An Overview of Queryable Encryption

September 30, 2026

Contents

1 Introduction **3**

2 The Design Goals of Queryable Encryption **5**

3 An Overview of Queryable Encryption's Design **6**
3.1 How `QE` Handles Range Queries 10
3.2 How `QE` Handles String Queries 11

4 Reasoning about Database Encryption Security **13**
4.1 How we Evaluated Queryable Encryption 17

5 Adversarial Models **18**

6 Theoretical Analysis **19**
6.1 Range Queries in the Persistent Model 21
6.2 String Queries in the Persistent Model 23

7 The MongoDB Logging Architecture **23**

8 Insider Analysis **27**
8.1 Database User Roles 27
8.2 Database Administration Roles 28
8.3 Cluster Administration Roles 29
8.4 Backup and Restoration Roles 30
8.5 Custom Roles 31

9 General Guidelines **31**
9.1 Scheduling Compactions 31
9.2 Setting the Contention Factor 33
9.3 Handling Values with Sensitive Lengths 34
9.4 Using the Aggregation Pipeline 35
9.5 Additional Guidelines 35

10 Guidelines for Range Queries **36**
10.1 Reducing Storage for Bounded Domains . . . 36
10.2 Reducing Storage via the Sparsity Factor . . . 38
10.3 Reducing Contention via the Trimming Factor . . . 38
10.4 Correctness & Security Considerations . . . 39

11 Guidelines for String Queries **39**
11.1 The Tag Count of a Value . . . 40
11.2 Setting the Parameters . . . 42
11.3 Scheduling Compactions for String Indexes . . . 42
11.4 Additional Security Considerations . . . 44

12 The Research Program Behind Queryable Encryption **44**

References **48**

A Storage Estimates for Queryable Encryption **52**
A.1 Storage Cost per Collection . . . 52
A.1.1 Encrypted Data Collection . . . 52
A.1.2 Encrypted State Collection . . . 53
A.1.3 Encrypted Compaction Collection . . . 53
A.2 Encoding Overhead . . . 54
A.3 Total Storage . . . 54
A.3.1 Total Size of Documents . . . 54
A.3.2 Total Size of Indexes . . . 54
A.4 Calculating the Storage Overhead for a Schema . . . 54

1 Introduction

MongoDB is an open-source, highly-scalable, database management system (DBMS). In June 2022, MongoDB released the preview of `Queryable Encryption` (`QE`), a feature that provides developers the ability to encrypt selected fields in their databases client-side while retaining the ability to run expressive queries on the encrypted data. Since then, `QE` has become generally available in stages: equality queries in release `7.0`, range queries in release `8.0` and prefix, suffix and substring queries in release `9.0`. This document provides an overview of how `QE` works, its design goals, the security guarantees and tradeoffs it provides and guidelines on how to best configure and tune it. *This document is not a developer guide for* ***`QE`****. Documentation for* ***`QE`*** *can be found at [48].* Before describing how `QE` works, we recall the evolution of database encryption. This historical context is important to understand the design decisions made in building `QE` and where it sits within the set of database encryption technologies available to developers.

Database encryption technologies. The purpose of encryption in a DBMS is to minimize the occurrence of data breaches. According to IBM's Cost of Data Breaches report [53], the average cost of data breaches in 2022 was $4.35 million. Furthermore, 83% of the organizations surveyed had suffered more than one breach. Integrating encryption into a DBMS is a notoriously difficult problem; specifically, because the purpose of a database is to efficiently query and process data while the purpose of encryption is to make data unintelligible. Nevertheless, the importance of integrating encryption into database systems cannot be overstated since most data is stored and processed by database management systems.

Database encryption technology has evolved over time. We can roughly categorize most solutions within the three following groups:[1]

1. the *first generation* includes Db2's `Native Encryption` (`NE`) [29], SQL Server's `Transparent Data Encryption` (`TDE`) [42] and MongoDB's `Encrypted Storage Engine` (`ESE`).

   These work by encrypting database files using standard symmetric encryption schemes. Database file encryption provides encryption at rest and protects databases against breaches of disks and backups.

2. the *second generation* includes `Always Encrypted` for SQL Server (`AE`) and `Client-Side Field-Level Encryption` (`CSFLE`) for MongoDB as well as earlier research systems like `CryptDB` [4], `SEEED` [52] and `CipherBase` [10]. These systems work by encrypting database columns/fields using property-preserving encryption schemes like deterministic encryption [12] and order-revealing encryption [8, 14]. They provide encryption in use and protect databases against a larger but more nuanced set of data breaches.

3. the *third generation* includes `Queryable Encryption` for MongoDB and earlier research systems like `ESPADA` [17, 16], `Blind Seer` [50, 25] and `KafeDB` [35, 39, 58]. These systems

[1] We do not consider enclave-based databases like `Always Encrypted with secure enclaves` and earlier research systems like `TrustedDB` [11], `EnclaveDB` [51] and `StealthDB` [55] in this categorization; we discuss this class of systems separately below.

work by encrypting databases with structured encryption schemes. They provide encryption in use and protect databases against an even larger and more nuanced set of data breaches.

These three approaches are based on different underlying cryptographic primitives and provide different security and performance guarantees. One way to think about the evolution of database encryption is that, with each generation, we see an increase in the surface of the DBMS that benefits from encryption and a reduction in the data lost in the case of a breach.

TEE-based encrypted databases. A class of systems that we deliberately excluded from the categorization above encrypts data client-side but processes queries inside a trusted execution environment (TEE), which is a hardware-protected region of the server's processor and memory. Examples include `Always Encrypted with secure enclaves` for SQL Server and research systems like `TrustedDB` [11], `EnclaveDB` [51] and `StealthDB` [55]. We excluded them because their security rests on hardware isolation rather than on the structure of the encrypted data, so they cannot be placed among the three generations without a leakage analysis, and until recently no such analysis existed.

MongoDB's Cryptography Research Group recently conducted the first cryptanalysis of these systems [22, 20]. The attacks do not break the hardware: they assume the TEE works as intended and target what these systems write to untrusted storage. A database must persist its indexes to disk, so a TEE-based EDB writes encrypted index pages outside of the TEE, and the set of pages that an insertion modifies depends on the inserted data. Using this, the first work [22] shows that an adversary that only obtains snapshots of the disk can recover the order in which values were inserted, and the second [20] shows that several TEE-based EDBs that were claimed to reveal only this order also reveal the frequencies of the stored values, in some cases from a single snapshot. The second set of attacks was executed end-to-end against deployed systems, including `Always Encrypted with secure enclaves`. The takeaway is that hardware isolation does not remove leakage, and that TEE-based encrypted databases have to be evaluated with the same methodology as the cryptographic solutions described above.

It is important to keep a few things in mind when considering database encryption, however. The first is that *database encryption technology is not a silver bullet* and the second is that *database encryption is an evolving technology.*

Database encryption is not perfect. Database encryption solutions improve database security but they do not provide an absolute guarantee of security. Database encryption reduces the attack surface of the DBMS, but it cannot remove it completely. Nevertheless, when properly designed and deployed, database encryption, coupled with information security best practices like access control and auditing, can improve an organization's security and privacy posture.

Database encryption is evolving. While each generation of database encryption technology represents a significant technical improvement over the previous one, this technology should be examined and considered not only in the short term but with a long term outlook. In fact, MongoDB's investment and efforts in this technology are not limited to Queryable Encryption as it exists today but already include numerous R&D projects to develop the fourth and fifth

generation of database encryption. So, while QE is the state-of-the-art for what is available in commercial database platforms today, MongoDB is committed to keep improving and evolving this technology.

Should we be using imperfect cryptography? If database encryption is imperfect, it is natural to wonder if one should be using it at all. Ultimately, this is a question developers and organizations must make for themselves but the following may be helpful while contemplating this decision. Cryptography is not and has never been foolproof and should never be viewed as a silver bullet. Cryptography is only one of many security technologies and it should always be used as such. It is also important to understand that symmetric encryption schemes like AES and cryptographic hash functions like SHA-256 are believed to be secure but we do not know this with certainty. The reason these primitives are used in practice is because they were designed and analyzed using a rigorous and transparent process that the cryptography community has developed over years. This is also true for public-key encryption schemes.

2 The Design Goals of Queryable Encryption

QE is an alternative to MongoDB's CSFLE[2] that is designed to provide different functionality and guarantees. QE was designed with the following goals in mind:

1. *expressive queries*: CSFLE supports `find` operations with the equality `$eq` operator but cannot be extended to support other comparison operators like `$gt`, `$gte`, `$lt` or `$lte`. Note that this is a fundamental limitation of CSFLE. QE, on the other hand, was designed to be extendable to a large set of operators including range, prefix, suffix and substring operators.

2. *performance*: a CSFLE-encrypted database incurs almost no performance overhead compared to a plaintext database.

3. *non-interactive*: standard MongoDB operations are single-round and this should be preserved by both CSFLE and QE.

4. *stateless*: CSFLE and QE clients should be stateless so that they can be used by short-lived and lightweight clients (e.g., running in containers); they also should not require synchronization across clients.

5. *security*: CSFLE provides encryption in use with a specific security profile which we will discuss later. QE should address the same threat models as CSFLE but with a better security profile when possible.

[2]Throughout this document, when referring to CSFLE we mean CSFLE with deterministic encryption since CSFLE with randomized encryption is not queryable and, therefore, does not provide encryption in use.

Insider threats. To better understand the security goals of QE (and of CSFLE), it is important to consider how data breaches occur in the real world. According to the 2022 Cost of Data Breaches Report [53], 19% of breaches were primarily caused by stolen or compromised credentials, 16% by phishing, ≈ 15% by cloud misconfiguration, ≈ 12% by malicious insiders, ≈ 7% by system errors and ≈ 5% by accidental data loss or lost devices. While technologies like TDE and ESE protect data against lost or stolen devices, these vectors only accounted for ≈ 5% of breaches. The other attack vectors accounted for ≈ 69% of breaches. These statistics alone motivate the need to move beyond first-generation technologies like TDE and ESE to technologies that provide encryption in use like CSFLE, AE and QE. Taking a closer look at these attack vectors, however, one can see that they are not based on a complete compromise of the underlying system. In other words, 69% of the time, the attacker gained access to data without violating the security mechanisms of the underlying system (e.g., mechanisms like access control or process isolation).

This suggests that database encryption technologies that provide strong security guarantees even against attackers that cannot fully compromise the system are important. The fact that stolen credentials, phishing and malicious insiders already accounted for ≈ 47% of breaches suggests that solutions that provide strong security guarantees against insiders (malicious or not) are also of significant value. As we will see in Section 4, this is an important consideration in the design of QE.

3 An Overview of Queryable Encryption's Design

MongoDB is a document database management system that stores collections of JSON documents, each of which contains a variable number of field/value pairs. The architecture of a MongoDB cluster is illustrated in Figure 1 and consists of the following components:

1. mongod: the database server that stores the data and responds to queries. Multiple mongod instances can be combined to form a *replica set* which consists of a primary mongod and several secondary mongod instances which store copies of the data.

2. mongos: MongoDB is designed to scale horizontally by running on large clusters. This is achieved using sharding which splits databases among several mongod instances. In sharded clusters, applications communicate directly with mongos instances which route queries and write operations to shards.

3. mongosh: a shell which makes use of the Node.js driver and is used for administrative tasks.

4. drivers: a collection of language-specific libraries that applications can use to interact with a MongoDB deployment.

MongoDB deployments are flexible: shards and replica sets can be added, removed, or refined at any time depending on the application workload.

Queryable encryption. QE is implemented as a set of client-side libraries and server-side code that enable applications to encrypt sensitive fields in their documents so that they remain encrypted even while the server processes them. The QE libraries implement a novel database

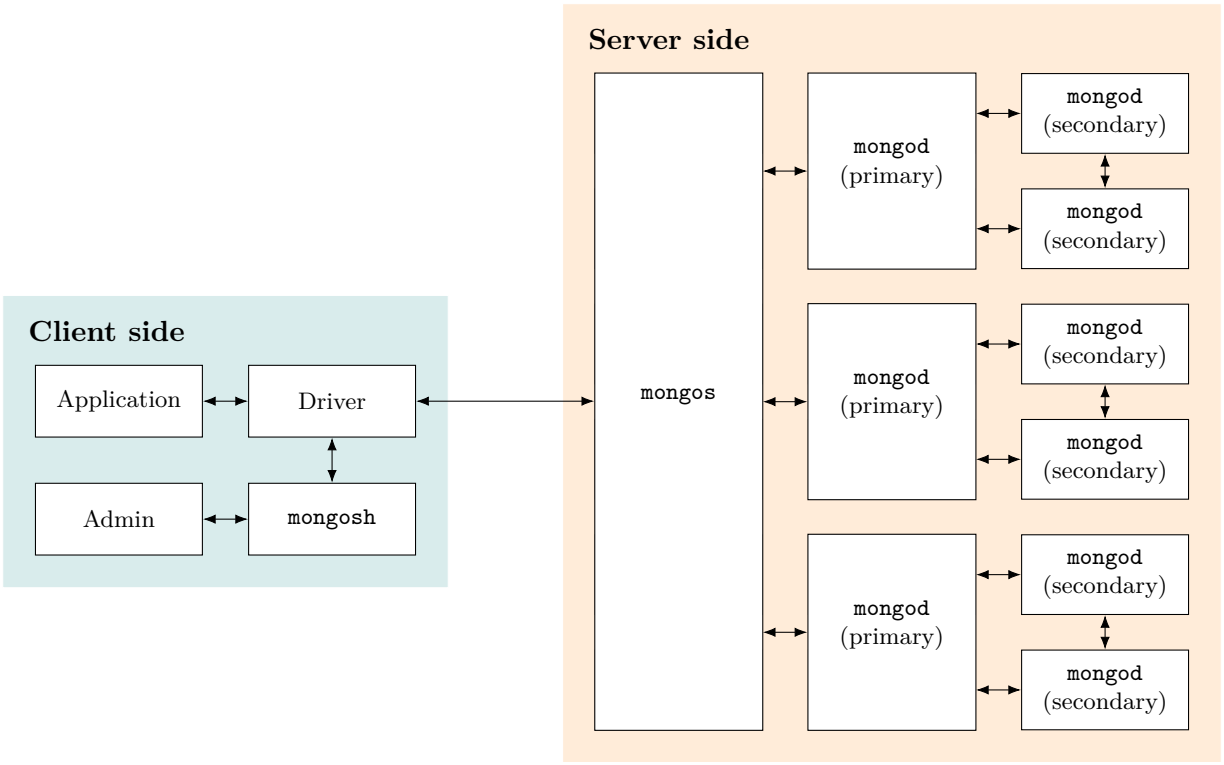


Figure 1: The architecture of a MongoDB cluster.

encryption scheme called OST which is described and analyzed in [37]. The components of QE include:

- the *drivers*: client-side language-specific libraries that applications use to interact with the database server. To use QE, applications load a driver to create and interact with an encrypted client.
- the *encrypted client*: applications make use of QE through encrypted clients that are responsible for, among other things, retrieving encrypted data encryption keys (DEK) from a key vault (see below), and translating plaintext insert, find, update and delete operations to their encrypted versions.
- the *KMS provider*: the encrypted client uses a customer master key (CMK) that is managed by a key management service (KMS) like AWS KMS, Azure Key Vault, Google Cloud Platform KMS or KMIP.
- the *key vault*: the encrypted client also makes use of a key vault collection maintained at the server that stores DEKs encrypted with the KMS-managed CMK.
- `libmongocrypt`: is a client-side library written in `C` used by encrypted clients for encryption, decryption and executing the client-side portions of encrypted insert, find, update and delete operations based on the OST database encryption scheme. Note that `libmongocrypt` does not introduce any I/O or network overhead and all the requests and interactions with the KMS are channeled through the driver.
- the *query analysis shared library*: is a client-side library that allows users to write insert, find, update and delete operations against encrypted data in plaintext and marks the necessary portions of those plaintext queries for encryption by `libmongocrypt` (such as which fields need to be encrypted and with which DEK). The query analysis functionality is only available in *MongoDB Enterprise Advanced* and *Atlas*.

When an application creates a QE-encrypted collection, it first creates an *encryption schema* that specifies which fields in the collection should be encrypted and what type of queries need to be supported. Currently, this can be either `none`, in which case field values are encrypted with standard encryption, or one of `equality`, `range`, `prefix`, `suffix` or `substring`, in which case OST is used. The string types `prefix`, `suffix` and `substring` apply only to fields that hold valid UTF-8 strings, and a field can be indexed for both prefix and suffix queries, whereas no other combination of query types on a single field is allowed. It then creates a QE-enabled client that can be set to use either *automatic encryption* or *explicit encryption*. QE-enabled clients with automatic encryption transparently handle encryption and decryption using the query analysis shared library. For example, if the application inserts a new document, the query analysis functionality uses the encryption schema to determine which fields need to be encrypted and marks those fields accordingly. `libmongocrypt` then uses those markings to encrypt and produce the appropriate cryptographic tokens needed for server-side query execution. On the other hand, if the client supports explicit encryption, the application is responsible for specifying each value's query type, DEK and other QE-specific parameters. Finds, updates and deletes work similarly.

QE's database encryption scheme. The cryptographic core of QE is called OST. OST is a structured encryption scheme—specifically, a document database encryption scheme—described and analyzed in [37]. QE implements an *emulated* version of OST which, roughly speaking, means that it implements the OST scheme but in a way that uses MongoDB's native data structures. Here, we only provide a high level overview of how QE works:

1. *create collection*: when creating a QE-encrypted collection, three collections are created: the *encrypted data collection* (edc) which stores the encrypted documents, the *encrypted state collection* (esc) which stores an auxiliary encrypted structure needed for search, and the *encrypted compaction collection* ecoc which stores an auxiliary encrypted structure needed to compact the esc if it gets too large. Note that the esc and the ecoc are standard MongoDB collections but their documents contain cryptographic data that, together, make up an "emulated" encrypted data structure. To highlight the fact that the documents in the esc and the ecoc are not regular documents—in the sense that they do not store user data—we sometimes refer to them as meta documents.

2. *document insertions*: a document with a set of fields $\mathbf{F}$ with a subset $\mathbf{EF} \subseteq \mathbf{F}$ of QE-encrypted fields is inserted as follows. For all fields $f \in \mathbf{EF}$, f's value v_f is encrypted using AES256-CBC-SHA256. In addition, an array __safeContent__ that stores cryptographic *search tags* is added to the document. These tags are generated using a pseudo-random function (PRF) and information stored in the esc.

3. *finds*: to find documents that match a given filter, a set of cryptographic *tokens* are sent to the server. The server uses these tokens with the esc to generate a set of cryptographic search tags and returns the documents whose __safeContent__ array stores any of these tags.

4. *updates*: to update a document that matches a given filter, a set of cryptographic tokens and an updated encrypted value are sent to the server. The server uses these tokens with the esc to generate a set of cryptographic search tags. One of the documents whose __safeContent__ contains one of these tags is then updated with the new encrypted value.

5. *deletes*: to delete a document that matches a filter, a set of cryptographic tokens and an updated encrypted value are sent to the server. The server uses these tokens with the esc to generate a set of cryptographic search tags. One of the documents whose __safeContent__ contains one of these tags is then deleted.[3]

6. *compacts*: the esc grows in size so QE supports a *compaction* operation to reduce its size. To compact the esc, a cryptographic compact token is sent to the server. The server uses this token with information from the ecoc to delete stale items in the esc.

For more details on how OST works, we refer the reader to [37]. In the rest of this document, we refer to a triple (edc, esc, ecoc) as an encrypted database edb.

[3]The specific document is chosen arbitrarily by the server.

3.1 How QE Handles Range Queries

In release 8.0, QE handles range queries over a variety of numerical data types. Here, we will give an overview of how ranges are handled but we highlight that ranges queries in QE have a different security/leakage profile than equality queries so they should be used with some degree of caution. We discuss this in more detail in Section 4, but the summary is that *range queries should not be used if one is concerned with persistent adversaries.*

Numerical data types. QE supports a variety of numerical types, including, sint32, int64 (date), sint64, bin64 (IEEE 754 binary64) and dec128 (IEEE 754 decimal128). We briefly describe below each of the supported numerical data types.

- The signed integer sint32 numerical type represents 32-bit integers that can take values from $-2,147,483,648$ to $2,147,483,647$.
- The unsigned integer int64 (date) numerical type represents 64-bit integers that can take values from 0 to $18,446,744,073,709,551,615$.
- The signed integer sint64 numerical type represents 64-bit integers that can take values from $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$.
- The IEEE 754 bin64 numerical type is a double-precision floating-point format that supports ≈ 15 decimal digits of significand. bin64 requires 64-bits and represents a large number of numerical values that can go up to $\approx 10^{308}$.
- The IEEE 754 dec128 numerical type is a decimal floating-point format that supports ≈ 34 decimal digits of significand. dec128 requires 128-bits and represents a large number of numerical values that can go up to $\approx 10^{6,144}$.

queries and some of the nuances and limitations of its approach.

A naive range-to-point transformation. Roughly speaking, QE handles range queries by transforming them to a set of equality queries which we sometimes refer to as *point* queries. To gain some intuition as to how this might work, consider a find query for a field f and integer range $[0, 3]$ and note that one could recover all the relevant documents by transforming it into point queries for 0, 1, 2 and 3. While this works and is relatively simple to implement, it has one major drawback: if the range is large then it will be transformed into a large number of point queries. More formally, if we define the width of a range query $[a, b]$ as $w = b - a$, then this simple transformation results in w point queries.

A better range-to-point transformation. A better approach is to decompose the *domain* $\mathbf{V}$ of the field into subsets $\mathbf{V}_1, \ldots, \mathbf{V}_m$ in such a way that a range query $[a, b]$ can be decomposed into a subset of $\mathbf{V}_1, \ldots, \mathbf{V}_m$. For example, suppose the domain of an integer field is $\{0, \ldots, 10\}$ and the range query is $[2, 6]$. If the domain was decomposed into, say, $\mathbf{V}_1 = \{0, 1, 2\}$, $\mathbf{V}_2 = \{3, 4, 5, 6\}$, $\mathbf{V}_3 = \{7, 8\}$ and $\mathbf{V}_4 = \{9, 10\}$ then the range query could be decomposed into point queries for $\mathbf{V}_1$ and $\mathbf{V}_2$ which is only 2 point queries. In fact, using this decomposition, a range query will be transformed into at most 4 point queries which is considerably better than the previous

approach which would require 10. The limitation of this transformation, however, is that it can result in false positives. For instance, in the example above, the point queries for $\mathbf{V}_1$ and $\mathbf{V}_2$ used for range $[2, 6]$ return all the documents with field values in the range $[2, 6]$ in addition to the documents with field values in range $[0, 1]$.

State of the art. This can be handled, however, by decomposing the domain in a more clever way as shown in [24, 19]. In these works, the authors decompose the domain based on the structure of a binary tree and propose efficient algorithms to decompose range queries into point queries. These techniques result in no false positives and a query expansion that is upper bounded by $\log N$. This was further generalized in [40] which shows that these decompositions can be generalized beyond binary trees to more general combinatorial objects called *hypergraphs*. This work provides several different hypergraph constructions together with efficient algorithms that can be used to achieve a variety of tradeoffs between false positives, leakage, storage and query amplification.

Hypergraphs. To handle ranges, `QE` builds on the approach of [40] but uses a new hypergraph family that was custom-designed for our purposes. The details of this hypergraph family can be found in [40] but here we will only mention that the hypergraphs are described by the following parameters:

- the *sparsity factor* sparsifies the hypergraph which in turn improves `QE`'s storage overhead as well as its insert and update amplification at the cost of increasing its find amplification.
- the *trimming factor* trims the hypergraph which in turn increases `QE`'s throughput under concurrent insert and update operations and decreases its storage amplification at the cost of increasing its find amplification.
- if the *lower bound* and *upper bound* on the field values stored in the collection are higher and lower, respectively, than the default bounds of the field's numerical type, then `QE` can construct a smaller hypergraph which results in better performance for all operations.

3.2 How `QE` Handles String Queries

In release `9.0`, `QE` handles prefix, suffix and substring queries over encrypted string fields. Here, we give an overview of how these queries are handled, but we highlight that string queries have a different security/leakage profile than equality queries so they should be used with some degree of caution. We discuss this in more detail in Section 4, but the summary is the same as for range queries: *string queries should not be used if one is concerned with persistent adversaries.*

String search queries. A prefix query for a string s returns the documents in which the queried field's value starts with s, and a suffix query returns the documents in which the value ends with s. For example, a prefix query for `Mac` on an encrypted name field returns the customers whose last name begins with `Mac`, whereas a suffix query for `1234` on an encrypted social security number field returns the records whose number ends with those four digits. A substring query for s returns the documents in which s occurs anywhere in the value, so a substring query for

`eng` on an encrypted job title field returns both `Engineer` and `Engineering Manager`. A field is indexed for these queries when its encryption schema specifies the `prefix`, `suffix` or `substring` query type and, in addition, every string field supports exact matches on its values.

A string-to-point transformation. Recall from Section 3.1 that `QE` handles a range query by decomposing it into a small set of point queries. String search queries are also reduced to point queries, but the decomposition is less sophisticated. When a document with a substring field is inserted, the client derives from the field's value all of its substrings with lengths between a minimum and a maximum length, and it adds search tags for each of these substrings in the same way it would for the value of an equality field. A find operation with a substring filter then sends the cryptographic tokens for the queried substring alone, and the server processes it exactly like an equality find. Prefix- and suffix fields work the same way except that the client only derives the value's prefixes or suffixes. Note the contrast with ranges: the range transformation expands a find operation into up to $\log N$ point queries, whereas the string transformation expands an insert operation into as many point insertions as the value has substrings within the length bounds, and find operations remain single point queries.

Query and string lengths. The number of substrings of a value grows quickly with the value's length and with the range of lengths that can be queried, and each of these substrings adds search tags to the encrypted database. To keep this cost bounded, each query type configured on a string field specifies a minimum and a maximum query length, set with the parameters `lb` and `ub`, and only substrings with lengths in this range are indexed and can be queried. Substring fields additionally specify a maximum string length `mlen`, and inserting a longer value into the field fails. For substring indexes, the minimum query length must be at least 2, the maximum query length at most 6 and the maximum string length at most 50 characters. Moreover, the number of substrings indexed for a single value, including the dummies described below, is limited to 84,000. We discuss how to set these parameters in Section 11.

Case and diacritic folding. String search queries can be made case insensitive, diacritic insensitive or both, with two per-field parameters `caseSensitive` and `diacriticSensitive`. When a field is case insensitive, the client folds the case of the value before deriving its substrings and folds the case of queried strings in the same way, so a substring query for `eng` also matches `ENG`. Diacritic insensitivity works the same way by stripping diacritical marks. Folding also determines what an exact match means on these fields: two values match exactly when their folded forms are equal.

Padding with dummies. The search tags of a string field are stored with the document, so the number of tags is visible to anyone that can inspect the encrypted collection, and this number would reveal the number of distinct substrings of the value, which is more about the value than `QE` is designed to leak. To avoid this, the client pads: it adds tags derived from *dummies*, which are strings that are intentionally made invalid UTF-8 so that no query can ever match them, until the number of tags reaches an upper bound that depends only on the length of the value's ciphertext and on the field's parameters. We call this bound the value's *tag count* and quantify it in Section 11. As a result, the tags stored for a string field reveal the length of its value only up

to the precision that the ciphertext itself already reveals. This is a guarantee about the stored collections; we return to it, and to what an adversary that observes insert operations in transit sees in addition, in Section 6.

4 Reasoning about Database Encryption Security

Why is database encryption hard to design? Database encryption and, specifically, in-use database encryption, is challenging to design. There are several reasons for this, but the main one is a fundamental insight of cryptographic design that is often missed: it is possible (and common) to design insecure cryptographic primitives/protocols from completely secure building blocks. In other words, the fact that a cryptographic primitive/protocol makes use of a secure building block does not necessarily imply that the protocol is secure. In the context of database encryption, this leads to the common mistake of thinking that a database encryption solution is secure simply because it makes use of a standard encryption scheme like AES. It is important to understand that standard AES encryption (e.g., using Counter Mode or CBC mode) is designed to do one thing and one thing only: to encrypt a message with a given key so that it can be decrypted with the same key. Specifically, it is not designed to support *queries* on the ciphertext. This means that any database encryption solution—even one that is completely based on AES—has to be considered and analyzed as a new cryptographic object that does not automatically inherit the security properties of the underlying encryption scheme.

Why is database encryption hard to analyze? Once a solution is designed, its security needs to be analyzed. The security of cryptographic primitives is hard to analyze in general and is particularly so for database encryption for the following reasons. Consider a standard symmetric encryption scheme like AES256-CBC. Most people's intuitive understanding of AES256-CBC's security is that it "protects the plaintext" in the sense that, if a message m is encrypted with AES256-CBC then an adversary cannot learn anything about the message m from an encryption ct of m. While this seems correct, it is not completely true because an adversary can learn an approximation of m's length from the ciphertext ct. This bit of information is leaked because AES256-CBC pads messages to the nearest multiple of 128 bits and outputs a ciphertext of that size. This detail is usually omitted from discussions about the security of encryption because it does not seem to matter in many situations. However, consider a situation where the adversary knows that you are encrypting either "MongoDB" or "Queryable Encryption". An example AES256-CBC encryption of "MongoDB" in `base64` is

`U2FsdGVkX18aLwGD3rvdkptOLFTVJOJ98O7xPigWCYw=`

whereas an example AES256-CBC encryption of "Queryable Encryption" is

`U2FsdGVkX194wyPXIu4lQnmhsWoaGrxEOEhOPjw+oAc+a3VGH1FyAv1t9h4kWpeX`.

Given either of these ciphertexts, the adversary can easily guess which message was encrypted simply by looking at its length. Database encryption solutions are much more complex than this but this simple example gets at the heart of why analyzing database encryption is so challenging.

What is leakage in cryptography? As seen in the simple example above, every encryption scheme leaks something about the plaintext. In the case of AES256-CBC, the leakage is an approximation of the message's length. In the case of the one-time pad—which is a "perfectly secure" encryption scheme—the leakage is the exact length of the message. There are many possible ways of designing database encryption solutions, but the important thing to understand is that they all leak something. Furthermore, leakage is not unique to database encryption—it exists in regular/standard encryption and other cryptographic and privacy primitives such as secure multi-party computation and differential privacy. The question is not whether a particular solution leaks something; rather, the important questions are: (1) what exactly does it leak?; (2) can this leakage be exploited or not? and (3) can it be suppressed (preferably efficiently)? For most solutions, the answer to these questions is difficult to ascertain because it depends on a variety of factors including what exactly the adversary has access to and what information it already knows about the data, e.g., in our previous example the adversary knew that the ciphertext was either an encryption of "MongoDB" or of "Queryable Encryption".

Tradeoffs. Database encryption solutions can be designed in a variety of ways using a multitude of different cryptographic techniques and primitives. One should not get fixated on which specific primitive is used as a building block because what ultimately matters is how secure the final solution is and how well it performs. When evaluating a database encryption solution it is important to understand the tradeoffs it provides between:

1. *performance*: how much overhead does it add to database operations? does it increase the size of the data? Keep in mind that the performance of an encrypted database may be very different than the performance of a plaintext database even using the same DBMS. In other words, using a DBMS' s in-use encryption solution may radically change the workloads for which the system can be used. Because of this, standard benchmarks do not necessarily capture realistic workloads and thus the real-world performance profile of a solution.

2. *functionality*: what fraction of the query language does the solution support? Most in-use database encryption solutions only support a fraction of the system's query language and are only usable for certain applications.

3. *leakage*: what exactly does the solution leak and against what type of adversary? Under what conditions is that leakage exploitable?

Methodologies. Due to the complexity of the cryptographic designs and the subtleties involved in analyzing their security, database encryption solutions are usually analyzed using one of two methodologies:

1. *heuristic*: the solution is carefully designed for a given real-world commercial DBMS and to address a set of threats that are implicitly understood by the designer. The solution is then vetted by stakeholders and adopted if they feel comfortable with its design.

2. *theoretical*: the solution is carefully designed for an abstracted and idealized DBMS to address a set of threats that are explicitly and formally captured. The solution is then formally analyzed with respect to the adversarial model by formalizing the leakage profile

of the solution, then providing a proof that shows that the solution leaks at most what is captured by the leakage profile.

These two approaches each have advantages and disadvantages. The theoretical approach provides clear, explicit and strong security guarantees that can be verified, but it does this in the context of an idealized and simple DBMS which does not capture the complexities of commercial systems. The heuristic approach provides vague security guarantees that are harder to verify, but it does so in the context of a real DBMS without making any assumptions about how the system works. Unsurprisingly, all previous commercial database encryption technologies—including `CSFLE` and `Always Encrypted` for `SQL Server`—are designed and analyzed using the heuristic methodology, and most of the database encryption solutions proposed in the research literature are designed and analyzed using the theoretical methodology. `QE`, on the other hand, was designed and analyzed using a new, hybrid approach. Again, we stress that database encryption is a quickly evolving field and that, while our hybrid methodology is a step forward, it is not perfect and will continue evolving.

Evaluating leakage profiles via cryptanalysis. The theoretical methodology of proving that a solution achieves a given leakage profile has value, but only up to a certain point. In particular, it does not tell us whether or not a particular leakage profile is exploitable or even whether it is better or worse than another. So, how does one evaluate a leakage profile? There is not a really satisfactory answer to this question at this stage in the research literature. Establishing a formal and sound way to do this is one of the major problems still left open in the field. Currently, the best approach we have is to use cryptanalysis or, in other words, to design *leakage attacks* that try to exploit various leakage profiles. If a leakage profile withstands scrutiny, we can have some confidence that it might be safe. Notice that this is how most (if not all) practical real-world cryptography works. Nevertheless, there are some interesting research ideas on how to evaluate leakage profiles mathematically and without relying purely on cryptanalysis [56, 31, 30, 41, 36, 21]. These ideas are still in their infancy but are promising and likely to contribute to the evolution of database encryption technology.

Interpreting cryptanalytic results. Without wanting to downplay the importance and usefulness of cryptanalytic results, it is also important to learn how to interpret them correctly. This is difficult and subtle in the case of database encryption due to several reasons:

1. *emerging*: leakage attacks are a relatively new form of cryptanalysis so they are still poorly understood.

2. *evaluation*: there is no consensus on how leakage attacks should be evaluated empirically. Different papers use different methodologies and, more often than not, the evaluation criteria are chosen arbitrarily and without justification.

3. *assumptions*: leakage attacks rely on assumptions about the data and/or queries and what the adversary already knows about them. The reasonableness of these assumptions varies and experts may disagree on whether a particular assumption is reasonable in practice or not. The issue, however, is that often these assumptions are not properly and clearly stated or studied. And when assumptions are stated, they are rarely communicated as part of the work's high-level claims.

Again, we stress that our goal here is not to undermine the study of leakage attacks or to claim that these attacks should be dismissed. Indeed, this is an important area of research for MongoDB which is illustrated by the fact that MongoDB researchers contributed the following to the study of leakage attacks:

1. the first inference attacks against PPE-encrypted databases [49];
2. the state-of-the-art known-data and injection attacks against the volume pattern [13];
3. the state-of-the-art inference attacks against the query equality pattern [32];
4. an open-source software framework called `LEAKER` that implements the major leakage attacks and can be used to evaluate and analyze them on real-world data [33];
5. the first cryptanalysis of TEE-based encrypted databases, with attacks that recover the order of inserted values from snapshots of disk [22] and the frequencies of stored values from leakage profiles that were claimed to reveal only their order [20];
6. a statistical framework called Bayesian Leakage Analysis (BLA) that bounds what an adversary can infer from a leakage profile given its auxiliary information [21].
7. an algebraic framework called PolySys that automatically generates leakage attacks [23].

The takeaway is that the real-world impact of a leakage attack needs to be evaluated carefully. The cryptanalytic results it presents need to be put in context by asking the following questions:

1. what adversarial model does the attack work in and is that model relevant to my deployment scenario and threat model?
2. is the leakage profile exploited by the attack relevant to the database encryption technology I am using?
3. what are the limitations of the attack and are they properly communicated and disclosed?
4. what assumptions does the attack rely on and are they realistic?
5. how was the attack evaluated empirically and was the evaluation methodology meaningful?
6. are there any trivial countermeasures?

The answer to these questions can sometimes be hard to ascertain so it is always prudent to work with an expert in the area to gain perspective. This is no different than the need to evaluate and determine the practical relevance of published attacks against symmetric encryption schemes like `AES`, cryptographic hash functions like `SHA256` and public-key encryption schemes like `RSA`.

Capturing insider threats. The research literature on encrypted databases considers only two threat/adversarial models:

1. the *snapshot model* which captures disclosures of the encrypted database itself (but not, for example, any logs or data structures managed by the DBMS);

2. the *persistent model* which captures cases where the database server is completely and entirely compromised.

While these models are interesting, we found that they do not precisely capture the majority of threats that commercial database users are concerned with in practice. As discussed in Section 2, according to the 2022 Cost of Data Breaches Report [53], *insider threats* account for $\approx 47\%$ of breaches and at least $\approx 69\%$ of breaches did not require any compromise of the underlying system's security mechanisms. These statistics motivate more nuanced adversarial models and security analyses which we propose and describe in detail in Sections 5 and 8.

4.1 How we Evaluated Queryable Encryption

As we have seen, the design and implementation of database encryption is challenging. To evaluate `QE`, we adopted the following methodology:

1. *design and theoretical analysis*: we designed and analyzed `QE`'s core cryptographic scheme, OST, using the provable/reductionist security paradigm[4] to show that OST achieves a well-defined leakage profile with respect to a well-defined adversarial model: namely, security against multi-snapshot adversaries which we will define more precisely below. However, the analysis and proof of OST was done in an idealized setting where the adversary only has access to the encrypted structures. One way to think of this idealized setting is as if the analysis was carried out with respect to an idealized DBMS that does nothing else but store the encrypted database and run `QE`'s encrypted query algorithms. The results of this analysis are described in Section 6. The limitation of this analysis is that a real DMBS does more than just store the (encrypted) database and execute the query algorithms. In particular, real DMBSs log information about query executions, and this extra information can potentially alter the leakage profile achieved in the theoretical (multi-snapshot) security analysis.

2. *independent/external design review*: the theoretical design and security analysis of OST was reviewed by an independent cryptographer with considerable research experience in encrypted search.

3. *insider analysis*: as mentioned above, an important design goal of `CSFLE` and `QE` is protection against insider threats (either from stolen credentials, malicious insiders, misconfigurations, phishing or system errors). In these settings, `QE` needs to protect the database against an adversary that accesses the database through legitimate means and, in particular, using commands that are authorized for its role. In this step, we conducted an *insider analysis* in which we considered a set of important MongoDB built-in roles as follows:

[4]The provable/reductionist security paradigm is a methodology used to analyze the security of cryptographic primitives and protocols where one shows that the security of the primitive/protocol reduces to a standard computational assumption.

(a) *logging architecture and redaction*: we surveyed MongoDB's entire architecture to identify all the components that log information about `QE`-encrypted collections and, in some cases, ensured that `QE`-related information was redacted from the logs. MongoDB's logging architecture is extensive and complex, and some of its logs are necessary (e.g, for customer support). Nevertheless, when we could not redact information completely, we reduced it substantially, and, when we could not reduce it, we accounted for it in our insider analysis described in Section 8.

(b) *roles and commands*: for each role we considered, we surveyed its authorized commands using both MongoDB's documentation and code.

(c) *role-based leakage profile*: based on our survey of the roles' authorized commands and of MongoDB's logging architecture, we established which components a particular role has access to and, therefore, the leakage profile `QE` has with respect to this role. The results of this analysis are described in Section 8.

4. *internal code review*: after `QE` was implemented by the engineering team, the entire codebase was reviewed by the cryptography team to find gaps between implementation and design.

5. *independent/external code audit*: after the internal code review, the entire `QE` codebase was audited by an independent security firm.

Remark. While our approach to evaluating `QE`'s design and implementation is already extensive, we are continuously working to extend and improve how we evaluate `QE`. Section 12 describes the research program this evaluation draws on.

5 Adversarial Models

The security of an encrypted database can be analyzed using a variety of adversarial models which can be categorized along two dimensions: (1) the adversary's view, e.g., the database, the disk, certain logs, the memory or even the entire server; and (2) a schedule that determines when and for how long it has access to this view, e.g., a snapshot in time, multiple snapshots in time or continuously.

Existing models. The adversarial models considered in the literature include persistent adversaries and snapshot adversaries. Roughly speaking, persistent adversaries have access to the entire database server, whereas snapshot adversaries only receive the encrypted database. These two models have been useful in research, but, as first pointed out by [28], they do not necessarily capture real-world implementations of encrypted database systems. Indeed, DBMSs are very complex systems that satisfy many important requirements that are sometimes in conflict with security. To address this and to better capture the security guarantees of real-world encrypted database systems, we extend and increase the number of adversarial models for encrypted databases. In the following, we briefly and informally describe these new models.

Adversarial views. To analyze the security of real-world encrypted databases, we will consider a variety of possible adversarial models, each of which captures different kinds of attacks. The view of an adversary will be described along two dimensions as follows:

1. *surface*: the surface of a view is the subset of DBMS components the adversary can compromise. In the worst case, this could be the entire DBMS but in others it might include a subset of components like the network interface, plaintext databases, indexes, encrypted databases, various data structures, logs, files and disk.

2. *schedule*: the schedule of a view is the set of times at which the adversary can access the elements of its surface. One can consider a variety of different schedules, for example, after every operation, or only after a specific operation, or according to some probability distribution.

Definition 5.1. *Let* $\mathsf{dbms} = \{\mathsf{component}_1, \ldots, \mathsf{component}_m\}$ *be a database management system and* $\mathbf{op} = (\mathsf{op}_1, \ldots, \mathsf{op}_n)$ *be a sequence of operations. We say that an adversary* $\mathcal{A}$ *has a* $(\mathsf{surface}, \mathsf{schedule})$*-view if it can observe every component in* $\mathsf{surface} \subseteq \mathsf{dbms}$ *at each time in* $\mathsf{schedule}$*. We will sometimes say that* $\mathcal{A}$ *is a* $(\mathsf{surface}, \mathsf{schedule})$*-adversary.*

Using this framework, we can describe various kinds of snapshot adversaries as well as persistent adversaries. For example, traditional single-snapshot adversaries are (edb, t)-adversaries for $t \in [n]$, where $[n]$ denotes the set of integers $\{1, \ldots, n\}$. Traditional multi-snapshot adversaries are $(\mathsf{edb}, [n])$-adversaries and persistent adversaries are (dbms, ∞)-adversaries, where dbms refers to the entire system (including the network interface) and ∞ is the *continuous schedule* which allows the adversary to access the compromised components at any time and for as long as it wants.

6 Theoretical Analysis

In this section, we provide an overview of the theoretical analysis we conducted on `QE`. Before doing so, we also provide an overview of such an analysis for `NE`, `TDE`, `ESE` and `CSFLE` which, as far as we know, have never been formally analyzed.

Analysis of first-generation solutions. Solutions like `NE`, `TDE` and `ESE` encrypt the database files at the storage engine level with an externally managed key. The purpose of these solutions is to protect against disk theft or other forms of disk-level compromise. A detailed treatment of the security of `TDE` and `ESE` is beyond the scope of this document, but one can see that their security is actually more subtle when we consider different adversary models. For example, `TDE` and `ESE` reveal *at least* the size of the database against (disk, t)-adversaries (this depends on the specific design) and leak more to (disk, S)-adversaries for any set $S \subseteq [n]$ such that $|S| \geq 2$. In other words, `NE`, `TDE` and `ESE` leak something to an adversary that can access at least two snapshots of disk. This is because, whenever a change is made to the database, `TDE`/`ESE` only re-encrypts the pages that were affected by the change. If an adversary gets disk snapshots before and after the change, it can learn the number of pages that were affected by the update by looking for the pages that changed. In turn, the number of changed pages could be correlated

with, for example, the number of documents/rows that match an update's filter. In practice, this leakage may not be an issue, but this highlights that these solutions leak information even to an adversary that can only access disk. The takeaway is that every solution leaks something, and even standard and well-understood database encryption technologies exhibit subtle behaviors upon closer inspection that can depend on the adversarial model in question.

Analysis of second-generation solutions. Second generation solutions like `AE`, `CSFLE` and earlier research systems like [4, 52, 10] support queries on encrypted data by encrypting values with deterministic encryption. Surprisingly, the use of deterministic encryption as a database encryption solution has never been formally analyzed until now. At a minimum, these solutions have the following leakage profile against $(\mathsf{edb}, [n])$-adversaries:

1. `init` operations do not reveal anything beyond public parameters;
2. insert operations reveal the *shape* of the inserted document, by which we mean an upper and lower bound on the length of the values of each encrypted field.[5] They also leak the *field-level value equality*, i.e., whether two encrypted values for the same field across different documents are the same. In particular, the field-level value equality itself reveals the *frequency* of the field/value pairs in a collection.
3. `find` operations reveal nothing;
4. `deleteOne` operations reveal when the deleted document was inserted or updated in the past;
5. `updateOne` operations reveal the shape of the update and when the updated document was inserted or updated in the past. In addition, it reveals the field-level value equality of the updated field/value pairs which, in turn, reveals their frequency.

Note that both `CSFLE` and `AE` support more than these operations but, here, we only consider this subset because this is what `QE` supports.

Analysis of queryable encryption. Recall that the weakest kind of adversaries are (edb, t)-adversaries, which only get a single snapshot of the encrypted database. A stronger class of adversaries are $(\mathsf{edb}, [n])$-adversaries which receive multiple snapshots of the encrypted database, specifically after every operation. As we saw above, due to its use of deterministic encryption, `CSFLE` leaks the frequency pattern to (edb, t)-adversaries. On the other hand, `QE`'s underlying structured encryption scheme, OST, does not. In fact, it achieves a much better leakage profile against both single and multi-snapshot adversaries. A theoretical analysis of OST is available in [37], and OST has the following leakage profile against $(\mathsf{edb}, [n])$-adversaries (an updated analysis that also covers insertMany and deleteMany operations is forthcoming):

1. `init` operations do not reveal anything beyond public parameters;
2. insert operations reveal the shape of the inserted document;

[5] The length is revealed because AES-CBC encryption is used. This leakage can be avoided if every field value is padded to be the same length before passing them to `CSFLE` or `AE`.

3. `find` operations reveal nothing;

4. `deleteOne` operations reveal when the deleted document was inserted or updated in the past;

5. `updateOne` operations reveal the shape of the update and when the updated document was inserted or updated in the past;

6. `compact` operations reveal the number of unique values for all fields that were inserted since the last compaction.

7. insertMany operations, like insert operations, reveal the shape of the inserted documents.

8. deleteMany operations reveal when each deleted document was inserted or updated in the past and, if the filter is over an encrypted field, which of the deleted documents share that field's value. For example, if the filter selects the documents whose encrypted `name` field is equal to `Bob`, the adversary learns which documents have the same `name` and therefore how many there are, but not that the value is `Bob`.

Since deleteMany operations whose filter is over a plaintext field reveal nothing about the encrypted data, we recommend restricting deleteMany filters to plaintext fields whenever possible.

Comparison to second-generation solutions. A summary of the leakage profiles of `TDE`, `ESE`, `CSFLE`, `AE` and `QE` against single- and multi-snapshot adversaries over different surfaces is provided in Table 1. The above analysis is against $(\mathsf{edb}, [n])$-adversaries; that is, adversaries that can obtain a copy of the encrypted database after every operation. Against (edb, t)-adversaries, however, `QE` only reveals the shape of the documents in the collection. This is in contrast to second-generation solutions based on deterministic encryption like `CSFLE` and `AE` which reveal the shape of the documents and the frequency of each encrypted value in the collection/table.

Reminder. It is important to keep in mind that the analysis we summarized here (and can be found in detail in [37]) is *theoretical* in the sense that the surface only includes edb which, in the case of `QE`, are the user collection and all the auxiliary (encrypted) collections `QE` creates in order to support search. In other words, `QE` has the leakage profile described above *only* against adversaries that recover the encrypted database; i.e., the user's `QE`-encrypted collection together with `QE`'s auxiliary encrypted collections. If the surface changes, `QE`'s leakage profile will change as well. We discuss and analyze this in more detail in Sections 7 and 8.

6.1 Range Queries in the Persistent Model

As discussed in Section 3.1, `QE` handles range queries by transforming them to point queries but this does not necessarily mean that the security of range queries is the same as the security of equality queries. Concretely speaking—and in line with `QE`'s design goals—range queries provide similar security as equality queries against multi-snapshot adversaries (which, roughly speaking,

[6]This leakage profile is the result of a preliminary analysis. A formal and complete treatment is underway and when finished this document will be updated with the final results.

	Surface	Schedule	(Informal) Leakage
NE & TDE & ESE	disk	t	at least dbsize
	db	t	at least database
	dbms	t	database, queries, updates [6]
CSFLE & AE	edb	t	shape, valeq_f
	edb	$[n]$	shape, upshape, writetime, valeq_f
	dbms	∞	opeq, shape, upshape, writetime, valeq_f [6]
QE	edb	t	shape
	edb	$[n]$	shape, upshape, writetime, compuniq, delvaleq_f
	dbms	∞	opeq, shape, upshape, writetime, comptime, valeq_f [6]

Table 1: Leakage profiles of first-, second- and third-generation database encryption technologies against a variety of adversaries. The leakage pattern dbsize reveals the size of the database, database reveals the entire database, shape reveals upper and lower bounds on the values' lengths, the field-level value equality valeq_f reveals whether two field/value pairs are equal; the update shape upshape reveals upper and lower bounds on the updated values of a an updated document; the document operations writetime reveals when a document is inserted, updated or deleted; the operation equality opeq reveals if and when two operations are executed on the same parameters (e.g., field/value pairs, filters etc.); the post-compaction uniqueness compuniq reveals the number of unique values inserted since the last compaction, comptime reveals when a compaction occurs; and the deleted value equality delvaleq_f reveals, when a deleteMany filter is over an encrypted field f, which of the deleted documents share the same value in f. We stress that these descriptions are not completely formal and precise and are only meant to convey high-level intuition. For a formal and precise description of OST's leakage, see [37]; an updated analysis that also covers insertMany and deleteMany is forthcoming.

correspond to data exfiltration attacks) but are *considerably* weaker than equality queries against persistent adversaries which, roughly speaking, correspond to advanced persistent threats.

Recommendation. Even though QE is not designed for persistent adversaries, for completeness, MongoDB's Cryptography Research team conducted an extensive cryptanalytic study of QE's range queries against persistent adversaries by designing and executing leakage attacks against the leakage profile of QE range queries under various conditions (e.g., different target and auxiliary distributions). The attacks were designed with PolySys [23], an algebraic framework we developed that automatically generates leakage attacks from a description of a leakage profile (see also Section 12). The technical details of the attacks are complex and beyond the scope of this document but, in our exploratory experiments, the attacks achieved very high recovery rates—in some settings recovering up to 98% of the data. In our view, there is not yet a sound methodology for evaluating leakage attacks (see Section 4), so these results should not be read as precise measurements of risk. However, since we take a worst-case approach to security, the fact that such attacks succeed at all is sufficient for us to adopt the guideline that ***QE*** *should only be used against multi-snapshot adversaries and this is especially true for range queries. More concretely, this means that* ***QE****—and especially range queries—should only be used when one is concerned with data exfiltration, i.e., breaches that result in data being exposed but not*

both data and query transcripts/logs. In other words, if there is any chance that query transcripts/logs could be revealed to an adversary in addition to the (encrypted) data then `QE` *and especially range queries should not be used. This is true even if only a small number of query transcripts/logs might be revealed.*

6.2 String Queries in the Persistent Model

As with range queries, the fact that string queries are reduced to point queries does not mean that they inherit the security of equality queries. Against multi-snapshot adversaries, string fields achieve a profile similar to that of equality fields, as summarized above. Against persistent adversaries, however, the transcript of operations reveals more: the tokens of two find operations reveal whether they queried the same substring, and the response to each find reveals which documents matched it. This is a finer-grained signal than the corresponding leakage of equality queries because the substrings of a value are correlated with each other and with the value itself. For example, the substrings of `Engineer` overlap heavily with those of `Engineering`, so the leakage of queries on a substring field gradually accumulates information that relates the field's values to one another.

Insert operations also reveal more in this model than the stored collections do: within a single insert operation, the tokens sent for the padding entries are identical to one another, so an adversary that observes the operation can count them and learn the number of distinct substrings that the inserted value actually has. For prefix and suffix fields this determines the folded value's exact length whenever that length is at most the maximum query length. None of this is visible in the stored collections, where padding entries are indistinguishable from real ones, so the padding guarantee of Section 3.2 holds for snapshot adversaries but not for adversaries that observe insert operations.

Recommendation. Given the leakage described above, we recommend that *string search queries only be used when one is concerned with data exfiltration, meaning breaches that expose the (encrypted) data but not query transcripts/logs. If there is any chance that query transcripts/logs could be revealed to an adversary in addition to the encrypted data, then string queries should not be used. This is true even if only a small number of query transcripts/logs might be revealed.*

7 The MongoDB Logging Architecture

In this section, we provide an overview of MongoDB's logging and caching architecture. This is important for the following reason. Our theoretical analysis of `QE`'s underlying database encryption scheme, `OST`, is against a database-level snapshot adversary; that is, against an adversary that has access to the encrypted database. In the context of `QE`, this means access to $\mathsf{edb} = (\mathsf{edc}, \mathsf{esc}, \mathsf{ecoc})$. DBMSs, however, log all sorts of information about queries so, as pointed out in [28], an adversary that can access these logs sees more than `edb`. To account for this, we survey MongoDB's logging architecture and make use of it in our insider analysis in Section 8.

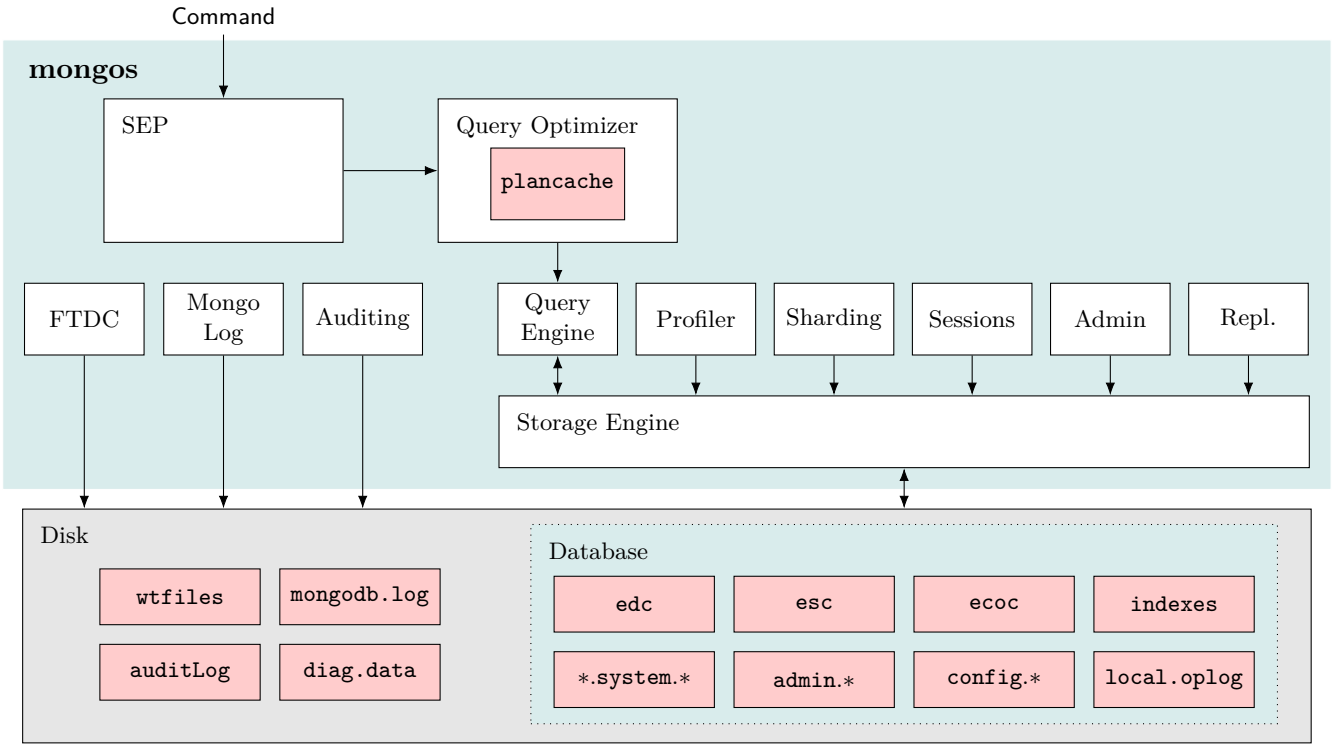


Figure 2: The `mongod` logging architecture.

The `mongod` architecture. `mongod` is composed of many components and logs information in a variety of internal databases, collections and files. For our purposes, however, the most important ones are illustrated in Figure 2 and summarized here:

1. *service entry point (SEP)*: when a command arrives at a `mongod` instance, it is processed by the service entry point which, among other things, verifies whether the user is authorized to run this command. If the command is a CRUD operation and the user is authorized the command is passed to the query optimizer.

2. *query optimizer and engine*: the query optimizer takes a query and finds an optimized query plan so that the query can be executed efficiently. In the process, the query optimizer may cache the plan it finds it its `plancache`. Once a plan is found, it passes it to the query engine to execute. The query engine executes the plan in conjunction with the storage engine.

3. *The (structured) Log*: `mongod` instances log various events to either the operating system's `syslog`, `stdout` or to a log file which is set to `/var/log/mongodb/mongodb.log` by default on Unix systems. `mongodb.log` captures a variety of messages such as those related to access control, database operations, network activities, replica set elections, diagnostic data, operations on indexes, storage engine journal, query planning, recovery (see [47] for a complete list).

4. *database profiler*: the database profiler logs database operations, configuration and admin commands to a capped collection called ∗.`system.profile` for any profiled database. The profiler is off by default and needs to be explicitly turned for a given database. It has several verbosity levels that dictate how much detail is logged (see [46] for more details).

5. *auditing*: MongoDB Enterprise includes an auditing service that logs system activity like authentication events, collection creations, renames and drops, index creations, role creation, granting and revocations, and system shutdowns and startups. The auditing service can also log database operations like `insert`, `find`, `findAndModify` etc. The auditing service is turned off by default and must be explicitly turned on. The audit log can be output to the `console`, to the `syslog` or to a file, e.g., `data/db/auditLog.json`.

6. *replication*: to provide redundancy and availability, every `mongod` instance is replicated across other `mongod` instances in MongoDB Atlas (and optionally in Enterprise Advanced and Community). The main instance is called the primary and the replicas are referred to as secondaries. The primary and secondaries form a replica set. To support replication, every `mongod` creates a database called `local` which stores information needed to synchronize and coordinate the replicas. One of the collections, `local.oplog.rs` stores every write operation made to the database. We refer to this collection as `local.oplog` and it is the primary mechanism by which secondaries synchronize their copy of the database to the primary's. While `local.oplog` stores write operations on `QE`-encrypted fields, it is a capped collection which means that it has a fixed size and gets overwritten if it grows larger.

7. *config*: MongoDB also logs information in a database called `config`.∗ in order to support sharding and causally consistent sessions. The config database includes a variety of collections (see [45] for more details).

8. *full time diagnostic data collection (FTDC)*: to enable support, `mongod` runs a service called FTDC that collects statistics about CPU, memory and disk utilization, network performance and CRUD operation performance. FTDC statistics are stored in a directory called `diagnostic.data` on the file system. By default, FTDC collects information every second and the `diagnostic.data` directory has a 200MB limit before it is overwritten. FTDC can be disabled on both `mongos` and `mongod`.

9. *admin*: MongoDB instances include an `admin`.∗ database which stores user authentication and authorization data such as usernames and passwords. It also contains information about the roles and privileges assigned to each user.

10. *other database-specific system collections*: MongoDB also maintains various internal database-specific `system` collections including ∗.`system.js` which stores client-defined JavaScript code for server-side computation, and the ∗.`system.views` which contains information about the database's views.

11. *WiredTiger*: the WiredTiger storage engine stores all the collections described above in disk and also generates additional files including the journal, `diagnostic.data`, `WiredTigerHS.wt`, `WiredTigerLAS.wt` and various other files for configuration purposes.

Commands. We summarize some of the commands that can be run to manage MongoDB databases and clusters. This is not a comprehensive list and is focused on commands that will be relevant to the insider analysis we carry out in Section 8.[7]

1. `$collStats`: returns statistics about a given collection;
2. `$planCacheStats`: returns statistics about the query plan;
3. `trafficRecord`: records (decrypted) network communication.
4. `auditConfigure`: configures auditing for `mongod` and `mongos` instances at runtime;
5. `listIndexes`: returns information about the indexes of a given collection including, the keys and options used to create the index;
6. `$renameCollectionSameDB`: allows the user to change the name of an existing collection on the current database;
7. `serverStatus`: returns statistics gathered by the FTDC service, including information related to CPU, disk and memory usage. This includes read and write latencies, statistics on primary node elections, locking information and the number of bits spilled to disk. Note that this data is accumulated across a single machine and does not pertain to a specific database or collection. `serverStatus` can, however, be repeatedly executed to gather finer-grained statistics. These statistics will still be across a single machine but could be at the granularity of a second.
8. `setParameter`: sets a variety of server parameters. For a full list, see [3];
9. `$allCollectionStats`: returns statistics over all the collections;
10. `$operationMetrics`: returns various internal billing metrics such as the number of documents read, the number of index entries read or the number of cursor seeks;
11. `$currentOp`: returns information on in-progress operations;
12. `top`: returns usage statistics for each collection;
13. `getLog`: returns the most recent 1024 logged `mongod` events from a cached data structure.
14. `$connPoolStats`: returns information about the open connections to the cluster and replica set;
15. `setClusterParameter`: allows a user to change the `changeStreamOptions` parameter of a cluster. This parameter changes the retention policy of pre- and post-change document images in a change stream.

[7]Note that aggregation pipeline stages have a $ sign prepended to their name.

Role	Surface	Schedule	Leakage
`read`	edb	$[n]$	shape, upshape, writetime, compuniq
`readWrite`	edb	$[n]$	shape, upshape, writetime, compuniq
`dbAdmin`	edb	$[n]$	shape, upshape, writetime, compuniq
`clusterManager`	edb	$[n]$	opeq, shape, upshape, writetime, compuniq, valeq_f [8]
`clusterMonitor`	edb	$[n]$	opeq, shape, upshape, writetime, compuniq, valeq_f [8]
`hostManager`	edb	$[n]$	opeq, shape, upshape, writetime, compuniq, valeq_f [8]
`backup`	edb	$[n]$	opeq, shape, upshape, writetime, compuniq, valeq_f [8]
`restore`	edb	$[n]$	shape, upshape, writetime, compuniq

Table 2: Insider analysis against eight MongoDB built-in roles.

8 Insider Analysis

In this section, we study the security guarantees that `QE` offers against eight MongoDB built-in roles. We summarize this analysis in Table 2 and begin with some background.

Built-in roles. MongoDB uses role-based authentication to manage access to its resources. A role consists of a set of: (1) authorized resources, like databases, collections, and clusters; and (2) authorized commands to read, write, update and manage the databases and system. In our analysis, a role's surface consists of the resources it is authorized to access and the outputs of its authorized commands. When we say that a role has access to a particular resource (e.g., `local.oplog`), we implicitly assume it is authorized to run a `$find` command on it, so we do not mention these `$find`s explicitly in its list of authorized commands. We primarily focus on the built-in roles of MongoDB Enterprise Advanced (`EA`), but our analysis also applies to the built-in roles of `Atlas`, with the exception of a few instances which we highlight. For more on MongoDB's built-in roles, see [2].

Reminder. We assume throughout that the roles do not grant access to the encryption keys. If a `QE`-encrypted collection includes both plaintext and `QE`-encrypted fields then the role may be able to access/read/write the plaintext data but it cannot decrypt the `QE`-encrypted values or the `QE` auxiliary collections.

8.1 Database User Roles

Database user roles include the `read` and `readWrite` role which allow a user to execute read and read and write operations, respectively.

Read role. The `read` role allows a user to perform read operations on a set of authorized databases and their associated management collections:

[8] This leakage profile is the result of a preliminary analysis. A formal and complete treatment is underway and when finished this document will be updated with the final results.

- *surface*: the surface of the `read` role includes its authorized databases and their associated $*$.`system.js` collections. Note that when creating a `read` role it is *possible* to authorize it to access the `config`.$*$, `local`.$*$ and `admin`.$*$ databases. Here we assume that `read` is *not* authorized to access the `local`.$*$ database. The surface also includes the output of its authorized commands like `$collStats` and `$planCacheStats`. For the full list of commands, see [44].
- *schedule*: we assume `read` can access its surface after every operation;
- *view:* the only databases/collections in `read`'s surface that holds `QE`-related data is $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$. In addition, none of `read`'s authorized operations return `QE`-related data. Therefore, `read`'s view is that of an $(\mathsf{edb}, [n])$-adversary.

Read and write role. The `readWrite` role allows a user to perform read and write operations on a set of authorized databases and their associated management collections:

- *surface*: the surface of the `readWrite` role includes its authorized databases and their associated $*$.`system.js` collections. Note that when creating a `readWrite` role it is possible to additionally grant it access to the `config`.$*$, `local`.$*$ and `admin`.$*$ databases. Here, we assume that the `readWrite` is *not* authorized to access the `local`.$*$ database. The surface also includes the output of its authorized commands like `$collStats` and `$planCacheStats`. For the full list of commands, see [44].
- *schedule*: we assume `readWrite` can access its surface after every operation;
- *view:* the only databases/collections in `readWrite`'s surface that holds `QE`-related data is $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$. In addition, none of `readWrite`'s authorized operations return `QE`-related data. Therefore, `readWrite`'s view is that of an $(\mathsf{edb}, [n])$-adversary.

8.2 Database Administration Roles

The `dbAdmin` role allows a user to perform administrative operations on a set of authorized databases and their associated management collections:

- *surface*: the surface of the `dbAdmin` role includes its authorized databases and their associated $*$.`system.profile` collection *if the profiler is turned on.* The surface also includes the output of its authorized commands like `$collStats`, `$planCacheStats`, `listIndexes` and `renameCollectionSameDB`. For a full list of `dbAdmin`'s authorized commands, see [44].
- *schedule*: we assume `dbAdmin` can access the elements of its surface after every operation;
- *view*: the only databases/collections in `dbAdmin`'s surface that has `QE`-related data is $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$. In addition, the authorized operations that return `QE`-related data only return information (e.g., statistics) about `edb`. Therefore, `dbAdmin`'s view is that of an $(\mathsf{edb}, [n])$-adversary.

8.3 Cluster Administration Roles

Cluster administration roles include the `clusterManager`, `clusterMonitor` and `hostManager` roles. Unlike the previous roles, which are applied over specific databases, these roles are applied at the level of MongoDB clusters and replica sets.

Cluster manager role. The `clusterManager` role allows a user to manage a cluster.

- *surface*: the surface of the `clusterManager` role includes all the databases on a cluster, including user databases and their associated management collections like $*$.`system.profile` and $*$.`system.js`. It also includes the system-wide internal databases like `config`.$*$, `admin`.$*$ and `local`.$*$. The exact kind of access is varied but, for our purposes, we assume it has full access to these databases to capture the worst case. `clusterManager` can run administrative commands like `addShard`, `listShards`, `removeShard`, `$planCacheStats` and `setClusterParameter`. For the full list of commands, see [44].
- *schedule*: we assume `clusterManager` can access its surface after every operation;
- *view*: the only databases/collections in `clusterManager`'s surface that includes `QE`-related data are $\mathsf{edb} = (\texttt{edc}, \texttt{esc}, \texttt{ecoc})$ and `local.oplog`. Note that `local.oplog` stores *transaction numbers* which are session-wide unique identifiers associated to every transaction and that are generated using a counter. Transaction numbers are created and assigned even for failed transactions so the difference between two transactions numbers is a function of the number of failed transactions which is itself correlated with the number of concurrent insert/updates on a field/value pair which is, in turn, correlated with that field/value pair's frequency. These correlations are loose and affected by many variables but, in the worst case, could reveal frequency information. None of its authorized operations output any `QE`-related data. Therefore, its view is that of an (dbms, ∞)-adversary.

Remark. Our analysis of `clusterManager` assumed that it had full access to `local`.$*$, which is not the case in practice. More precisely, `clusterManager` can only update `local`.$*$, but one could theoretically use updates to learn partial information by issuing updates for known field/value pairs and observing whether the update went through or not. Because of this, we modeled the worst case and assumed full access, but, in practice, this is highly "pessimistic".

Cluster monitor role. The `clusterMonitor` role allows a user to monitor a cluster:

- *surface*: the surface of the `clusterMonitor` role includes all the databases on a cluster, including user databases and their associated management collections like $*$.`system.profile` and $*$.`system.js`. It also includes system-wide internal databases like `config`.$*$, `admin`.$*$ and `local`.$*$. As for `clusterManager`, the exact access is varied but, for our purposes, we assume it has full access to these databases to capture the worst case. `clusterMonitor` can run a larger number and more powerful set of commands than `clusterManager`, including, `serverStatus`, `$collStats`, `$allCollectionStats`, `$operationMetrics`, `$currentOp`, `top`, `getLog`, and `$connPoolStats`. For a full list, see [44].
- *schedule*: we assume `clusterMonitor` can access its surface after every operation;

- *view*: the only databases/collections in `clusterMonitor`' s surface that include `QE`-related data are $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$ and `local.`$*$ which includes `local.oplog`. The outputs of `serverStatus`, `$operationMetrics` and `$connPoolStats` contain `QE`-related data. Because of this we assume the worst-case and include the transcripts of all query operations in the view. In summary, in the worst-case, the view of `clusterMonitor` is the view of a (dbms, ∞)-adversary.

Host manager role. The `hostManager` role allows a user to monitor and manage clusters.

- *surface*: the surface of the `hostManager` role includes all the databases on a cluster, including user databases and their associated management collections like $*$`.system.profile` and $*$`.system.js`. It also includes system-wide internal databases like `config.`$*$, `local.`$*$ and `admin.`$*$. As for the roles above, the exact access is varied but, for our purposes, we assume it has full access to these databases to capture the worst case. `hostManager` has less monitoring capabilities than `clusterManager` but is authorized to run a wider set of server management commands, including, `setParameter`, `trafficRecord` and `auditConfigure`. For a full list, see [44].

- *schedule*: we assume `hostManager` can access the elements of its surface after every operation;

- *view*: if auditing is not enabled, the only databases/collections in `hostManager`'s surface that include `QE`-related data are $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$. `hostManager`'s only authorized command that can reveal `QE`-related data is `trafficRecord`, whose output could include the transcripts of query operations. Therefore, the view of `hostManager` is that of a (dbms, ∞)-adversary. On the other hand, if auditing is enabled, then `hostManager` has access to `auditLog` as well. However, its view remains that of a (dbms, ∞)-adversary.

8.4 Backup and Restoration Roles

We analyze two roles: `backup` and `restore`. These roles also operate at the level of a cluster as opposed to specific databases.

Backup role. The `backup` role allows a user to backup the data on a cluster:

- *surface*: the surface of the `backup` role includes all the databases on a cluster, including user databases and their associated management collections like $*$`.system.profile` and $*$`.system.js` and system-wide internal databases like `config.`$*$, `local.`$*$ and `admin.`$*$. `backup` can run a variety of commands including `serverStatus`. For a full list, see [44].

- *schedule*: we assume `backup` can access the elements of its surface after every operation;

- *view*: the only databases/collections in `backup`'s surface that include `QE`-related data are $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$ and `local.`$*$ which includes `local.oplog`. Its only authorized command that can contain `QE`-related data is `serverStatus`. Therefore, the view of `backup` is that of a (dbms, ∞)-adversary.

Restore role. The `restore` allows a user to restore backed-up data to a cluster.

- *surface*: the surface of the `restore` role includes all the databases on a cluster, including user databases and their associated management collections like $*$.`system.js` and system-wide internal databases like `config`.$*$ and `admin`.$*$. Note that when creating a `restore` role, it is possible to authorize it to access $*$.`system.profile` and `local`.$*$ databases. Here, we assume that the `restore` is *not* authorized to do so. `restore` can run a variety of commands including `$createIndex` and `$createCollection`.
- *schedule*: we assume `restore` can access the elements of its surface after every operation;
- *view*: the only databases/collections in `restore`'s surface that include `QE`-related data are $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$. None of its authorized commands contain any `QE`-related data. Therefore, the view of `restore` is that of an $(\mathsf{edb}, [n])$-adversary.

8.5 Custom Roles

MongoDB allows one to define custom roles. If a custom role is necessary, we suggest creating roles as close as possible to the built-in roles analyzed in Section 8 and, if possible, with a smaller subset of permissions. For example, `hostManager` has access by default to the trafficRecord and auditConfigure commands which allow it to capture all the network traffic received by a MongoDB instance and to set new audit configurations for `mongod` and `mongos`, respectively. Removing access to these commands from `hostManager`' set of authorized commands significantly improves `QE`'s leakage profile against this role.

9 General Guidelines

In this section, we describe a list of guidelines for setting various `QE` parameters as well as discuss several security considerations. The guidelines in this section apply to all query types unless stated otherwise; the compaction and contention discussions are stated for equality fields, and Section 11 adapts them to string fields.

9.1 Scheduling Compactions

Compaction is a `QE` operation that shrinks the size of an encrypted database by removing stale data from the `ecoc` and `esc`. After compaction, the `ecoc` is reset to an empty collection and the `esc` shrinks in size. Since compactions need to be started by clients/users, we provide some guidelines for when to schedule them. The discussion in this subsection is stated for equality fields; Section 11 adapts it to string fields, for which compactions matter considerably more. Consider a simple `QE`-encrypted collection with a single encrypted field, `name`, and $1,000$ documents such that 500 documents have `name = Alice`, 250 documents have `name = Bob` and 250 have name `name = Eve`. Prior to compaction, the `ecoc` and the `esc` of such a collection will each hold $1,000$ meta documents (recall that meta documents hold metadata and not client data). After compaction, the `esc` shrinks to 3 meta documents and the `ecoc` is empty. This is

a storage saving of around 99.85%. Generally speaking, we have

$$\mathsf{metadocs}(\texttt{esc}) \geq \mathsf{compactions} \cdot \mathsf{encfields},$$

where $\mathsf{metadocs}(\texttt{esc})$ is the number of meta documents in the `esc`, $\mathsf{compactions}$ is the number of compactions executed so far and $\mathsf{encfields}$ is the number of encrypted fields in the collection. This suggests that the less often one compacts the better the storage savings. More precisely, the ith compaction removes the meta documents that insertions added to the `esc` since the previous compaction and leaves one meta document per unique field/value pair inserted in that interval, so it shrinks the `esc` by exactly

$$\mathsf{reclaim}_i \triangleq \mathsf{encfields} \cdot \mathsf{docinserts}_i - \mathsf{valinserts}_i, \tag{1}$$

which we call the *reclaim* of the ith compaction. Here, $\mathsf{docinserts}_i$ is the number of documents inserted between the $(i-1)$th and the ith compaction and $\mathsf{valinserts}_i$ is the number of unique encrypted field/value pairs inserted in the same interval.[9]

Security. As discussed in Section 6, compactions reveal $\mathsf{valinserts}_i$, the number of unique `QE`-encrypted field/value pairs since the last compaction.[10] In the example above, $\#\mathsf{valinserts} = 3$ which means that the adversary learns that the number of unique field/value pairs inserted since the last compaction was 3 but will not learn the inserted values themselves. Intuitively speaking, in the absence of auxiliary information, the adversary will only know that the inserted documents are one out of

$$\binom{m}{m_i} \cdot m_i! \cdot \left\{ {n_i \atop m_i} \right\}$$

possibilities, where $n_i \triangleq \mathsf{docinserts}_i$, $m_i \triangleq \mathsf{valinserts}_i$, $m \triangleq \mathsf{possiblevals}$ is the number of possible values that a field can take, and $\left\{ {n_i \atop m_i} \right\}$ denotes the Stirling numbers of the second kind. In our example above, with a single `QE`-encrypted field, `name`, and $1,000$ possible names, the adversary would know that the set of inserted documents are 3 documents out of a set of $2^{1,612}$ possibilities. We stress that this is only a heuristic argument and not a formal argument because it does not take into account auxiliary information. It suggests that a compaction that covers more insertions reveals less about them, which is one reason to stretch the period beyond the cost optimum of Equation (**??**); another is that an adversary that is present in the system for a bounded window of time observes fewer compaction counts when the period is longer. We caution, however, that comparing the leakage of two compaction schedules in full generality is subtle, because the counts released by frequent compactions do not determine the counts released by rare ones or vice versa, and a principled treatment of this choice remains open.

[9] For simplicity, we only discuss the case of insertions. Accounting for updates and deletions will slightly change the equation. Note however that the same reasoning will still hold.

[10] This is the worst-case and occurs when `QE`'s contention factor is set to 1. The contention factor is a parameter that allows one to tradeoff insert and update throughput for find performance (see Section 9.2). If the contention factor is larger than 1, then compactions reveal the number of unique QE-encrypted field/value/partition triplets since the last compaction.

9.2 Setting the Contention Factor

Concurrent operations may result in *contention* which, in turn, decreases the throughput of read and write operations. For example, contention can result from multiple clients inserting documents with the same _id value. Contention could also occur if multiple clients insert or update documents with the same field/value pair. For every QE-encrypted field f, QE defines a parameter $\mathsf{cf}_f \geq 0$ called the *contention factor*. Increasing the contention factor of a field increases the insert and update throughput over that field, but decreases the performance of find operations. All contention factors are set to 8 by default because—based on extensive experiments—we found that it provides good performance for a variety of workloads.

Changing the contention factor. The default contention factor should be appropriate for most settings but it can be tuned to the characteristics of one's workload. In particular, if the workload tends to have a very large number of concurrent insertions or updates on a field f, then it might be reasonable to increase the contention factor in order to increase insertion or update throughput. Conversely, if the workload tends to have a very small number of concurrent insertions or updates, then decreasing the contention factor might help improve read throughput. In the following, we describe a way to tune the contention factor as a function of the expected workload. Suppose ω_f is an estimate of the total number of concurrent insertions or updates on a given field f within a small interval of time, say, 30ms. The contention factor cf_f can then be set as

$$\mathsf{cf}_f = \left\lceil \frac{\omega_f^* \cdot (\omega_f^* - 1)}{0.2} \right\rceil,$$

where $\omega_f^* = \lceil \omega_f / \mathsf{possiblevals} \rceil$. Note that if the number of concurrent insertions/updates on a field is not known, ω_f could be set to the server's number of virtual cores. Note that the above formula is designed to bound the probability of contention to be at most 0.1 but it can be tuned to achieve different bounds. As an example, if $\omega = 100$, $\mathsf{possiblevals} = 100$, then setting $\mathsf{cf}_f = 1$ would be enough to avoid contention. In this example, the contention factor is considerably smaller than the default value of 8 and will lead to increased read throughput without impacting the insertion and update throughput. On the other hand, if $\omega = 100$, $\mathsf{possiblevals} = 50$, then setting $\mathsf{cf}_f = 10$ would be enough to avoid contention. In this example, the contention factor is larger than the default value of 8 and will result in decreased read throughput without impacting the insertion or update throughput.

Impact on find efficiency. As mentioned above, increasing the contention factor can decrease the efficiency of find operations. More precisely, the find efficiency is roughly

$$\mathsf{time}_{\texttt{find}} = O\Big(\mathsf{cf}_f \cdot \log\big(\mathsf{metadocs}(\mathsf{esc})\big) + \#\mathbf{R}\Big),$$

where $\mathbf{R}$ is the set of documents returned by the find operation. Notice that if $\mathbf{R}$ is small, the contention factor will impact the read efficiency more. In particular, using large values of cf_f can significantly decrease read efficiency.

Impact on leakage. The contention factor should never be set as a function of secret/sensitive information. For example, it should not be set as a function of the frequencies of field/value pairs. Consider the case where $\mathsf{valinserts} = 1000$ and $\omega = 100$. Based on the approach outlined above, one can set $\mathsf{cf}_f = 1$. On the other hand, suppose one notices that for some field f, there are about 20 values that tend to appear very frequently in the collection and, therefore, sets $\mathsf{cf}_f > 1$. This choice of cf_f could reveal more information to a multi-snapshot adversary because the contention factor is calculated as a function of the insert/update distribution. Because of this, we suggest that the contention factor be kept at the default value and that, if it must be tuned, that it be set according to the approach outlined above.

9.3 Handling Values with Sensitive Lengths

The values of `QE`-encrypted fields are encrypted using AES256-CBC-SHA256 which is an authenticated encryption scheme based on `encrypt-then-mac` instantiated with AES256-CBC and `HMAC-SHA256`. Note that AES256-CBC reveals an approximation of the plaintext. This is because it pads the plaintext to the next multiple of 128 bits so, given a ciphertext ct, one knows that

$$|\text{ct}| - 128 < |m| \leq |\text{ct}|.$$

For fixed-length data types, the length of the value is public and already known to the adversary so we focus on variable-length data types like `string` or `binData` [43].

Plaintext recovery through length. One of problems that can occur with revealing the length of values is that the length could be used to recover the plaintext values. This depends, however, on the workload and the auxiliary information the adversary has about the workload. For example, consider a field that has 3 possible values v_1, v_2 and v_3 such that $|v_1| = 120$, $|v_2| = 200$ and $|v_3| = 300$ and that these possible values are known to the adversary—though which document holds which value is unkown. Encrypting these values with AES256-CBC results in three ciphertexts ct_1, ct_2 and ct_3 of lengths $|\text{ct}_1| = 128$, $|\text{ct}_2| = 256$ and $|\text{ct}_3| = 384$. In this case, the ciphertext not only reveals an approximation of value lengths but these approximations can be used to recover the plaintexts as follows. We know that ct_1 must be an encryption of v_1 since it is the smallest ciphertext. Similarly, we know that ct_2 must be an encryption of v_2 since it is the second smallest ciphertext and we know that ct_3 must be an encryption of v_3 since it is the largest ciphertext. To handle collections with sensitive fields that have values of distinct lengths, we propose two client-side approaches that avoid the above issue. *Note that these solutions are not provided by the MongoDB drivers and must be implemented by the application.*

Padding. The simplest solution is to pad each value of the field to length of the maximum-length value in the field's domain before encrypting it. For example, suppose a field f can take values in the set $\mathbf{V} = \{v_1, \cdots, v_n\}$ and suppose v_n is the value of largest length. Before encrypting the collection with `QE`, one should pad every f-value v such that $|v| = |v_n|$. The downside of this approach is that it increases the size of the collection.

Mapping. Another solution is to map the f-values to another set of fixed-length values before passing the collection to `QE`. More precisely, suppose a field f has domain $\mathbf{V} = \{v_1, \cdots, v_n\}$

where the values have a different lengths. With this approach, one would create a new domain $\mathbf{W} = \{1, \cdots, n\}$ of fixed-length numbers and a map $T : \mathbf{V} \to \mathbf{W}$ that maps values in $\mathbf{V}$ to values in $\mathbf{W}$. This mapping T can be represented as a table for example. Here, we assume that n is not too large and that the numbers 1 through n are represented using a fixed number of bits, e.g., $\log_2(n)$ bits. Before encrypting the collection with `QE`, one would replace every f-value v in the collection with $w = T(v)$. Unlike the padding approach, the size of the collection does not increase much, but the tradeoff is that the table T needs to be stored and managed client-side.

9.4 Using the Aggregation Pipeline

MongoDB's aggregation pipeline is a framework that allows clients to build queries as a sequence of stages, each of which processes the documents in a collection and passes the result as input to subsequent stages. Aggregation pipelines can be evaluated over `QE`-encrypted collections, so we discuss the security guarantees that `QE` provides when running aggregation pipelines.

Automatic encryption. When `QE` is used in automatic mode, the drivers have access to a client-side query analysis module which will constrain the stages that can be used in a pipeline. In particular, the query analysis module will disallow any stage that modifies the database. This will guarantee that no pipeline can change the leakage profile of `QE` with respect to snapshot adversaries.

Explicit encryption. When `QE` is used in explicit mode, the drivers do not have access to client-side query analysis. This means any stage and pipeline can be executed which could alter `QE`'s leakage profile. Therefore, when using `QE` with explicit encryption, we recommend avoiding any stages that can modify the database.

9.5 Additional Guidelines

Partial encryption of documents. `QE` allows clients to choose which fields are encrypted. This choice might seem straightforward at first: simply encrypt the fields that are "sensitive". This choice is far from simple, however, and requires a careful and nuanced understanding of how database encryption works. The main difficulty is that fields can be correlated in the sense that knowledge of one can provide information about the other. In such a case, encrypting only one of the two fields is not useful because the other field could reveal information about the first. For example, consider a collection with the two fields `city` and `state`. Encrypting the `state` field alone does not provide any meaningful security because one can likely guess a document's state from its city. Note, however, that these kinds of correlations can be more subtle so one has to carefully decide which set of fields need to be encrypted. If in doubt, our suggestion is to encrypt all fields, even if this increases storage and performance overhead.

Schema. When using `QE`, one specifies the fields that should be encrypted, what kind of queries need to be supported on those fields, the contention factor and which data encryption keys should be used with which field in a `QE` schema. The schema is stored server-side but it can also be stored client-side. Currently, `QE` does not provide any mechanisms to verify the integrity of a schema so

if it is only stored server-side and the server is compromised, an attacker could tamper with the schema so that a previously QE-encrypted field is labeled as unencrypted. The consequence of this is that if the client retrieves the tampered schema, it will send plaintext values for that field instead of encrypted values. Because of this, our suggestion is to keep the QE schema client-side at all times.

Enabling audit logs. MongoDB Enterprise Advanced (EA) has an auditing capability that, when enabled, tracks a variety of system and user events. We note that enabling the audit log will allow certain roles to recover details of all the interactions between a QE client and the server which can affect QE's leakage profile. Out of all the roles we analyzed in Section 8, the only one that can access the audit log—when auditing is turned on—is hostManager. This, however, does not increase its view since the information contained in the audit log can also be obtained from the $trafficRecord command.

10 Guidelines for Range Queries

In this Section, we describe guidelines for setting range-specific parameters for QE and discuss their performance and security impact. Note that in QE, these parameters are already set to default values that should provide good performance and storage tradeoffs for a variety of workloads.

10.1 Reducing Storage for Bounded Domains

As discussed in Section 3.1, QE supports a variety of numerical types, including, sint32, int64 (date), sint64, bin64 (IEEE 754 binary64) and dec128 (IEEE 754 decimal128). The storage as well as the performance of all the operations grows as a function of the numerical field's domain. That is, the larger a field's domain is, the more storage QE will use and the slower it will perform. Because of this, it is important that applications use the exact numerical type they need. For example, if a field really only stores signed 32-bit integers then sint32 should be used as opposed to the larger int64 or bin64.

Bounded integer domains. Because QE's storage and performance of all its operations is affected by the domain size of a field, applications can optionally specify more precise bounds on a field's domain. In the case of integer values (i.e., sint32, int64 and sint64), one can set two parameters lb and ub that specify the lowest and highest possible values that can be queried with respect to this field. For example, even though a field age is represented as an sint32, an application might choose to set $\mathsf{lb} = 0$ and $\mathsf{ub} = 150$ instead of using the entire sint32 domain which ranges from $-2,147,483,648$ to $2,147,483,647$. Doing so would yield a saving of about $4\times$ in storage and find, update and insert performance.

To see why, first note that the domain sizes of sint32, sint64 and int64 are $N = 2^{32}$, $N = 2^{64}$, and $N = 2^{64}$, respectively. On the other hand, for any of these numerical types, if lb and ub are set then the domain size will be

$$N = 2^{\lceil \log_2(\mathsf{ub}-\mathsf{lb}+1)\rceil}.$$

Very roughly speaking, QE's storage grows logarithmically with the size of the domain N. More precisely, every encrypted value requires $O(k \cdot \log_2 N)$ bits instead of $O(|v|_{\mathbf{2}})$, where $|v|_{\mathbf{2}}$ is the bit length of the numerical value and k is the security parameter. Note that $k \geq |v|_{\mathbf{2}}$ for all numerical data types. Going back to the previous example, we have $\log_2 N = \lceil \log_2(\mathsf{ub}-\mathsf{lb}+1) \rceil = \lceil \log_2(150-0+1) \rceil = 8$ which is $4\times$ smaller than the logarithm of the (unbounded) domain size of `sint32`.

Bounded floating-point domains. In the case of floating-point values (i.e., `bin64` and `dec128`), applications can optionally specify precise bounds on a field's domain by setting three parameters: lb, ub and prc which specify the lowest possible value, the highest possible value and the query precision that should be supported. The lb and ub parameters are as above. The prc parameter determines the precision used during a query, in other words, the number of digits after the decimal point that will be considered when querying. For instance, for a field that represents a blood sugar rate (in mmol/L), one can set $\mathsf{lb} = 0.0$, $\mathsf{ub} = 20.0$ and $\mathsf{prc} = 1$ instead of using the entire domain of the `bin64` numerical type which can represent roughly 2^{64} different values. This would lead to about $8\times$ savings in terms of storage requirements and find, update, and insert operations performance. To see why, note that the domain sizes of `bin64` and `dec128` are $N = 2^{64}$ and $N = 2^{128}$, respectively. On the other hand, if lb, ub and prc are set, then the bounded domain size of a `bin64` field will be

$$N = \min\left(2^{64}, 2^{\left\lceil \log_2\left(\left(\mathsf{ub}-\mathsf{lb}+1\right)\cdot 10^{\mathsf{prc}}\right)\right\rceil}\right),$$

and the bounded domain size of a `dec128` field will be

$$N = \min\left(2^{128}, 2^{\left\lceil \log_2\left(\left(\mathsf{ub}-\mathsf{lb}+1\right)\cdot 10^{\mathsf{prc}}\right)\right\rceil}\right).$$

Very roughly speaking, and similarly to the integer case, QE's storage grows logarithmically with the size of the domain N, i.e., every encrypted value requires $O(|v|_{\mathbf{2}} \cdot \log_2 N)$ bits of storage. Going back to the previous example, we have $\log_2 N = \log_2(\min(2^{64}, 2^{\lceil \log_2((20-0+1)\cdot 10^1)\rceil})) = \log_2(2^{\lceil \log_2(200)\rceil})) = 8$ which is $8\times$ smaller than the logarithm of the (unbounded) domain size of `bin64`.

It is important to note that setting the query precision *does not change the numerical values stored in the collection in any way*; it only affects how QE will execute range queries over the collection. In other words, when storing `bin64` or `dec128` values with the lb, ub and prc parameters set, the values stored in the (encrypted) collection remain as-is.

Effects of floating-point rounding. We note that floating-point rounding combined with a bounded domain (i.e., setting the lb, ub and prc parameters) can affect the accuracy of a range query. For example, when considering `bin64` values, the 64-bit numbers 1.000 000 000 000 000 1 and 1.000 000 000 000 000 2 are both rounded to 1.000 000 000 000 000. So if, say, $\mathsf{label} = 0$, $\mathsf{ub} = 1000$ and $\mathsf{prc} = 15$, QE will view them both as 1.000 000 000 000 000 during find operations. As a result, when querying for the range $[1.000\ 000\ 000\ 000\ 000\ 2, 10]$, the documents containing 1.000 000 000 000 000 1 will be returned even though they are technically out of range.

10.2 Reducing Storage via the Sparsity Factor

As we have seen, a field's domain size affects the storage requirements of `QE` as well as its find, insert and update performance. However, `QE`'s storage consumption can also be tuned using a parameter `sp` called the *sparsity factor*. The sparsity factor can be set to integer values in $\{1, \ldots, 8\}$ but the default is set to 2. The larger the sparsity factor is, the less storage `QE` will consume but the slower find operations will run.

More precisely, the storage requirements of `QE` when encrypting a numerical field is

$$O\left(k \cdot \frac{\log N - \mathsf{tf}}{2^{\mathsf{sp}}}\right) \tag{2}$$

per encrypted field, where `tf` is another parameter called the *trimming factor* discussed below. Based on this, one can see that increasing the sparsity factor from 2 to 4 will decrease the storage overhead by a factor of 2.

Effect on inserts and updates. Increasing the sparsity factor also improves insert and update performance because it reduces the number of tags that need to be inserted in the `esc`. More precisely, inserting a document is

$$O\left(\left(\log N - \mathsf{tf}\right) \cdot \frac{\log\left(\mathsf{metadocs}(\mathtt{esc})\right)}{2^{\mathsf{sp}}}\right), \tag{3}$$

assuming no conflicts occur.

Find performance. As mentioned above, increasing the sparsity factor decreases `QE`'s storage requirements but increases its find performance. More precisely, find operations with a range filter are

$$O\left(\left(2^{\mathsf{tf}} + \log N\right) \cdot \mathsf{cf} \cdot \log N \cdot \log\left(\mathsf{metadocs}(\mathtt{esc})\right) \cdot 2^{\mathsf{sp}}\right). \tag{4}$$

So while increasing the sparsity factor from 2 to 4 decreases the required storage by a factor of 4, it increases find time by a factor of 4.

Recommendations. We recommend keeping the default values for the sparsity factor, which is 2, as our experiments show that this achieves the best tradeoff for many workloads. However, there are cases where custom sparsity factors can lead to significant improvements. Specifically, we recommend increasing the sparsity factor if workload has significantly more insert and update operations than find operations. On the other hand, one should decrease the sparsity factor if one prefers the best find performance at the cost of slower insert and update times and if storage overhead is not a major concern.

10.3 Reducing Contention via the Trimming Factor

The trimming factor, `tf`, is a `QE` parameter that can range from 0 to $\log N$, where N is the size of a field's domain, and that is set to 6 by default. While the sparsity factor is used to tune `QE`'s

tradeoff between storage, update, insert, and find performance, the trimming factor tunes QE's throughput under concurrent inserts and updates. Specifically, increasing the trimming factor leads to increased throughput and smaller storage overhead but higher find times.

The effect of the trimming factor on the storage overhead as well as the insert, update and find performances can be understood from Equations (2), (3) and (4). Specifically, one can see from Equation (2) how increasing the trimming factor reduces storage, from Equation (3) how it reduces the insert and update times and from Equation (4) how it increases the find time.

Recommendations. As with the sparsity factor, we recommend using the default value of the trimming factor since this achieves the best tradeoffs for most workloads. However, there are cases where custom trimming factors lead to significant improvements. Specifically, we recommend increasing the trimming factor to increase throughput of workloads that have significantly more inserts and updates than finds. On the other hand, one should decrease the trimming factor if one would rather achieve better find performance at the cost of slower inserts and updates, lower throughput and if storage is not a major concern.

10.4 Correctness & Security Considerations

When setting QE's parameters for numerical fields there are two important issues one needs to take into account. The first one is related to the correctness of find operations and the second is related to security.

Correctness. There are combinations of parameters for which find queries may fail. We stress that this will never occur with the default parameters. More precisely, find queries may fail if the following Equation is violated:

$$\min\left(N, 2^{\mathsf{sp}-1}\cdot\left(2^{\mathsf{tf}}+2\log_2 N-1\right)\right) < 300,000,$$

where N is the field's domain size, sp is the sparsity factor and tf is the trimming factor. At a very high level, this limit is due to the size limit of BSON documents.

Security. For security reasons, it is important that the trimming factor be independent of the distances between the values in the collection. More precisely, if a collection has values $\{v_1, \ldots, v_n\}$ for some field f, the trimming factor should not be set as a function of the elements in the set

$$\left\{v_i - v_j : \text{for all } i, j \in [n] \text{ s.t. } i \neq j\right\}.$$

Practically speaking, the easiest way to achieve this is to only increase or decrease the trimming factor to affect throughput while ignoring the specific values in the collection.

11 Guidelines for String Queries

QE string search operations incur the highest storage and write overhead among all QE query types. As described in Section 3.2, every insertion adds a number of tags, and, as we will see below,

this number of tags is determined by the types of queries the field supports, as well as the choice of underlying parameters. In this section, we quantify the storage cost, adapt the compaction guidelines of Section 9.1 to the string case, and discuss additional security considerations.

11.1 The Tag Count of a Value

First, recall that `QE` can support the following string search operations: (1) exact and substring search, (2) exact and prefix search, (3) exact and suffix search, (4) exact, suffix, and prefix search. Substring, suffix, and prefix search are parametrized by the following quantities:

- $\texttt{mlen} \in \mathbb{N}_{>0}$ is the maximum number of characters a string can have for the purposes of substring search;
- $\texttt{ub} \in \mathbb{N}_{>0}$ is the upper bound on the number of characters a substring, prefix, or suffix search query can have. Note that $\texttt{ub} \leq \texttt{mlen}$;
- $\texttt{lb} \in \mathbb{N}_{>0}$ is the lower bound on the number of characters a substring, prefix, or suffix search query can have. Note that $\texttt{lb} \leq \texttt{ub}$.

`QE`'s storage, insertion time, and find-time complexity are all functions of the tag count, which we denote by $t(v)$. Intuitively, the tag count, in the substring case for example, depends not only on the number of substrings the inserted value contains but also on the number of dummy elements. As explained in Section 3.2, a design principle of `QE` is that the leakage against a multi-snapshot adversary consists only of the padded length of the inserted value, which we denote by $\ell_{\max}(v)$.[11] This principle applies to equality and range as well as to strings. As a consequence, the number of tags should depend only on this padded length and the public parameters corresponding to the string field, i.e., lb, ub, and/or `mlen`. Below, we show how to calculate the number of tags for the prefix-or-suffix, prefix-and-suffix, and substring cases.

- *prefix or suffix case*: the number of tags is equal to

$$t(v) = \min\Big(\mathsf{ub}, \ell_{\max}(v)\Big) - \mathsf{lb} + 1 + 1, \tag{5}$$

  where the trailing 1 accounts for the extra tag that supports exact search.

- *prefix and suffix case*: the number of tags is equal to

$$t(v) = \min\Big(\mathsf{ub}_1, \ell_{\max}(v)\Big) - \mathsf{lb}_1 + 1 + \min\Big(\mathsf{ub}_2, \ell_{\max}(v)\Big) - \mathsf{lb}_2 + 1 + 1, \tag{6}$$

  where ub_1, lb_1, ub_2, lb_2 are the upper and lower bounds for the prefix and suffix cases, respectively; and, as above, the trailing 1 accounts for exact search on the inserted value.

- *substring case*: the number of tags is equal to

$$t(v) = \min\left(\sum_{\ell=\mathsf{lb}}^{\mathsf{ub}} \Big(\texttt{mlen} - \ell + 1\Big), \sum_{\ell=\mathsf{lb}}^{\min(\mathsf{ub},\ell_{\max}(v))} \Big(\ell_{\max}(v) - \ell + 1\Big)\right) + 1, \tag{7}$$

[11] Note that since we use AES256-CBC, the ciphertext length exceeds the actual (folded) value length by at most 16 bytes, as this mode of operation pads the ciphertext to a multiple of 16 bytes. Note also that every plaintext value is first encoded to add a 5-byte header, so $\ell_{\max}(v)$ refers to the ciphertext length minus 5 bytes.

where the first sum computes the number of possible substrings of a string of length `mlen` given a specific lower and upper bound, whereas the second sum computes the number of substrings in a string of length $\ell_{\max}(v)$; the number of tags is the minimum of these two sums, since they represent the same underlying quantity computed under two different constraints. The number of tags $t(v)$ is therefore not the number of substrings of v given the field's parameters; it also counts dummies, as discussed in Section 3.2.

Concrete example. The number of tags generated for a substring field is substantially higher than the number generated for a prefix, suffix, or prefix-and-suffix field with similar lower and upper bounds. To see this, consider the following example: a lower bound of 2, an upper bound of 6, and a maximum string length of 50 (for the substring case). A short name like `Bob` has a padded length of 11 bytes, so the number of tags for a prefix or suffix field would be $\min(6, 11) - 2 + 2 = 6$, whereas for a substring field the number of tags would be $(10 + 9 + 8 + 7 + 6) + 1 = 41$. Among these 41 tags, only 3 correspond to real substrings, while the remaining 38 are dummy elements. This is already roughly 7× as many tags; and if we were to encrypt a value with 50 characters (instead of the three-character value `Bob`), the number of tags for a prefix field would remain unchanged at 6, whereas for the substring case it would be $(49 + 48 + 47 + 46 + 45) + 1 = 236$, which is about 40× as many as for a prefix field.

Storage, insert, and find complexity. As shown in the example above, substring search can generate a non-trivial number of tags. As previously stated, the number of tags has a direct impact on both the overall storage overhead and the insertion time. More precisely, the time complexity of inserting a value v is

$$\mathsf{time}_{\mathsf{insert}} = O\Big(t(v) \cdot \log\Big(\mathsf{metadocs}(\mathtt{esc})\Big)\Big), \tag{8}$$

whereas the storage cost of inserting v is

$$O(|v| + t(v)). \tag{9}$$

For a precise estimate of the storage overhead, in bytes, incurred by prefix, suffix, or substring search, we refer the reader to Appendix A. Both quantities are clearly linear in the tag count. The find-time complexity, however, is not substantially affected by the tag count, though, as we discuss below, other factors come into play. More precisely, the find-time complexity is the same for prefix, suffix, and substring search and, in fact, matches that of equality search; namely,

$$\mathsf{time}_{\mathtt{find}} = O\Big(\mathsf{cf}_f \cdot \log\Big(\mathsf{metadocs}(\mathtt{esc})\Big) + \#\mathbf{R}\Big),$$

where cf_f is the contention factor of the corresponding field. We can make two main observations. First, the number of documents in `esc`, which is hidden in the asymptotic expression above, is itself a function of the total number of tags across all insertions. This dependency is only logarithmic, but it can still have practical implications. The second observation is more subtle but important: the number of responses $\#\mathbf{R}$ matching a query can grow as the lower bound decreases (this holds for prefix and suffix search, but is more pronounced for substring search).

In particular, for a small lower bound (say 2), we need to store all substrings of length 2, and 2-character substrings are far more likely to coincide across distinct values than longer substrings. For example, `Engineer` and `Encode` share the common prefix `En` when the lower bound is set to 2, but not if the lower bound is set higher.

11.2 Setting the Parameters

As illustrated above, the parameters of a string field have a direct impact on storage, insertion, and find complexity. In particular, as shown in Equations 8 and 9, both insertion time and storage are linear in the tag count. This means that the smaller the tag count, the faster the insertion and the more compact the encrypted structure. In addition, as illustrated in Equations 5, 6, and 7, under the same parameters the tag count for substring search is significantly higher than for prefix, suffix, or prefix-and-suffix search. Given these two facts, we propose the following simple recommendations:

- *Tightness*: for prefix, suffix, and substring search alike, we recommend setting the parameters as tightly as possible. For example, in the substring case, if all queries an application makes have 4 characters, then setting $\mathsf{lb} = 2$ would be wasteful; setting $\mathsf{lb} = 4$ would be more appropriate.
- *Choice of expressiveness*: as highlighted above, substring search incurs the highest overhead among all string search operations. It is therefore important to use it only if the application really needs to match strings in the middle of a value. If the application only queries for prefixes or suffixes, enabling substring search on the field is wasteful.

Setting the parameters independently of the data. As with the trimming factor of Section 10.3, the string parameters should never be chosen as a function of the values stored in the collection. Recall that the parameters are public, as they are stored in the schema on the server. They should therefore not reveal anything about the data. For example, setting lb so as to reduce the number of matching results (say, to improve the find-time complexity) would reveal to the server that many values in the collection share common substrings (or prefixes, in the prefix search case). Therefore, all parameters should be chosen to reflect the application's query patterns.

11.3 Scheduling Compactions for String Indexes

Our recommendation for scheduling compactions is the same as the one presented in Section 9.1, where we suggest compacting less frequently so that the storage savings per compaction are higher. The security discussion there also carries over to the string case. There are, however, two differences between the equality and string cases. First, in the equality case every insertion grows the `esc` by one meta document, whereas in the string case every insertion grows the `esc` by $t(v)$ meta documents, and, as shown in the concrete example above, this number can be significant. This implies that, in the string case, the `esc` can grow at a significantly higher rate than in the equality case, so a compaction will reclaim a substantial amount of storage. Second, as we explain below, the compaction of a string field introduces additional dummy elements.

The addition of these dummies reduces the reclaimed storage, but it is necessary to maintain the same leakage profile as for equality and range. We can write the reclaim $\mathsf{reclaim}_i$ as follows. Between the $(i-1)$th and the ith compaction, let $\mathbf{V}_i$ denote the multiset of values inserted into a string field and $\mathbf{U}_i$ the set of unique values among them, so that $\#\mathbf{V}_i = \mathsf{docinserts}_i$ and $\#\mathbf{U}_i = \mathsf{valinserts}_i$. Then the ith compaction shrinks the `esc` by exactly the reclaim

$$\mathsf{reclaim}_i = \sum_{v \in \mathbf{V}_i} t(v) - \sum_{v \in \mathbf{U}_i} t(v), \tag{10}$$

which reduces to Equation (1) when $t(v) = 1$; for a collection with several encrypted fields, the reclaims add across the fields. The reclaim above can be rewritten as

$$\sum_{v \in \mathbf{U}_i} (\mathsf{freq}_i(v) - 1) \cdot t(v),$$

where $\mathsf{freq}_i(v)$ is the number of times v was inserted since the last compaction. As in the equality and range cases, the storage that compaction reclaims comes entirely from *repeated values*. In particular, two distinct values that share many substrings save nothing: since they are distinct, each still contributes $t(v)$ meta documents to the `esc` after compaction. The only savings come from values that have been inserted more than once.

Concrete example. We revisit the example of Section 9.1: a collection of 1,000 documents whose single encrypted field `name` holds `Alice` in 500 documents, `Bob` in 250, and `Eve` in 250, now indexed for substring queries with $\mathsf{lb} = 2$, $\mathsf{ub} = 6$, and $\mathtt{mlen} = 50$. All three names have a padded length of 11 bytes, so every value has a tag count of 41. Prior to compaction, the `esc` and the `ecoc` each hold $1{,}000 \cdot 41 = 41{,}000$ meta documents. After compaction, the `ecoc` is empty and the `esc` holds $3 \cdot 41 = 123$ meta documents, a storage saving of around 99.7%. Counting the three names' distinct substrings would suggest only $10 + 3 + 3 = 16$ meta documents, but this is not what happens: compaction preserves the individual tag count of each distinct value, independently of any overlap between the values.

Security. A natural question is why we keep the dummy meta documents in the `esc` collection rather than removing them during compaction. The reason is security. To see why, recall that the compaction leakage of `QE` consists of the number of unique values inserted since the last compaction, which we denote by $\mathsf{valinserts}_i$. For the string case, we maintain the same leakage: here too, our design principle is to reveal only the number of unique values inserted since the last compaction, which is captured by the observed quantity $\sum_{v \in \mathbf{U}_i} t(v)$. Note that $t(v)$, for all $v \in \mathbf{U}_i$, is known and is a function of only the public parameters and the length of the padded value, which is already leaked as part of the insertion. The only information compaction reveals is $\mathsf{valinserts}_i$, and string search therefore achieves parity with all other types of search operations in `QE`. While parity is important for consistency, it is especially important in the string case. To see why, consider what would happen if compaction instead shrank the `esc` to one meta document per distinct substring, which is the smaller size one might expect. That count depends on how much the inserted values overlap, so the adversary could distinguish, for example, a batch of unrelated names from a batch of names sharing a common stem, and more generally could track how similar the collection's values are to one another.

11.4 Additional Security Considerations

Contention factor. The contention factor guidance of Section 9.2 is stated per field/value pair. On a string field, concurrent insertions contend when their values share a substring, and short substrings are shared by many values; the trigram `ing`, for example, occurs in a large fraction of English words. Contention is therefore more likely than on an equality field with the same workload. Nevertheless, we recommend keeping the default contention factor and, as with equality fields, never setting it as a function of the stored values or their frequencies. To change the contention factor, we recommend an approach similar to the one discussed in Section 9.2. In particular, suppose ω_f is an estimate of the total number of concurrent insertions or updates on a given field f within a small interval of time, say 30ms. The contention factor cf_f can then be set as

$$\mathsf{cf}_f = \left\lceil \frac{\omega_f^* \cdot (\omega_f^* - 1)}{0.2} \right\rceil,$$

where $\omega_f^* = \lceil \omega_f / \mathsf{possiblevals}_{\mathsf{lb}} \rceil$ and $\mathsf{possiblevals}_{\mathsf{lb}}$ is the number of possible strings of length lb. If the number of concurrent insertions/updates on a field is not known, ω_f can be set to the server's number of virtual cores. Note that the above formula is designed to bound the probability of contention to at most 0.1, but it can be tuned to achieve different bounds. As an example, if $\omega_f = 100$, $\mathsf{lb} = 2$, and the characters are numeric (e.g., an `ssn` field), then the number of possible 2-character values is $\mathsf{possiblevals}_{\mathsf{lb}} = 10^{\mathsf{lb}} = 100$, so $\omega_f^* = 1$ and setting $\mathsf{cf}_f = 1$ is enough to avoid contention.

Values with sensitive lengths. Section 9 describes two application-side approaches for fields whose value lengths are sensitive: padding every value to a maximum length and mapping the values to fixed-length numbers. In the string case, only the first approach applies: mapping values to numbers does not preserve the substring structure needed for string search. Padding works because the number of tags revealed during an insertion is a function of the padded length of the value.

Additional guidelines. All the guidelines related to the aggregation pipeline (Section 9.4), partial document encryption, schema, and audit logs remain the same, and we refer the reader to Section 9.5 for more details.

12 The Research Program Behind Queryable Encryption

When we started working on `QE`, we quickly realized that there was a non-trivial gap between the state of the art in research and the problems that needed to be solved to produce a real-world system. Closing this gap motivated a series of research directions that we have been pursuing to support `QE`. Some of these directions, such as the design of encrypted data structures and the study of leakage, predate `QE`. Others, such as cryptographic services, concurrent encrypted structures and the security of TEE-based encrypted databases, were not, and grew directly out of the problems we encountered while building `QE`. In this section, we give an overview of these directions.

We note that not everything described in this section is used in QE, at least not yet. Moving from research to product takes time, and not every research result can be used directly, for a variety of reasons. A result may depend on assumptions that do not hold in a production database, it may need further work to be made efficient enough at scale, or it may need to wait for the right point in the product's evolution. The research described here therefore includes both work that underlies the current system and work that lays the foundation for its future evolution.

Cryptographic services. QE is not a one-shot protocol but a long-lived, stateful system that supports multiple operations and serves many clients concurrently. The same is true of many modern cryptographic deployments, including key management services, cryptographic wallets and identity management systems. General frameworks such as universal composability and its variants can model these systems, including their concurrent execution, but they are not designed around the features that make them distinctive: persistent state shared across many operations, per-operation costs that must be sublinear in the size of that state, and security guarantees that are expressed in terms of leakage. As a result, analyses of such systems tend to rely on special-purpose definitions that are hard to reuse or compare. We are developing a framework of *cryptographic services* that is tailored to systems of this form, treats leakage as a first-class part of the security definition, and captures the setting in which sessions are concurrently and adversarially interleaved. This provides a foundation for analyzing QE as a whole rather than one operation at a time.

Encrypted data structures. The cryptographic core of QE is a scheme called OST which was introduced and analyzed in [37]. We are continuing to extend OST to support richer query types while maintaining provable security guarantees. Range queries, for example, are handled using range hypergraphs [40] and, as of release 9.0, QE also supports prefix, suffix and substring queries. We are currently working on a series of papers on OST that cover how it handles equality, range and string search queries, and that will include a complete rewrite of the original OST paper [37].

Building QE also raised questions about encrypted structures that prior work had not addressed. The first is concurrency. Modern database servers execute many operations simultaneously but, until recently, encrypted data structures had only been studied in a sequential setting. Concurrency raises new questions, both about consistency and about leakage, since what the server learns can depend on how operations are interleaved. We initiated the study of concurrent encrypted structures in [6], and followed up with constructions that trade the strong consistency guarantee of that work for a weaker but still natural one [5].

The second is security against injection attacks, in which an adversary inserts data of its choice and observes how queries interact with it [57]. Most dynamic encrypted structures aim for forward privacy [54, 15], which was motivated by the goal of protecting against these attacks. In [9], we showed that forward privacy does not by itself provide this protection, introduced a stronger notion of injection security, and designed the first injection-secure schemes. More recent work [7] makes these schemes substantially more efficient.

Leakage suppression. Leakage suppression studies how to design encrypted structures that reveal less. A *leakage suppressor* is a compiler that takes an encrypted structure and produces a new one with a better leakage profile. Our work in this area has focused on the query equality pattern, which reveals whether and when two queries are the same. It is the most common leakage pattern in the persistent setting and one of the most costly to hide. We initiated the area with general suppressors for static [38] and then dynamic [27] encrypted structures. These are asymptotically efficient but not yet practical, so more recent work has focused on closing that gap: by exploiting the distribution of queries [26], and by developing a general framework for suppression that explains existing constructions and guides the design of new ones [34]. The long-term goal of this line of work is to give `QE` users the option of hiding query equality at a practical cost in the persistent setting.

Leakage analysis. Leakage profiles describe *what* an encrypted database solution leaks, but not *how much* an adversary can infer from that leakage. Bayesian Leakage Analysis (BLA) [21] is a statistical framework we developed that bounds what an adversary can infer from a leakage profile given its auxiliary information. BLA gives us a principled way to compare leakage profiles and to reason about the real-world impact of leakage beyond the cryptanalysis of specific attacks. While we have not yet applied BLA to `QE` specifically, it has substantially improved our understanding of leakage and leakage attacks, and that understanding informs all of our work. PolySys [23] is an algebraic framework we developed that automatically generates leakage attacks. Rather than designing attacks by hand, PolySys lets us systematically explore what can be recovered from a given leakage profile, which makes it a tool for stress-testing new designs before they are deployed. We used PolySys to cryptanalyze the leakage of OST in the persistent model, and the results formed the basis of our recommendation not to use range or substring queries when persistent adversaries are a concern.

Security of TEE-based encrypted databases. We are interested in, and study, all encrypted database technologies, not just the ones that `QE` is currently built on. As we studied TEE-based encrypted databases, however, we realized that, even though they are deployed in real-world products, their security had not been studied as rigorously as that of cryptographic approaches. This led us to start a new line of work on their security. Leafblower [22] is the first leakage attack against TEE-based encrypted databases and recovers the order of inserted values from snapshots of disk. Dendrochronology [20] extracts the frequencies of stored values from TEE-based encrypted databases whose leakage profiles were claimed to reveal only their order. These results show that TEE-based designs are subject to leakage that must be analyzed as carefully as that of cryptographic designs.

Performance evaluation. Benchmarking is central to how databases are designed, tuned and compared, and the database community has invested heavily in standardized benchmarks such as TPC-C and TPC-H [1] and YCSB [18]. These benchmarks specify the datasets, workloads, metrics and procedures used to evaluate a system, so that results are reproducible and comparable across systems. They are not, however, suited to encrypted databases. This is because the performance of an encrypted database can depend on properties that traditional benchmarks do not capture, such as the history of operations performed on the data rather than only its

current contents, and the parameters of the underlying cryptographic schemes, which trade efficiency against leakage. To our knowledge, there are no standardized benchmarks for encrypted databases, and the question of how to benchmark them has received little attention in the research literature. We therefore had to develop our own approach and decide what an encrypted database benchmark should measure, design datasets that control for the properties that drive performance, design evaluation methods, and run the resulting benchmarks for each of QE's query types. The results of these benchmarks informed both the implementation of QE and the default values of its parameters.

Acknowledgements

The QE team would like to thank David Cash for feedback on the design and analysis of OST and Trail of Bits for their feedback on the implementation of QE. We would also like to thank Zichen Gui, Kenneth Paterson, Tianxin Tang and the MongoDB customers who provided feedback on QE's preview release.

References

[1] Tpc benchmarks. http://www.tpc.org.

[2] MongoDB Built-In Roles, June 2023. https://www.mongodb.com/docs/manual/reference/built-in-roles/.

[3] MongoDB Server Parameters, June 2023. https://www.mongodb.com/docs/manual/reference/parameters/.

[4] R. Ada Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[5] A. Agarwal and Z. Espiritu. Sequentially consistent concurrent encrypted multimaps. In *IEEE European Symposium on Security and Privacy (EuroS&P '25)*, 2025.

[6] A. Agarwal, S. Kamara, and T. Moataz. Concurrent encrypted multimaps. In *Advances in Cryptology - ASIACRYPT 2024*, 2024.

[7] A. Agarwal, S. Kamara, and T. Moataz. Injection-secure encrypted multi-maps. In *Advances in Cryptology - ASIACRYPT 2026*, 2026.

[8] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2004.

[9] G. Amjad, S. Kamara, and T. Moataz. Injection-secure structured and searchable symmetric encryption. In *Advances in Cryptology - ASIACRYPT 2023*, pages 232–262, 2023.

[10] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[11] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.*, 26(3):752–765, 2014.

[12] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[13] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

[14] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[15] R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.

[16] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[17] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, 2010.

[19] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data*, pages 185–198. ACM, 2016.

[20] Z. Espiritu. Extracting frequencies from TEE-based encrypted databases via dendrochronology. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, 2026. To appear.

[21] Z. Espiritu, S. Kamara, and T. Moataz. Bayesian leakage analysis: A framework for analyzing leakage in cryptography. *IACR Communications in Cryptology*, 2(1), 2025.

[22] Z. Espiritu, S. Kamara, T. Moataz, and V. Ogier. Leafblower: a leakage attack against TEE-based encrypted databases. In *IEEE Symposium on Security and Privacy (S&P '26)*, 2026.

[23] Z. Espiritu, S. Kamara, T. Moataz, and A. Park. Polysys: an algebraic leakage attack engine. In L. Bauer and G. Pellegrino, editors, *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 3357–3376. USENIX Association, 2025.

[24] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science*, volume 9327, pages 123–145, 2015.

[25] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: a scalable private dbms. In *IEEE Symposium on Security and Privacy*, pages 395–410. IEEE, 2015.

[26] M. George, S. Kamara, T. Moataz, and Z. Espiritu. Structured encryption and distribution-aware leakage suppression. In *Asiacrypt '25*, 2025.

[27] M. George, S. Kamra, and T. Moataz. Structured encryption and dynamic leakage suppression. In *Advances in Cryptology - EUROCRYPT 2021*, 2021.

[28] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *Workshop on Hot Topics in Operating Systems (HotOS '17)*, pages 162–168, New York, NY, USA, 2017. ACM.

[29] IBM. Db2 native encryption. https://www.ibm.com/docs/en/db2/11.5?topic=rest-db2-native-encryption.

[30] M. Jurado, C. Palamidessi, and G. Smith. A formal information-theoretic leakage analysis of order-revealing encryption. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16, 2021.

[31] M. Jurado and G. Smith. Quantifying information leakage of deterministic encryption. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, pages 129–139, New York, NY, USA, 2019. Association for Computing Machinery.

[32] S. Kamara, A. Kati, J. D. Maria, T. Moataz, A. Park, and A. Treiber. MAPLE: MArkov Process Leakage attacks on Encrypted Search. Technical Report 2023/810, IACR ePrint Cryptography Archive, 2023. https://eprint.iacr.org/2023/810.

[33] S. Kamara, A. Kati, T. Moataz, T. Schneider, A. Treiber, and M. Yonli. Sok: Cryptanalysis of encrypted search with leaker – a framework for leakage attack evaluation on real-world data. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 90–108, 2022.

[34] S. Kamara, K. Kwegyir-Aggrey, and T. Moataz. Query-equality suppression for encrypted multi-maps via kernel emulation. In preparation, 2026.

[35] S. Kamara and T. Moataz. SQL on structurally-encrypted databases. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part I 24*, pages 149–180. Springer, 2018.

[36] S. Kamara and T. Moataz. Bayesian leakage analysis: A framework for analyzing leakage in encrypted search. Technical Report 2023/813, IACR ePrint Cryptography Archive, 2023. https://eprint.iacr.org/2023/813.

[37] S. Kamara and T. Moataz. Design and analysis of a stateless encrypted document database, 2023. https://www.mongodb.com/collateral/stateless-document-database-encryption-scheme.

[38] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakae suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[39] S. Kamara, T. Moataz, S. Zdonik, and Z. Zhao. Opx: An optimal relational database encryption scheme. Technical report, IACR ePrint Cryptography Archive, 2020.

[40] K. Kongsala, S. Kamara, and T. Moataz. Encrypted range multi-maps via range hypergraphs. Manuscript, 2026.

[41] E. M. Kornaropoulos, N. Moyer, C. Papamanthou, and A. Psomas. Leakage inversion. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, nov 2022.

[42] Microsoft. Transparent data encryption (tde). https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver16.

[43] MongoDB. Bson types. https://www.mongodb.com/docs/v5.0/reference/bson-types/.

[44] MongoDB. Built-in roles. https://www.mongodb.com/docs/manual/reference/built-in-roles/.

[45] MongoDB. Config database. https://www.mongodb.com/docs/manual/reference/config-database/.

[46] MongoDB. Database profiler. https://www.mongodb.com/docs/manual/tutorial/manage-the-database-profiler/.

[47] MongoDB. Log messages. https://www.mongodb.com/docs/manual/reference/log-messages/.

[48] MongoDB. Queryable encryption, 2023. https://www.mongodb.com/docs/manual/core/queryable-encryption/.

[49] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 644–655. ACM, 2015.

[50] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[51] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[52] SAP Software Solutions. SEEED. https://www.sics.se/sites/default/files/pub/andreasschaad.pdf.

[53] I. Security. Cost of a data breach report, 2022. https://www.ibm.com/downloads/cas/3R8N1DZJ.

[54] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[55] D. Vinayagamurthy, A. Gribov, and S. Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *PoPETs*, 2019(3):370–388, 2019.

[56] C. V. Wright and D. Pouliot. Early detection and analysis of leakage abuse vulnerabilities. *IACR Cryptol. ePrint Arch.*, page 1052, 2017.

[57] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, 2016.

[58] Z. Zhao, S. Kamara, T. Moataz, and S. Zdonik. Encrypted databases: From theory to systems. In *Conference on Innovative Data Systems Research (CIDR '21)*, 2021.

A Storage Estimates for Queryable Encryption

In this section, we describe a framework for estimating the concrete storage cost of `QE` in bytes. The main factor controlling storage is the *tag count* $t(v)$ of an inserted value v, which depends on the query types enabled on the field and on their parameters (see Section 11.1). The framework expresses storage as a function of $t(v)$ alone, so the same estimates apply to every `QE` query type (i.e., equality, range, prefix, suffix and substring) and only the tag count differs from one type to another. We first estimate the storage of a single encrypted field and then show how to estimate the storage required for N documents given a schema S.

A.1 Storage Cost per Collection

`QE` involves three collections: the encrypted data collection (`edc`), the encrypted state collection (`esc`), and the encrypted compaction collection (`ecoc`). In this section, we study the case of a single encrypted field per document; the analysis generalizes straightforwardly to multiple fields, as we detail in Section A.4.

Below, we give the storage cost attributable to the $t(v)$ tags in each of the three collections, for some inserted value v.

A.1.1 Encrypted Data Collection

Size of an `edc` document. The `edc` stores one encrypted document per inserted document in the collection. An encrypted document consists of an encrypted payload for each `QE`-indexed field and a `safeContent` array that holds a set of cryptographic tags. The size of an encrypted document is therefore determined by the following components:

- **encrypted value:** the value itself is encrypted with AES-CBC and authenticated with an HMAC. This costs 16 bytes for the IV, 32 bytes for the HMAC, and $\left\lceil \frac{|v|+6}{16} \right\rceil \cdot 16$ bytes for the PKCS#7-padded ciphertext. An encrypted value therefore has size

$$48 + \left\lceil \frac{|v|+6}{16} \right\rceil \cdot 16;$$

- **re-encrypted value:** a re-encrypted value consists of an AES-CTR re-encryption of the encrypted value together with a 16-byte key ID, which adds another 32 bytes (16 bytes for the CTR IV and 16 bytes for the key ID). A re-encrypted value therefore has size

$$80 + \left\lceil \frac{|v|+6}{16} \right\rceil \cdot 16;$$

- **payload:** a payload consists of a re-encrypted value, 1 byte for the FLE payload type, 16 bytes for the key ID, 1 byte for the original BSON type, 12 bytes for the prefix/suffix/substring counts, and 96 bytes of metadata per tag. A payload therefore has size

$$110 + \left\lceil \frac{|v|+6}{16} \right\rceil \cdot 16 + 96 \cdot t(v);$$

- **safeContent:** the `safeContent` array stores the tags, each of which is 32 bytes. The `safeContent` array therefore has size $32 \cdot t(v)$ bytes;

- **id field:** the document has an `_id` field, which has size 12 bytes.

The total `edc` storage in bytes per document attributable to $t(v)$ tags is therefore

$$128 \cdot t(v) + 122 + \left\lceil \frac{|v|+6}{16} \right\rceil \cdot 16.$$

Size of `edc` indexes. The following indexes are built on the `edc`:

- **safeContent field index:** its size is close to that of the `safeContent` array and is therefore approximately $32 \cdot t(v)$ bytes;
- **id field index:** its size is close to that of the `_id` value and is therefore approximately 12 bytes.

The total `edc` index storage in bytes per document attributable to $t(v)$ tags is therefore

$$32 \cdot t(v) + 12.$$

A.1.2 Encrypted State Collection

For simplicity, we assume that no compaction has been run, since compaction slightly complicates the calculation. Each tag results in one meta document in the `esc`, which requires 32 bytes of payload. The total `esc` storage in bytes per document attributable to $t(v)$ tags is therefore

$$32 \cdot t(v).$$

A.1.3 Encrypted Compaction Collection

The `ecoc` stores the compaction tokens used during compaction. It stores one encrypted compaction token per tag, and each encrypted token is represented as a document in the `ecoc` with the following three fields:

- **id field:** an `_id` that is 12 bytes;
- **name field:** the name of the tag's field, which we assume is 32 bytes;
- **value field:** the encrypted token, which consists of a 16-byte IV, a 32-byte encrypted token, and a 3-byte counter. The value in this field is therefore 51 bytes.

Under the 32-byte field-name assumption, the total `ecoc` storage in bytes attributable to $t(v)$ tags is

$$95 \cdot t(v).$$

Note that compaction empties the `ecoc`, so this term represents the pre-compaction state.

A.2 Encoding Overhead

Because these collections are stored using BSON encoding, each field/value pair incurs additional bytes (beyond the field/value itself) for encoding information such as element type bytes (e.g., string, int), field names, and so on. Empirically, we observed this overhead to be less than 1.2×. Additionally, the overhead of the `_id` field index and the `safeContent` index is at most 64× and 2.5×, respectively, based on empirical measurements of on-disk storage. In memory, the overhead of the `_id` field index and the `safeContent` index is significantly smaller and at most 1.42× and 1.63×, respectively, again based on empirical measurements. This encoding overhead stems from plaintext encoding, which applies equally to unencrypted collections; it is therefore not specific to `QE`.

A.3 Total Storage

We now state the storage estimate in bytes for a single document insertion, which results in multiple documents across the `QE` collections as described above, together with the storage estimate in bytes for the resulting indexes.

A.3.1 Total Size of Documents

The total document storage estimate in bytes across all three collections, $\mathrm{T}_{\mathrm{doc}}$, is bounded by

$$\mathrm{T}_{\mathrm{doc}} \leq 1.2 \cdot \left(255 \cdot t(v) + 122 + \left\lceil \tfrac{|v|+6}{16} \right\rceil \cdot 16\right). \tag{11}$$

A.3.2 Total Size of Indexes

The total on-disk index storage estimate in bytes across all three collections, $\mathrm{T}_{\mathrm{index}}$, is bounded by

$$\mathrm{T}_{\mathrm{index}} \leq 2.5 \cdot 32 \cdot t(v) + 64 \cdot 12 \leq 1.2 \cdot (67 \cdot t(v) + 640). \tag{12}$$

The total storage per single document insertion, T, accounting for all documents across the `QE` collections as well as the indexes, is therefore bounded by

$$\mathrm{T} \leq 1.2 \cdot \left(322 \cdot t(v) + 762 + \left\lceil \tfrac{|v|+6}{16} \right\rceil \cdot 16\right).$$

In-memory index storage. The in-memory index storage in bytes, $\text{M-T}_{\mathrm{index}}$, is significantly smaller and is bounded by

$$\text{M-T}_{\mathrm{index}} \leq 1.63 \cdot 32 \cdot t(v) + 1.42 \cdot 12 \approx 52 \cdot t(v) + 17. \tag{13}$$

A.4 Calculating the Storage Overhead for a Schema

Given a schema S with m encrypted fields and a fixed number of documents N, we can use Equations 11 and 12 to estimate the storage required for N documents with the specified schema.

To do so, we compute the tag count $t(v_i)$ for the ith field, where v_i is the value inserted in that field, and use it to obtain the per-document storage estimate for that field (excluding indexes).

We then sum these estimates over all fields, add the index storage, and multiply by N to obtain the final storage estimate $\mathrm{T_{S,N}}$. More precisely,

$$\mathrm{T_{S,N}} \leq 1.2 \cdot N \cdot \Big(640 + 67 \cdot \sum_{i=1}^{m} t(v_i) + \sum_{i=1}^{m} \Big(255 \cdot t(v_i) + 122 + \Big\lceil \tfrac{|v|+6}{16} \Big\rceil \cdot 16\Big)\Big).$$

In-memory index storage. Based on Equation 13, the total in-memory index storage in bytes, $\mathrm{M\text{-}T_{S,N,index}}$, can be calculated as

$$\mathrm{M\text{-}T_{S,N,index}} \leq N \cdot \Big(17 + 52 \cdot \sum_{i=1}^{m} t(v_i)\Big).$$