An Overview of Queryable Encryption

October 29, 2023

Contents

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. 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 [30].* 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 [35], 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`) [14], SQL Server's `Transparent Data Encryption` (`TDE`) [24] 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` [3], `SEEED` [34] and `CipherBase` [5]. These systems work by encrypting database columns/fields using property-preserving encryption schemes like deterministic encryption [7] and order-revealing encryption [4, 9]. 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` [11, 10], `Blind Seer` [32, 12] and `KafeDB` [19, 22, 38]. These systems 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.

[1] We do not consider enclave-based databases like `Always Encrypted with secure enclaves` and earlier research systems like `TrustedDB` [6], `EnclaveDB` [33] and `StealthDB` [36] since the security of such systems has not been formalized clearly.

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.

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 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`.

[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.

Note this 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. QE' s performance should be within 10× of plaintext queries.

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.

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 [35], 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.

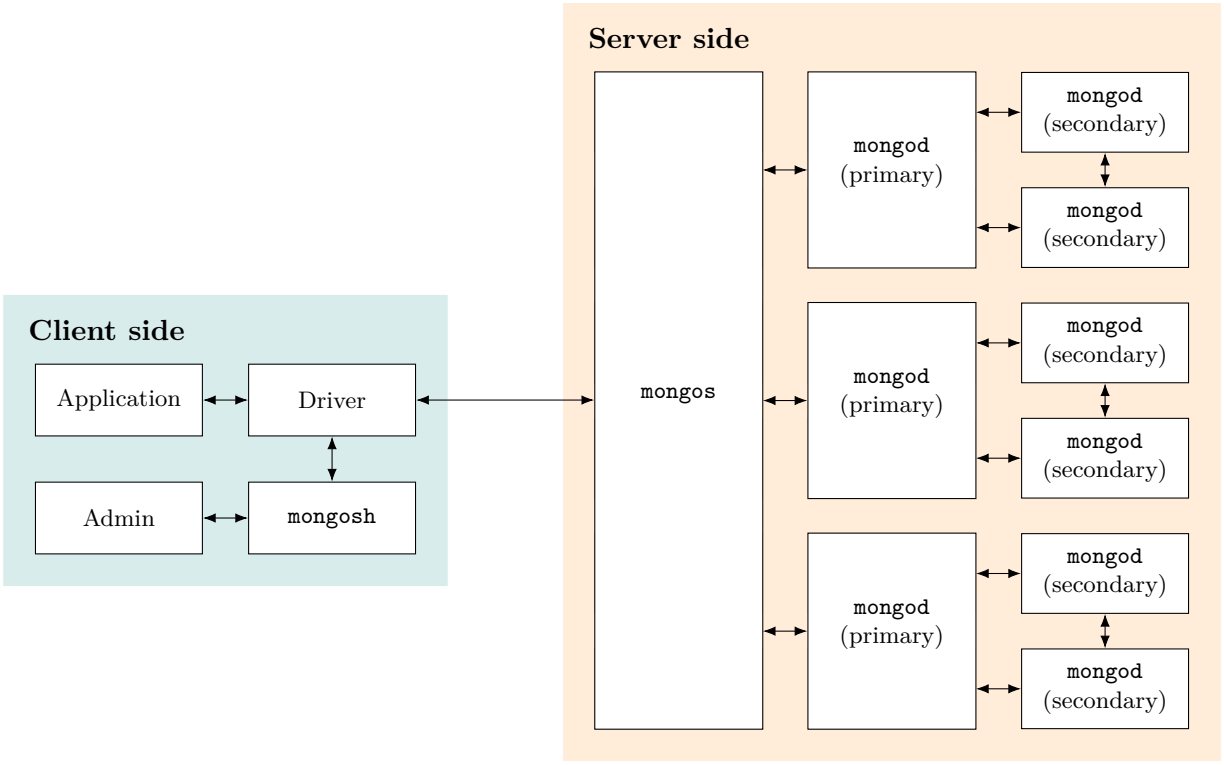

Figure 1: The architecture of a MongoDB cluster.

2. `mongos`: MongoDB is designed to scale horizontally by running on large clusters. This is achieved using shardingwhich 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 encryption scheme called OST which is described and analyzed in [21]. 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.
- the *encrypted client*: applications make use of `QE` through encrypted clients that are responsible for, among other things, retrieving encrypted DEKs from the key vault, translating plaintext insert, find, update and delete operations to their encrypted versions.
- `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 `equality` in which case OST is used. 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 [21]. `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 edc which stores the encrypted documents, the esc which stores an auxiliary encrypted structure needed for search, and the 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 [21]. In the rest of this document, we refer to a triple (edc, esc, ecoc) as an encrypted database edb.

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

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

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 cihertext 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 [37, 16, 15, 23, 20]. 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 [31];

2. the state-of-the-art known-data and injection attacks against the volume pattern [8];

3. the state-of-the-art inference attacks against the query equality pattern [17];

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 [18].

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 [35], *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

[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.

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:

   (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. We are currently working on new tools and methodologies which we plan to outline in the near future.

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 [13], 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.

Advesarial 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 *ibmne*, `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 [3, 34, 5] 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] 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`.

5. `updateOne` operations reveal the shape of the update and when the udpated 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 [21], where it is shown to 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;
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.

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 [21]) 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] 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
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
	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 udpated 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 and comptime reveals when a compaction occurs. 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 [21].

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 $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$. DBMSs, however, log all sorts of information about queries so, as pointed out in [13], 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.

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

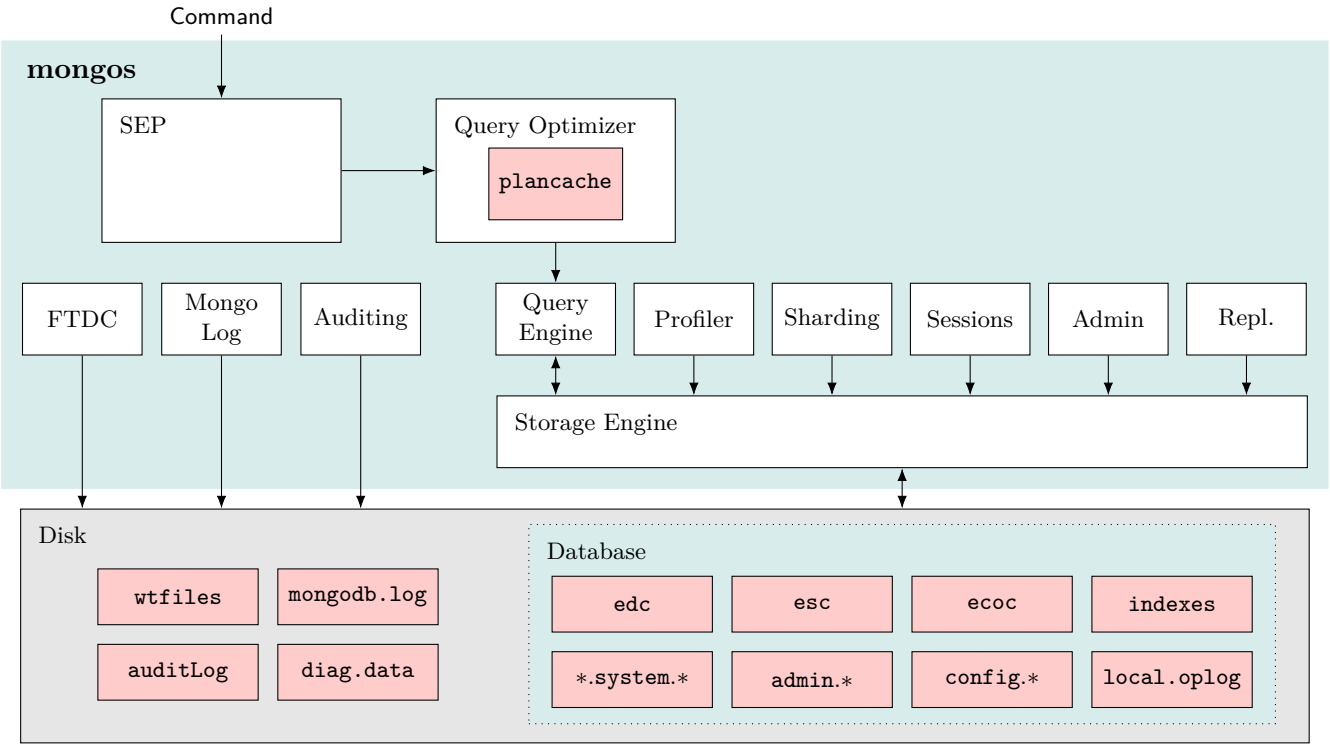

Figure 2: The `mongod` logging architecture.

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 [29] 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 [28] 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 [27] 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;

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

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. `$renameCollectionSameDB`: allows the user to change the name of an existing collection on the current database;
7. `serverStatus`: returns statistics gathered by the FTDC service, including information related to CPU, disk and memory usage. This includes read and write latencies, statistics on primary node elections, locking information and the number of bits spilled to disk. Note that this data is accumulated across a single machine and does not pertain to a specific database or collection. `serverStatus` can, however, be repeatedly executed to gather finer-grained statistics. These statistics will still be across a single machine but could be at the granularity of a second.
8. `setParameter`: sets a variety of server parameters. For a full list, see [2];
9. `$allCollectionStats`: returns statistics over all the collections;
10. `$operationMetrics`: returns various internal billing metrics for Atlas serverless such as the number of documents read, the number of index entries read or the number of cursor seeks;
11. `$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.

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.

[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.

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.

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 [1].

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:

- *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 [26].

- *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 $\texttt{edb} = (\texttt{edc}, \texttt{esc}, \texttt{ecoc})$. In addition, none of `read`'s authorized operations return `QE`-related data. Therefore, `read`'s view is that of an $(\texttt{edb}, [n])$-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 $*.\texttt{system.js}$ collections. Note that when creating a `readWrite` role it is possible to additionally grant it access to the $\texttt{config}.*$, $\texttt{local}.*$ and $\texttt{admin}.*$ databases. Here, we assume that the `readWrite` is *not* authorized to access the $\texttt{local}.*$ database. The surface also includes the output of its authorized commands like `$collStats` and `$planCacheStats`. For the full list of commands, see [26].
- *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 $\texttt{edb} = (\texttt{edc}, \texttt{esc}, \texttt{ecoc})$. In addition, none of `readWrite`'s authorized operations return `QE`-related data. Therefore, `readWrite`'s view is that of an $(\texttt{edb}, [n])$-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 $*.\texttt{system.profile}$ 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 [26].
- *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 $\texttt{edb} = (\texttt{edc}, \texttt{esc}, \texttt{ecoc})$. 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 $(\texttt{edb}, [n])$-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`. [9] For the full list of commands, see [26].
- *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 $\mathsf{edb} = (\mathtt{edc}, \mathtt{esc}, \mathtt{ecoc})$ 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". In followup work, we will provide a more precise analysis of this role.

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 [26].
- *schedule*: we assume `clusterMonitor` can access its surface after every operation;

[9] Note that `clusterManager` can run the `$setFreeMonitoring` command which allows it to observe `mongodb.log`, slow queries and more. We do not include this in our analysis because it was deprecated in April 2023 and will be decommissioned in August 2023.

- *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 [26].

- *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 [26].

- *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 Guidelines

In this section, we describe a list of guidelines for setting various `QE` parameters as well as discuss several security considerations.

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. 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

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

where $\mathsf{metadocs}(\mathtt{esc})$ is the number of meta documents in the `esc`, compactions is the number of compactions executed so far and 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,

$$\mathsf{metadocs}(\mathtt{esc}_{i+1}) = \mathsf{metadocs}(\mathtt{esc}_i) - \mathsf{encfields} \cdot \mathsf{docinserts}_i + \mathsf{valinserts}_i,$$

where $\mathsf{metadocs}(\mathtt{esc}_{i+1})$ is the number of meta documents in the `esc` after the ith compaction, $\mathsf{metadocs}(\mathtt{esc}_i)$ the number of meta documents in the `esc` before the ith compaction, $\mathsf{docinserts}_i$ 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 between the $(i-1)$th and the ith compaction.[10] It follows that for a compaction operation to reduce the size of the `esc` by at least t meta documents, it should be executed whenever

$$\mathsf{encfields} \cdot \mathsf{docinserts}_i - \mathsf{valinserts}_i \geq t. \tag{1}$$

So, a possible strategy to schedule compactions is to keep track locally of the values $\mathsf{encfields}$, $\mathsf{docinserts}_i$ and $\mathsf{valinserts}_i$ and to compact whenever Equation (1) is satisfied.

Security. As discussed in Section 6, compactions reveal $\mathsf{valinserts}_i$, the number of unique `QE`-encrypted field/value pairs since the last compaction.[11] 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 $n_i \stackrel{\circ}{=} \mathsf{docinserts}_i$, $m_i \stackrel{\circ}{=} \mathsf{valinserts}_i$, $m \stackrel{\circ}{=} \mathsf{possiblevals}$ 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. However, it does suggest that, when $\mathsf{docinserts}_i$ is larger, the less information is revealed.

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

[10] 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.

[11] 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.

`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 $\omega = 100$, $\mathsf{valinserts} = 100$, 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 $\omega = 100$, $\mathsf{valinserts} = 50$, 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

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

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` [25].

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.

Acknowledgements

The QE team would like to thank David Cash for feedback on the design and analysis of OST and Jean-Philippe Aumasson 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] MongoDB Built-In Roles, June 2023. https://www.mongodb.com/docs/manual/reference/built-in-roles/.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

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

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

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

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

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

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

[31] 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.

[32] 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.

[33] 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.

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

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

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

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

[38] 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.