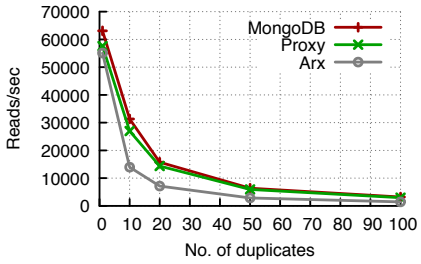

Figure 8: Arx-Eq read throughput with increasing no. of duplicates.

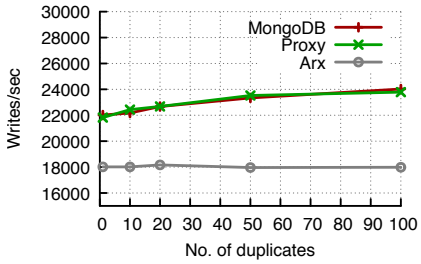

Figure 9: Arx-Eq write throughput with increasing no. of duplicates.

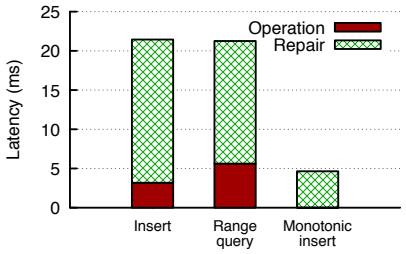

Figure 10: Arx-Range latency of reads and writes.

Fig. 9 compares the write throughput of Arx-Eq 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 Arx-Eq index are independent of the number of duplicates by virtue of security: each value looks different. Further, since each individual insert requires the computation of a single token (a constant-time operation), the write throughput of Arx-Eq remains stable in this experiment. As a result, the net overhead grows from 18% (when fields are unique) to 25% when there are 100 duplicates per value.

Latency follows a similar trend and remains stable for Arx-Eq at ~3.3ms. For a regular MongoDB index, the latency slowly improves from ~2.7ms to ~2.5ms as the number of duplicates grows to 100.

11.5 Performance of Arx-Range

Our garbled circuits are implemented in AES, which takes advantage of existing hardware implementations. For a 32-bit value, the garbled circuit is 3088 bytes long, the time to garble is ~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.

We now evaluate the latency introduced by Arx-Range. We pre-inserted 1M values into the index, and assumed a length of 128 bits for the index keys, 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 different operations on the index. Fig. 10 illustrates the latency of each operation, divided into two parts: (1) the time taken to perform the operation, and (2) the time taken to repair the index. The generation of fresh garbled circuits in order to repair the index contributes the most towards latency.

Range queries cost more than writes because the former traverse two paths in the index (for bounded queries), while the latter traverse a single path. The latency for a range query is ~6ms. We note that using the strawman in §5.1, one incurs a roundtrip overhead for each node in the path, which is roughly as long our entire query. Fig. 10 also highlights the improvement when the index can be optimized for monotonic inserts, which was common in the applications we evaluated. In fact, all three Arx-Range indices maintained for ShareLaTeX could be optimized for monotonic inserts, since the indexed fields included timestamps and version numbers.

11.6 Performance of Arx-Agg

Computing an aggregate over a range with Paillier as in CryptDB [76] takes significantly longer than with Arx. In Arx, this cost is essentially zero because traversing the index for a range query automatically computes the cover set. In CryptDB, one has to do a homomorphic multiplication for every value in the range. As a result, aggregating over a range size of 10,000 values, Arx takes ~0 ms for the aggregate and CryptDB takes 80ms. With the cost of the range, Arx is 13 times faster.

11.7 Performance of Arx-Join

Arx-Join builds on top of Arx-Eq and Arx-Range. As a result, the performance of joins is closely tied to the performance of the underlying index. In this section, we report only on the additional performance overhead Arx-Join brings.

For joins based on Arx-Eq, the client proxy computes two sets of covers instead of one, thereby incurring an additional latency of 14.5μs for a token tree of height 10 (Fig. 7). The server proxy expands the additional set, and uses it to decrypt the foreign key pointers. Therefore, the latency at the server increases by (i) the cost of expanding the second covering set (Fig. 7), and (ii) the cost of decrypting the foreign key pointers in the filtered documents (Fig. 6). As a result, a query that joins 10,000 documents increases the latency of Arx-Eq by 542.3μs +10,000 $\times$ 0.13μs = 2.38ms at the server.

Joins over Arx-Range are executed by adding the encrypted foreign key pointers to the index nodes, which are sent to the client proxy for decryption. Such joins increase the latency of Arx-Range by the time taken by the client proxy to decrypt the foreign key pointers. As a result, a join over 10,000 documents takes 10,000 $\times$ 0.13μs = 1.3ms longer.

11.8 Comparison to CryptDB

Comparing functionality in a strict sense, Arx supports fewer queries than CryptDB, but we find that their functionalities are nevertheless comparable and that Arx also enables a rich class of applications as shown above. Arx does not support group-by operations (for security issues), arbitrary conjunctions of range filters, and more generic joins, supported by CryptDB. For example, CryptDB supports all queries in the TPC-C benchmark [13], a widely used industry benchmark, while Arx supports 30 out of 31 queries.

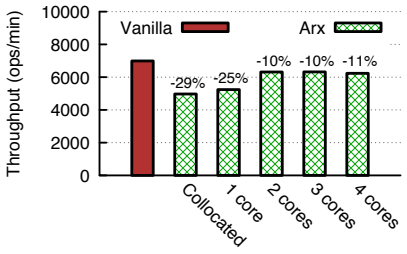

Figure 11: ShareLaTeX performance with Arx's client proxy on varying cores

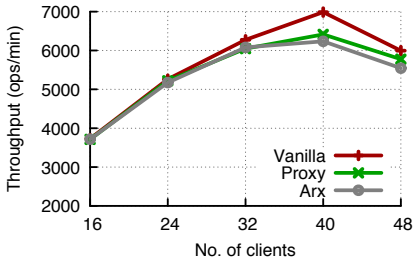

Figure 12: ShareLaTeX performance with increasing no. of client threads

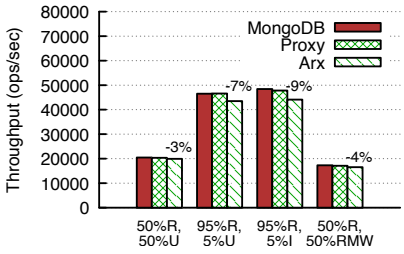

Figure 13: YCSB throughput for different workloads.

For performance, we cannot compare Arx to CryptDB directly because CryptDB is implemented on MySQL and applications vary. On one hand, CryptDB's order queries via order-preserving encryption are faster than Arx's, but this scheme is significantly less secure. On the other hand, Arx's aggregate over a range is an order of magnitude faster than CryptDB's for the same security, as evaluated in §11.6. Overall, Arx is a heavier solution due to the significant extra security, but it remains in the same category as CryptDB in terms of the performance impact it has on applications: both Arx and CryptDB report a throughput overhead on the order of 10%, and an added latency on the order of milliseconds per operation.

11.9 End-to-end evaluation

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

Evaluation on ShareLaTeX. We evaluate the end-to-end overhead of Arx using ShareLaTeX [12], 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, document history, 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 [11]), multiple clients launch browsers in parallel and collaborate on projects, continuously editing documents and exchanging messages via chat. Fig. 11 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, but the performance improves considerably when two separate cores are allocated to Arx's client proxy, in which case the reduction in throughput stabilizes at a reasonable 10%.

Fig. 12 compares the performance of Arx with increasing load at the application server, when four separate cores are allocated to Arx's client proxy. It also shows the performance of MongoDB with the Netty [7] proxy without the Arx hooks. Note that each client thread issues many requests as fast as it can, bringing a load equivalent to many real users. At peak throughput with 40 client threads and 100% CPU load at the application, the reduction in performance owing to Arx is 11%, of which 8% is due to Arx's proxy, and the remaining 3% due to Arx's encryption and indexing schemes.

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

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. Second, even though Arx-Range is not cheap, Arx-Range is just one operation out of a set of multiple operations Arx runs, with the other operations being faster and overall more common, such as Arx-Eq and the building blocks in §4.

YCSB Benchmark. Since Arx is a NoSQL database, we also evaluate its overhead on the YCSB benchmark [31] running against the client proxy. We first 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 Arx-Eq index and the rest of the fields are encrypted with BASE.

Fig. 13 shows the average performance of Arx versus vanilla MongoDB, across different workloads with varying proportions of reads and writes. In the figure, "R" refers to proportion of reads, "U" to updates, "I" to inserts, and "RMW" to read-modify-write. The reduction in throughput is higher for read-heavy workloads as a result of the added latency due to sequential decryption of the result sets. Overall, the overhead of Arx ranges from 3%-9%, based on the workload. Increase in latency due to Arx is also unremarkable—for example, average read latency increases from 2.31ms to 2.43ms under peak throughput, while average update latency increases from 2.36ms to 2.47ms in the 50% read-50% update workload. Arx's performance on YCSB is high because YCSB conforms to Arx-Eq's optimized case when fields are unique. In general, this shows that indexing primary keys is fast with Arx.

11.10 Storage

Arx increases the amount of data stored in the database for the following reasons: (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 Arx-Eq indexing. Further, Arx-Range indices are larger than regular B+Trees, because each node in the index tree stores garbled circuits. Vanilla ShareLaTeX with 100K documents per collection required 0.56GB of storage in MongoDB, with an additional 48.7 MB for indices. With Arx, the data storage increased by ~1.9× to 1.05GB for the reasons described above. The application required three compound Arx-Range indices, which together occupied 8.4GB of memory at the server proxy while indices maintained by the database occupied 56.5MB. This resulted in a net increase of ~16× at the DB server, for storing the encrypted data along with Arx's indices. However, there remains substantial scope for optimizing the size of the indices in our implementation. For example, a serialized dump of the three Arx-Range indices occupied 3.1GB of memory. Moreover, our choice of workload did not include large data items such as images or videos, which typically would not require indexing by Arx-Range. In such cases, the overhead imposed by Arx would proportionately decrease.

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

12 RELATED WORK

We divide related work into two categories: (1) we compare Arx with state-of-the-art EDBs, and (2) we discuss protocols related to Arx's building blocks, Arx-Eq and Arx-Range.

12.1 Encrypted databases

Early approaches [44] used heuristics instead of encryption schemes with provable security. Regarding PPE-based EDBs [19, 71, 76, 83], we have already compared Arx against them extensively in §1 and §3.2. 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 [26, 35] is based on searchable encryption. The main drawback is that it is too restricted in functionality. It does not support (i) joins; (ii) order-by-limit queries, which are commonly used for pagination (more common than range queries in TPC-C [13]); and (iii) aggregates over a range because the range identifies a superset of the relevant documents for security, yielding an incorrect aggregate. Further, inserts, updates and deletes are not efficient as discussed in §12.2. Regarding security, while being significantly more secure than PPE-based EDBs for a snapshot attacker, for persistent 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. In Arx, we address all these aspects. Similarly, concurrent works [47, 52] support equality-based queries but are too restricted in functionality. They do not support range, order-by-limit, or aggregates over range queries, and the former does not support updates and inserts 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. The drawbacks of BlindSeer with respect to Arx are performance and functionality. BlindSeer requires a high number of interactions between the client and the server. For example, for a range query, the client and the 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. Additionally, BlindSeer does not handle inserts easily, and does not support deletes (also needed for updates), which are crucial for the applications targeted in Arx. BlindSeer's Bloom filter approach is fundamentally not fit for deletes/updates. It also does not handle aggregates over ranges or order-by-limit.

12.2 Work related to our building blocks

Work related to Arx-Eq. Arx-Eq 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 [24–26, 32, 35, 45, 48, 54, 55, 67, 68, 79, 80], none of them meet the following desired security and performance from a database index. Besides semantic security, when inserting a field, the access pattern should not inform the attacker of what other fields the field equals, and an old search token should not allow searching on newly inserted data (forward privacy), both crucial in reducing leakage [25]. Second, inserts, updates and deletes should be efficient and should not cause reads to become slow. Arx-Eq meets all these goals. Perhaps the closest prior work to Arx-Eq is [26]. 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 does not have forward privacy and the revocation lists leak various metadata. Work concurrent to Arx-Eq, Sophos [25] also provides forward privacy, but Sophos uses expensive public key cryptography, instead of symmetric key as in Arx.

Work related to Arx-Range. There has been a significant amount of work on order-preserving encryption (OPE) both in the research community [15, 16, 21, 22, 46, 56, 60, 61, 69, 75, 84, 86] and in industry [17, 29, 77]. OPE schemes are efficient but leak a significant amount of information [66]. Order-revealing encryption (ORE) provides semantic security [23, 27, 57]. The most relevant of these is the construction by Lewi and Wu [57] that is more efficient than Arx-Range because it does not need replenishment, but is also less secure because it leaks the position where two plaintexts differ. As a result, it is not strictly more secure than OPE.

13 CONCLUSION

We presented Arx, a practical and functionally-rich database system that encrypts data only with semantically secure schemes. Arx provides significantly stronger security than databases based on property preserving encryption. It supports real applications with a modest performance overhead.

REFERENCES

[1] 2016. Amazon Web Services: Overview of Security Processes. (2016). http://d0.awsstatic.com/whitepapers/Security/AWSSecurityWhitepaper.pdf.

[2] 2016. Google Cloud Platform: Security Whitepaper. (2016). https://cloud.google.com/security/whitepaper.

[3] 2017. Budget Manager. (2017). https://github.com/kdelemme/budget-manager/.

[4] 2017. Chino.io: Security and Privacy for Health Data in the EU. (2017). https://chino.io/.

[5] 2017. iQrypt: Encrypt and query your database. (2017). http://iqrypt.com/.

[6] 2017. Leanote. (2017). https://leanote.com/.

[7] 2017. Netty Project. (2017). http://netty.io/.

[8] 2017. NodeBB. (2017). https://nodebb.org/.

[9] 2017. PencilBlue. (2017). https://pencilblue.org/.

[10] 2017. Redux. (2017). https://github.com/raineroviir/react-redux-socketio-chat/.

[11] 2017. Selenium. (2017). http://www.seleniumhq.org/.

[12] 2017. ShareLaTeX. (2017). https://www.sharelatex.com/.

[13] 2017. TPC-C Transation Processing Benchmark. (2017). http://www.tpc.org/tpcc/.

[14] 2017. UNCAP: Ubiquitous iNteropable Care for Ageing People. (2017). http://www.uncap.eu/.

[15] Divyakant Agrawal, Amr El Abbadi, Faith Emekci, and Ahmed Metwally. 2009. Database Management as a Service: Challenges and Opportunities. In *Proceedings of the 25th International Conference on Data Engineering (ICDE)*. Shanghai, China.

[16] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. Paris, France.

[17] George Weilun Ang, John Harold Woelfel, and Terrence Peter Woloszyn. 2012. System and Method of Sort-Order Preserving Tokenization. US Patent Application 13/450,809. (2012).

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

[19] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. 2013. A secure coprocessor for database applications. In *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications (FPL)*. Porto, Portugal.

[20] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of Garbled Circuits. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC.

[21] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*. Cologne, Germany.

[22] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Proceedings of the 31st International Cryptology Conference (CRYPTO)*. Santa Barbara, CA.

[23] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. 2014. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.

[24] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2014. A Survey of Provably Secure Searchable Encryption. *ACM Computing Surveys (CSUR)* (2014).

[25] Raphael Bost. 2016. Sophos - Forward Secure Searchable Encryption. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[26] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[27] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. 2016. Practical Order-Revealing Encryption with Limited Leakage. In *Proceedings of the 23rd International Conference on Fast Software Encryption (IACR-FSE)*.

[28] CipherCloud. 2017. Cloud Data Protection Solution. (2017). http://www.ciphercloud.com.

[29] CipherCloud. 2017. Tokenization for Cloud Data. http://www.ciphercloud.com/tokenization-cloud-data.aspx. (2017).

[30] Cloud Threat Intelligence. 2017. Skyhigh Cloud Security labs, Skyhigh Networks. (2017). https://www.skyhighnetworks.com/.

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2011. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*. Cascais, Portugal.

[32] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. 79–88.

[33] F. Betül Durak, Thomas M. DuBuisson, and David Cash. 2016. What Else is Revealed by Order-Revealing Encryption?. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[34] Michael Egorov and MacLane Wilkison. 2016. ZeroDB white paper. *CoRR* abs/1602.07168 (2016). http://arxiv.org/abs/1602.07168.

[35] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*. Vienna, Austria.

[36] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. 2015. Black-Box Garbled RAM. In *Proceedings of the 56th Annual Symposium on Foundations of Computer Science (FOCS)*.

[37] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2015. TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption. (2015). Cryptology ePrint Archive, Report 2015/1010.

[38] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*. New York, NY.

[39] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996).

[40] Google. 2017. Encrypted BigQuery client. (2017). https://github.com/google/encrypted-bigquery-client.

[41] Tim Greene. 2015. Biggest data breaches of 2015. (2015). http://www.networkworld.com/article/3011103/security/biggest-data-breaches-of-2015.html.

[42] Patrick Grofig, Martin Haerterich, Isabelle Hang, Florian Kerschbaum, Mathias Kohler, Andreas Schaad, Axel Schroepfer, and Walter Tighzert. 2014. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Lecture Notes in Informatics, Sicherheit*.

[43] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2016. Leakage-Abuse Attacks against Order-Revealing Encryption. Cryptology ePrint Archive, Report 2016/895. (2016). http://eprint.iacr.org/2016/895.

[44] Hakan Hacigumus, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. Madison, WI.

[45] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Xiaodong Song. 2014. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1028–1039.

[46] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2010. MV-OPES: Multivalued-Order Preserving Encryption Scheme: A Novel Scheme for Encrypting Integer Value to Many Different Values. *IEICE Trans. on Info. and Systems* (2010).

[47] Seny Kamara and Tarik Moataz. 2016. SQL on Structurally-Encrypted Databases. Cryptology ePrint Archive, Report 2016/453. (2016). http://eprint.iacr.org/.

[48] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*. 965–976.

[49] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-End Integrity Protection for Web Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*.

[50] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[51] Jeremy Kepner, Vijay Gadepally, Peter Michaleas, Nabil Schear, Mayank Varia, Arkady Yerukhimovich, and Robert K. Cunningham. 2014. Computing on Masked Data: A High Performance Method for Improving Big Data Veracity. *CoRR* (2014).

[52] Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang. 2016. Better Security for Queries on Encrypted Databases. Cryptology ePrint Archive, Report 2016/470. (2016). http://eprint.iacr.org/.

[53] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *Proceedings of the 8th International Conference on Cryptology and Network Security (CANS)*.

[54] Kaoru Kurosawa. 2014. Garbled Searchable Symmetric Encryption. In *Financial Cryptography and Data Security (FC) - 18th International Conference*. 234–251.

[55] Billy Lau, Simon P. Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. 2014. Mimesis Aegis: A Mimicry Privacy Shield-A System's Approach to Data Privacy on Public Cloud. In *Proceedings of the 23rd USENIX Security Symposium*. 33–48.

[56] Seungmin Lee, Tae-Jun Park, Donghyeok Lee, Taekyong Nam, and Sehun Kim. 2009. Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases. *IEICE Trans. on Info. and Systems* (2009).

[57] Kevin Lewi and David J. Wu. 2016. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. (2016).

[58] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2010. Authenticated Index Structures for Aggregation Queries. *ACM Trans. Inf. System Security* (2010).

[59] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2010. Authenticated Index Structures for Aggregation Queries. *ACM Transactions on Information and System Security* 13, 4 (2010).

[60] Dongxi Liu and Shenlu Wang. 2012. Programmable Order-Preserving Secure Index for Encrypted Database Query. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*. Honolulu, HI.

[61] Dongxi Liu and Shenlu Wang. 2013. Nonlinear order preserving index for encrypted database query in service cloud environments. *Concurrency and Computation: Practice and Experience* (2013).

[62] Ralph Merkle. 1979. *Secrecy, authentication and public key systems / A certified digital signature*. Ph.D. Dissertation. Stanford University.

[63] Microsoft SQL Server 2016. 2017. Always Encrypted Database Engine. (2017). https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[64] Moni Naor and Vanessa Teague. 2001. Anti-persistence: History Independent Data Structures. In *Proceedings of the 33rd ACM Symposium on Theory of Computing (STOC)*. Crete, Greece.

[65] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Report 2015/668. (2015). http://eprint.iacr.org/2015/668.

[66] Muhammad Naveed, Seny Kamara, and Chares V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, CO.

[67] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. 2014. Dynamic Searchable Encryption via Blind Storage. In *IEEE Symposium on Security and Privacy, (SP)*. 639–654.

[68] Wakaha Ogata, Keita Koiwa, Akira Kanaoka, and Shin'ichiro Matsuo. 2013. Toward Practical Searchable Symmetric Encryption. In *Advances in Information and Computer Security - 8th International Workshop on Security, IWSEC*. 151–167.

[69] Gultekin Özsoyoglu, David A. Singer, and Sun S. Chung. 2003. Anti-Tamper Databases: Querying Encrypted Databases. In *Proceedings of the 17th IFIP WG 11.3 Working Conference on Database and Applications Security (DBSec)*. Estes Park, CO.

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

[71] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[72] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (IEEE S&P)*.

[73] Fred Pennic. 2015. Anthem suffers the largest healthcare data breach to date. (2015). http://hitconsultant.net/2015/02/05/anthem-suffers-the-largest-healthcare-data-breach-to-date/.

[74] Raluca Ada Popa. 2014. *Building Practical Systems that Compute on Encrypted Data*. Ph.D. Dissertation. MIT.

[75] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. 2013. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*. San Francisco, CA.

[76] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal.

[77] Fahmida Y. Rashid. 2011. Salesforce.com Acquires SaaS Encryption Provider Navajo Systems. *eWeek.com* (2011).

[78] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 26th ACM Special Interest Group on Data Communication Conference (SIGCOMM)*.

[79] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (IEEE S&P)*. Oakland, CA.

[80] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS*.

[81] Emil Stefanov, Elaine Shi, and Dawn Song. 2012. Towards Practical Oblivious RAM. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[82] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM SIGSAC Conference on Computer and Communications Security*. 299–310.

[83] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*. Riva del Garda, Italy.

[84] Liangliang Xiao, I-Ling Yen, and Dung T. Huynh. 2012. Extending Order Preserving Encryption for Multi-User Systems. Cryptology ePrint Archive, Report 2012/192. (2012).

[85] Andrew C. Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*.

[86] Dae Yum, Duk Kim, Jin Kim, Pil Lee, and Sung Hong. 2011. Order-Preserving Encryption for Non-uniformly Distributed Plaintexts. In *Intl. Workshop on Information Security Applications*.

[87] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.

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

A FURTHER DESIGN DETAILS

A.1 Arx-Range Optimizations

A.1.1 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 $a \leq v$, and outputs a key k_L or k_R based on the result of the comparison.[1]

For each bit i of the input, the transition table stores four ciphertexts. Let I_i^0, I_i^1 denote the i-th input labels in the encoding e_a for the current circuit; let L_i^0, L_i^1 be the corresponding labels for the left child, and R_i^0, R_i^1 for the right child. The table stores the four ciphertexts shown below. Here E denotes a double-key-cipher implemented as $E(A, B, X) = O(A||B) \oplus X$, where O 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.

$$\begin{Bmatrix} E(k_L, I_i^0, L_i^0) \\ E(k_L, I_i^1, L_i^1) \end{Bmatrix} \quad \begin{Bmatrix} E(k_R, I_i^0, R_i^0) \\ E(k_R, I_i^1, R_i^1) \end{Bmatrix}$$

The values in the transition table are not stored in a fixed order. Instead, the point-and-permute technique [20] is employed, which means that if the least significant bit of I_i^0 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 §B.1.

A.1.2 Concurrency. Arx-Range provides limited concurrency because each index node needs to be repaired before it can be used

[1] This key is the label of the output wire in the instantiation of the scheme.

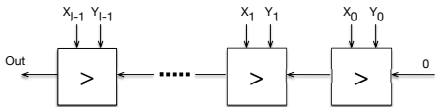

Figure 14: Schematic diagram of a garbled comparison circuit

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 §5.3 avoids the tree traversal and the repair. Of course, queries to another Arx-Range index or to other parts of the database can proceed in parallel.

A.1.3 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 [53], 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 [87] 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.

We now describe the design of our comparison circuit in greater detail.

Short comparison circuit. An ℓ-bit comparison circuit takes as input two ℓ-bit integers $a = (a_{\ell-1}, \ldots, a_0), b = (b_{\ell-1}, \ldots, b_0)$ represented in binary with the most significant bit first and outputs $(a < b)$, i.e. 1 if $(a < b)$ and 0 if $(a \geq b)$. We use the comparison circuit of [53] that, for comparing two ℓ-bit integers needs only ℓ non-XOR gates, which is optimal. The ℓ-bit comparison circuit is a concatenation of ℓ one-bit comparators. A one-bit comparator takes as input three bits: one bit of a, one bit of b and a carry bit c. The output z of the one-bit comparator is defined as follows:

$$z = \begin{cases} c & a = b \\ 1 & a < b \\ 0 & a > b \end{cases}$$

The correct one-bit comparator circuit is implemented as $z = b \oplus ((a \oplus c) \wedge (b \oplus c))$. The ℓ-bit comparator is just the concatenation of ℓ one-bit comparators with the initial carry $c_0 = 0$. The invariant that holds in each step is that when the i-th one-bit comparator is processed, the carry $c_{i+1} = ((a_i, \ldots, a_0) < (b_i, \ldots, b_0))$. Per induction the final output $c_{\ell+1}$ is hence correct.

Comparison against a constant. The circuits we are really interested in in our scheme are not generic comparison circuits but in fact circuits that compare the input against a hardwired constant b. To achieve this, we could garble a generic comparison circuit and store the correct input labels for the bits of b along with it. But we do not want to store the additional ℓ ciphertexts for the b-labels as this would be a significant overhead. Instead we directly want to garble a circuit that already represents the desired functionality. In terms of the comparison circuit, the values of b are known in advance and thus the one-bit comparator reduces to a single AND gate if $b_i = 0$ and a single NOR gate, if $b = 1$.

Though garbled circuits incur significant performance overheads in general due to their size, this is not true in our setting. 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 PROOF PRELIMINARIES AND PROOFS

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 the overall security definition and proof sketch.

B.1 Branching Garbled Circuit Chains

Arx-Range 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 start by reminding the reader of the security definition of a garbling scheme. In most settings where garbled circuits are used, the circuit f is public. Circuit privacy can always be achieved through a universal circuit but this incurs a significant performance penalty. In Arx-Range, it is publicly known that the circuits are comparison circuits, but it should remain secret which constant those circuits compare against. To quantify the amount of information that can be leaked without compromising security, we define the XOR-topology of a circuit.

Definition B.1 (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.

Definition B.2 (Garbled Circuit Security).

- **GC Circuit Privacy (**gc-circuit-prv.sim$_\Phi$**):** Intuitively, the garbled circuit F should not reveal any more information than $\Phi(f)$. More 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 (**gc-eval-prv.sim$_\Phi$**):** Intuitively, the collection (F, X) should not reveal any more information about x than $f(x)$. More 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 B.3 (BGCC). A *branching garbled circuit chain* is a tuple of algorithms $\mathcal{G}$ = (Generate, Encode, 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, 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 a 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 $X_b = \mathsf{Encode}(e_{f(x)}, x)$.

Definition B.4 (BGCC Security).

- **BGCC Circuit Privacy (**bgcc-circuit-prv.sim$_\Phi$**):** Intuitively F should not reveal any more information than $\Phi(f)$. More 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 (**bgcc-eval-prv.sim$_\Phi$**):** Intuitively, the collection (F, X) should not reveal any more information about x than $(f(x), X_{f(x)})$. More 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.

B.1.1 BGCC Construction. Following the high-level description in §A.1.1 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 [87] for efficiency, which satisfies Def. B.2.

Construction B.5 (BGCC). Let $\mathcal{G}^*$ = (Garble*, Encode*, Eval*, Decode*) be the half-gate garbling scheme. Let f be a boolean circuit with n inputs and 1 output and O a random oracle.

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 the 1-label for the output wire in F.
(3) $I^0 \leftarrow \mathsf{Encode}^*(e, 0^n)$
(4) $I^1 \leftarrow \mathsf{Encode}^*(e, 1^n)$
(5) $O_0^0 \leftarrow \mathsf{Encode}^*(e_0, 0^n)$
(6) $O_0^1 \leftarrow \mathsf{Encode}^*(e_0, 1^n)$
(7) $O_1^0 \leftarrow \mathsf{Encode}^*(e_1, 0^n)$
(8) $O_1^1 \leftarrow \mathsf{Encode}^*(e_1, 1^n)$
(9) for $i = 1, \ldots, n$ do:
  (a) $b' = \mathrm{LSB}(I^0[i])$
  (b) $T_d^0[i] \leftarrow O(K_d \,\|\, I_{b'}[i]) \oplus O_d^{b'}[i] \quad \forall d \in \{0, 1\}$
  (c) $T_d^1[i] \leftarrow O(K_d \,\|\, I_{1-b'}[i]) \oplus O_d^{1-b'}[i] \quad \forall d \in \{0, 1\}$
(10) Let $F = (F^*, d^*, T_d^b[i]), e = e^*$ and output (F, e)

Encode(e, x) is the same as Encode$^*(e, x)$

Eval(F, X):
(1) Parse the input as $F = (F^*, d^*, T_d^b[i])$.
(2) $b \leftarrow \mathsf{Decode}^*(d^*, \mathsf{Eval}^*(F^*, X))$
(3) Let K_b, be the b-label of the output wire
(4) for $i = 1, \ldots, n$ do:
  (a) $b' = \mathrm{LSB}(X[i])$
  (b) $X_b[i] \leftarrow T_d^{b'}[i] \oplus O(K_b \,\|\, X[i])$
(5) output (b, X_b)

Theorem B.6. *The BGCC construction B.5 satisfies the syntax and correctness guarantees as stated in definition B.3.*

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

Theorem B.7. *The BGCC construction B.5 satisfies the BGCC security definitions* bgcc-circuit-prv.sim$_{\Phi_{\text{xor}}}$ *and* bgcc-eval-prv.sim$_{\Phi_{\text{xor}}}$.

Proof. As a first step, one needs to convince themselves of the fact that the underlying half-gate garbling scheme satisfies gc-circuit-prv.sim$_\Phi$ and gc-eval-prv.sim$_\Phi$ with leaking only the XOR-Topology.

Evaluation-privacy can be seen by looking at theorem 1 in [87]. The theorem unnecessarily includes the whole circuit topology in the leakage, but by looking at the proof one can see 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, which we omit in the interest of space.

The random oracle O is with probability $1 - \mathrm{negl}(\lambda)$ 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 easy:

Our simulator for bgcc-circuit-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 bgcc-eval-prv.sim$_{\Phi_{\text{xor}}}$ on input $(1^\lambda, \Phi(f), f(x), X_{f(x)})$ 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.2 Treap data structure

In Arx-Range 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. Therefore we use a *history-independent* treap [18] data structure to implement our index.

A treap is a probabilistic tree data structure with *expected* runtime guarantees, 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.

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

Definition B.8 (Shape of a Binary Search Tree). We define the *shape* of a binary search tree to be the underlying directed acyclic graph with edges being either marked as left or right, corresponding to the original tree.

Definition B.9 (Rank of an Insert). For a sequence of insert queries $Q = \{q_1, \ldots, q_n\}$ with q_i represented as a pair of a key and the number in the sequence, i.e. $q_i = (v_i, i)$, define the rank of a query as $rank(q_i) = |\{(v', i') \in Q | v' < v_i \vee (v' = v \wedge i' < i)\}|$, i.e. the number of queries in the sequence that either insert a smaller key or the same key and come before.

Fact B.10. *If a sequence of insert queries $Q = \{q_1, \ldots, q_n\}$ generates a treap with certain shape, then the sequence of inserts $Q' = \{(\text{rank}(q_1), 1), \ldots, (\text{rank}(q_n), n)\}$ generates a treap with the same shape.*

Fact B.11. *For a sequence of insert queries $Q = \{q_1, \ldots, q_n\}$ where each insert query is already assigned the random treap-priority, the order in which the queries are executed does not influence the shape of the final treap. The final treap will always have the same shape as the binary search tree that would be generated if the queries were executed in the order of their priority without rebalancing.*

B.3 Semantic security of the individual encryption schemes

We remind the reader of the standard semantic security definition of encryption schemes and then prove that our encryption schemes provide this guarantee. This will aid in proving the indistinguishability of the initial databases after the first phase in Def. 3.4.

Definition B.12 (The generic indistinguishability experiment under eavesdropping). We define the following experiment:

$\mathbf{Ind}^{\mathsf{Adv}}_{\mathsf{Gen},\mathsf{Enc}}(1^\lambda)$:

(1) $(m_0, m_1, \mathcal{A}) \leftarrow \mathsf{Adv}(1^\lambda)$, where m_0, m_1 are valid messages of the same length.
(2) $k \leftarrow \mathsf{Gen}(1^\lambda)$
(3) $b \leftarrow \{0,1\}$
(4) $c \leftarrow \mathsf{Enc}_k(m_b)$
(5) $b' \leftarrow \mathsf{Adv}(\mathcal{A}, c)$
(6) if $b = b'$, the output is 1, otherwise it is 0.

Definition B.13 (Indistinguishability of ciphertexts). An encryption scheme with the algorithms Gen, Enc has indistinguishability of ciphertexts if for every PPT adversary Adv there exists a negligible function negl such that

$$\Pr[\mathbf{Ind}^{\mathsf{Adv}}_{\mathsf{Gen},\mathsf{Enc}}(1^\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

BASE is implemented as AES in counter mode with a random initialization vector and Enc denotes encryption with this scheme.

Fact B.14. BASE *has indistinguishability of ciphertexts.*

EQunique is cryptographically a pseudorandom function.

Lemma B.15. EQunique *has indistinguishability of ciphertexts.*

Proof. We proof that EQunique is a pseudorandom function. Let's assume there exists an adversary $\mathcal{A}$ that has non-negligible advantage in distinguishing EQunique from a truly random function. Let $\{m_i\}_{1 \leq i \leq n}$ be the requests $\mathcal{A}$ makes to the oracle. We distinguish two cases. In the first case, let $h(m_i) \neq h(m_j), i \neq j$. Then $\mathcal{A}$ could also effectively distinguish AES from a random permutation. In the second case, let $h(m_i) = h(m_j)$ for some $i \neq j$, and $m_i \neq m_j$. Then $\mathcal{A}$ has non-negligible advantage in producing collisions for h. This completes the proof. □

EQ is implemented as $\mathsf{Enc}_k(m) = \mathrm{AES}_{\mathrm{AES}_k(\mathrm{m})}(r) \parallel r$ for a randomly chosen $r \leftarrow \{0,1\}^\lambda$ and AES a pseudorandom permutation.

Lemma B.16. EQ *has indistinguishability of ciphertexts.*

Proof. By the security of AES we have that $\mathsf{Enc}_k(m)$ is indistinguishable from $\mathrm{AES}_\$(r) \parallel r$ for a truly random \$, which is again indistinguishable from $\$ \parallel r$ which is truly random. □

AGG is the Paillier encryption scheme [70].

Lemma B.17. AGG *has indistinguishability of ciphertexts.*

Arx-Eq is implemented as $\mathsf{Enc}_k(x \parallel b, m) = h(\mathsf{Enc}_k(x, m) \parallel b)$ where x is a binary string and b a single bit and for the empty string $x = \epsilon$ we have $\mathsf{Enc}_k(\epsilon, m) = \mathsf{EQunique}_k(m)$ and h is a random oracle. An alternative implementation not based in the random oracle model is $\mathsf{Enc}_k(x \parallel b, m) = \mathrm{AES}_{\mathsf{Enc}_k(x,m)}(b)$ with AES a pseudorandom permutation.

To encrypt a list of plaintexts $L = (m_1, \ldots, m_n)$ with Arx-Eq, we have $c_i = \mathsf{Enc}_k(x, m_i)$ where x is the $\lceil \log n \rceil$-bit binary string representation of the number of occurrences of m_i in $\{m_0, \ldots, m_{i-1}\}$

Lemma B.18. *Arx-Eq has indistinguishability of ciphertexts.*

Proof. We prove the construction based on pseudorandom permutations. The construction based in the random oracle model is straight forward. Let $c = (c_1, \ldots, c_n)$ be a list of ciphertexts produced by Arx-Eq. We examine the list of hybrids H_i where $H_i = (c_1, \ldots, c_i, r_{i+1}, r_n)$ where r_j is a random string. We have that $H_n = c$ and H_0 is completely random. To see that $H_{i-1} \overset{c}{\approx} H_i$, recall that $c_i = \mathrm{AES}_{k_i}(b_i)$ for keys k_i, k_j constructed accordingly to the definition of Arx-Eq, and $c_j = \mathrm{AES}_{k_j}(b_j)$ and by construction if $k_i = k_j$ for $j < i$ then $b_i \neq b_j$. Hence c_i is indistinguishable from random. □

Arx-Range. We show indistinguishability of Arx-Range indices of the same size. Then we show the indistinguishability of queries to the index when they have the same leakage.

Lemma B.19. *Arx-Range has indistinguishability of indices.*

Proof. Let $\mathbb{I}_0$ respectively $\mathbb{I}_1$ be two Arx-Range indices of the same size n. We now prove $\mathbb{I}_0 \overset{c}{\approx} \mathbb{I}_1$.

First, the shape of both indices is perfectly indistinguishable through the treap properties. Specifically, the facts B.10 and B.11 say that a set of inserts $Q' = \{rank(q_1), \ldots, rank(q_n)\}$, which is exactly $\{1, \ldots, n\}$, generates the same shape as any other set of inserts of size n. Without loss of generality we therefore from now on assume the shapes of $\mathbb{I}_0$ and $\mathbb{I}_1$ were the same.

We proceed by showing that a node $v \in \mathbb{I}_0$ and the corresponding node $v' \in \mathbb{I}_1$ are computationally indistinguishable: $v \overset{c}{\approx} v'$. Each node contains a BGCC, an encryption of the key and an encryption of the payload (the value). The BGCCs are indistinguishable (Theorem B.7) and the latter two are indistinguishable through the properties of the encryption scheme. □

Lemma B.20. *Two queries on an Arx-Range index are indistinguishable (Def. 3.4) given that they have the same leakage as defined in Def. 3.3.*

Proof. Given that each pair of queries produces the same leakage, the outputs of the BGCCs in the execution are the same. It follows from Theorem B.7 that the queries are indistinguishable. The indistinguishability of the repair data follows with the same arguments as in the indistinguishability of the initial indices. □

B.4 Overall Security Proof Sketch

Sketch of proof of theorem 3.5. We want to show that the Arx database system is adaptively secure in the indistinguishability sense with leakage Leak as defined in Def. 3.1 and Def. 3.3 under standard cryptographic assumptions.

All encryption schemes and indices are used independent of each other, therefore we show the security of the individual schemes. Let $\mathsf{DB}^0, \mathsf{DB}^1$ be the databases after the first phase of the indistinguishability experiment. In §B.3 we already showed the semantic security of all encryption schemes we use individually. Given that $\mathsf{DB}^0, \mathsf{DB}^1$ are structurally the same they are therefore indistinguishable.

It remains to show that individual queries leak nothing beyond their specified leakage. We already showed this fact for Arx-Range in Lemma B.20. The proof for Arx-Eq is similar to the one of Sophos [25]. Aggregations and order-by queries do not leak anything on top because no secret key is used to amend queries with these operations and hence they can directly be simulated. We skip the proof for Arx-Join as it is just a composition of the other schemes. □