Security Analysis of MongoDB Queryable Encryption

Zichen Gui, Kenneth G. Paterson, and Tianxin Tang
Department of Computer Science, ETH Zurich, Zurich, Switzerland

Abstract

In June 2022, MongoDB released Queryable Encryption (QE), an extension of their flagship database product, enabling keyword searches to be performed over encrypted data. This is the first integration of such searchable encryption technology into a widely-used database system.

We provide an independent security analysis of QE. We show that certain logs, fundamental to the operation of QE and accessible to a real-world snapshot adversary, contain statistical information about the queries and data. This information can be extracted and exploited by our new inference attacks to recover both the queries and data, assuming adversarial access to an auxiliary dataset with a similar distribution to the original data.

Our analysis highlights the challenges of integrating searchable encryption technology into modern, complex database systems. In particular, our attacks stem from the interplay between QE and MongoDB's existing logging system. They show how such interactions can compromise query and data privacy.

1 Introduction

Encrypted Databases. Databases serve as the backbone of numerous applications. Cloud-hosted databases, in particular, have become increasingly popular due to their seamless deployment and high scalability. However, outsourcing client data to third-party cloud providers presents significant privacy risks.

To address these risks, we seek encryption methods for databases ensuring that, even when the encrypted database is outsourced to an untrusted server, the server is capable of efficiently answering client queries. Specifically, we aim for *query privacy* and *data privacy*. This means that the server learns neither the client's queries nor the contents of the database.

Searchable Symmetric Encryption / Structured Encryption. A long line of research has been conducted on searchable symmetric encryption (SSE) [5, 6, 8, 9, 13, 14, 26, 30, 31] and its generalization, structured encryption (STE) [10, 11]. These both focus on enhancing query and data privacy for outsourced data. A well-studied SSE/STE setting is searching on encrypted *documents* (or *records*), with each containing a set of *keywords*. In this setting, the most basic query type, *equality search*, returns all encrypted documents containing matching keywords in the queries. *Data privacy* pertains to the documents or records, while *query privacy* applies to the queried keywords.

MongoDB and QE. Despite two decades of research in SSE/STE, no SSE/STE schemes had been integrated into widely-used modern database systems. This landscape changed recently, when MongoDB introduced Queryable Encryption (QE) in their 6.0 official release for public preview. QE is an SSE/STE scheme that supports equality searches and has been integrated into MongoDB as a system component. QE operates on JSON-like documents, with each document containing a list of pairs of *field names* and *field values* (collectively referred to as *"fields"*). QE enables the client to specify which field names have their associated field values encrypted to support equality searches. Within QE, *field values* are treated similarly to *keywords* in SSE/STE.

As an SSE/STE scheme, QE aims to provide query and data privacy. MongoDB states that QE *"adds another layer of security for your most sensitive data, where data remains secure in-transit, at-rest, in memory, in logs, and in backups."*[1]

As the fifth-largest database vendor by market share, with around 13,000 business customers [12], MongoDB's pioneering effort to enhance database privacy through QE have far-reaching impact.

However, MongoDB has not, to date, provided a white paper or security proof to support the security claims.

[1] https://www.mongodb.com/products/queryable-encryption

This is cause for concern, as there is a rich body of literature on leakage-abuse attacks targeting SSE/STE [4, 7, 17, 20, 22, 28], and it is unclear if QE is resilient to these attacks.

Moreover, QE has complex interactions with MongoDB. These interactions have not been considered in the SSE/STE literature [5, 8, 10, 11, 13, 26]. More specifically, unlike the stand-alone schemes that are typically considered in the SSE/STE literature, QE is designed to be a *system component* of MongoDB. An immediate consequence of this design is that queries in QE may trigger events such as caching, logging, and backing up in MongoDB. These events are part of MongoDB's complex database system and it is unclear how they impact QE's security. We believe that this is a reasonable concern, as highlighted by Grubbs et al. [18].

Our Contributions. MongoDB's approach to integrating an SSE/STE scheme as a system component is likely to become the most popular approach for new SSE/STE products in the future. This is because this approach requires minimal engineering effort and entails minimal changes to the existing database system.

In this paper, we investigate whether MongoDB's approach yields secure SSE/STE systems and if this approach should be generally adopted. In this light, we ask the following question:

> *Is it intrinsically difficult to integrate SSE/STE schemes into modern database systems to provide query and data privacy?*

We answer this question affirmatively, using QE as concrete evidence. We discover devastating attacks that result from the interplay between QE and MongoDB's logging system. In particular, we find that MongoDB unintentionally leaks critical statistical information about the queries and data from the logs as a result of operations in QE. We exploit the statistical information in our inference attacks to achieve database reconstruction. We stress that our attacks only require a weak attack setting: we assume a "snapshot" attacker that has access to a snapshot of the encrypted database and some auxiliary information.

We analyse countermeasures to our attacks, concluding that, unfortunately, there is no simple fix to protect QE. Considering that modern database systems contain many other components besides the logging system, we conclude that it is fundamentally challenging to construct and securely integrate SSE/STE schemes into these systems.

QE and its Leakage. At a high-level, a QE operation starts with the client generating encrypted tokens using the field value from the query. These tokens are then sent to the server. The server continues with token derivation and utilizes the freshly generated tokens to read from or write to certain document collections, known as the *metadata collections*. These collections preserve a mapping between field-value-dependent encrypted tokens and their related document identifiers. With the document identifiers, the server can fulfill client query by operating on the corresponding encrypted documents. Due to the complexity of QE, we provide a detailed explanation, using a running example in Section 2. Notably, as a system component, QE interacts with MongoDB's logging system. During QE operations, the logging system records both reads and writes performed by the server in the `mongod` log (later referred to as `queryLog`) and records only writes in the operation log `opLog`. While the `queryLog` is optional for maintenance, the `opLog` is essential to ensure data consistency in deployments. These logs are crucial to our attacks as they offer a source of leakage about the queries and the data. Specifically, we exploit the equality leakage, which reveals whether two encrypted documents share identical field values in the same field. We elaborate on how we extract this equality leakage from logs in Section 4. To properly evaluate our attacks, which are statistical in nature, we need to repeat them many times on freshly built encrypted databases. But the insertion sub-protocol of QE proved too inefficient to meet our needs.[2] So we had to resort to *simulated leakage* in our experiments. More details about how we generate simulated leakage and the correctness of the procedure are presented in Appendix A.

Inference Attacks on QE. Our attacks rely on one of following two assumptions on the logs: a snapshot of `queryLog` and the encrypted documents are obtained after a sufficient number of find queries have been executed (we explore later how many are needed); a snapshot of `opLog` is obtained after a compaction operation — a specialized procedure of QE that reduces the metadata size (cf. Section 2). We justify these assumptions in Section 6, addressing that the leakage due to compaction is inevitable in QE's current design. In addition, we assume the attacker has access to a reference distribution for the plaintext database that is distributed in approximately the same way as the actual plaintext. This is a common assumption in the attack literature [4, 20, 23, 24].

The techniques used in our attacks are similar to those in Gui et al. [19], except that we fine-tune the attack techniques for relational databases as opposed to free-text databases in [19]. Specifically, given an assignment of field values to encrypted tokens, we use the auxiliary distribution to compute a likelihood estimate for that assignment. We then use simulated annealing to modify

[2] In particular, for anonymized person-level American Community Survey data (ACS) datasets used in our experiments, QE takes three to four days to insert 3 million records from ACS on a fast server.

the assignment in a step-by-step manner. Significant novelty in the attacks come in handling the complications of QE, which essentially allows multiple valid tokens per field value, and exploiting the length leakage in an integrated manner. We also use a refinement strategy in which we initially work with one field at a time in updating the assignment, but later switch to updates based on computing likelihoods across multiple fields at once, thus exploiting statistical correlations across fields. A broadly similar approach was used in [4]. Our attacks can also be seen as a concrete instance of the type of attack proposed by Grubbs et al. [18], where the logs of a database system played a central role.

Experimental Validation. For our attack against `queryLog`, we try to recover field values from 3M documents from American Community Survey (ACS) 2013 after seeing 100, 300 or 500 queries on each field, where the queries follow either the uniform distribution or a Zipf distribution. We use ACS 2012 as auxiliary data. For our attack against `opLog`, we try to recover field values from 30K, 300K or 3M documents from ACS 2013, using ACS 2012 as auxiliary data. Our attack achieves between 20% and 100% recovery rate on the field values with respect to the number of unique identifiers in the leakage, or between 40% and 100% recovery rate on the field values with respect to the documents. In the process, we also demonstrate that the length leakage (in the current implementation of QE) has a significant positive impact on the recovery rate of the attack, suggesting that using AES-CTR$ is a bad design decision.

Ethical Considerations. In our experiments, we use the anonymized American Community Survey (ACS) micro data on the person level from 2012 and 2013 and the corresponding codebook, publicly available from [1]. Our attacks do not in any way attempt to deanonymize this data.

On security notions. The assumptions we made for our attacks align with a weaker variant of the single snapshot model. The *(multi-)snapshot security* has been explored in previous works [2,21,27]. Typically, snapshot security requires that even when temporary states (e.g., memory) of the server are included in the snapshot along with the encrypted database, the adversary still cannot learn any information about the underlying queries and data beyond some limited leakage. In contrast, the adversary in our attacks is only able to obtain a copy of all the data stored on the server at a single point in time, but has no access to server state (e.g. its working memory or the tokens sent by the client). Our attacks *do* require access to the server logs, which are usually not included in snapshot security models but are common components of modern database systems. In view of their general availability and utility to attackers (as exposed by our attacks), we argue that such logs should be included in formal models for snapshot security.

These snapshot security definitions sharply contrast with stronger security notions in recent advances in SSE/STE [3,14–16,25,29]. They model the untrusted server, hosting the encrypted database, as an honest-but-curious adversary. This can be further divided into two categories: a *passive-persistent* adversary, who gains access to the encrypted database and is able to persistently observe the transcripts between the client and server; and an *active adversary*, who, in addition to the information above, can adaptively pick the queries.

Indeed, our attacks with the logs underscore the critical need for strong security notions when designing and deploying SSE/STE in the real world.

Countermeasures. We offer a detailed discussion of countermeasures in Section 8, including ideas such as disabling the logs, perturbing/batching the log entries, and encrypting the logs. However, all of these fixes come with disadvantages, making the development of practical countermeasures a challenging task. Because of this, we argue that a major rethink of the QE architecture may be needed.

Disclosure. We informed MongoDB of our analysis on 14.10.2022. They acknowledged receipt and we agreed on a 90-day disclosure period. During the disclosure period, we reached out to MongoDB to ask about their plans for implementing countermeasures to our attacks, but MongoDB did not respond to us with any concrete plans in this respect. The 90-day period concluded on 12.01.2023 and our obligations to MongoDB were discharged at that point. We made the paper public on 05.06.2023.

2 Overview of QE

Notation and Convention. For any $n \in \mathbb{N}$, we let $[n]$ denote the discrete range $[1,n]$. We use $\mathsf{out} \leftarrow\!\$\, A$ to indicate out is generated by some algorithm A using its internal randomness.

A Running Example. We give an overview of QE using a running example. In this example, the client encrypts a collection of documents containing demographic information using QE. The values under *"job"* are encrypted by QE and later referred to as an *indexed encrypted field*. This enables equality searches on this field. The values under *"address"* are encrypted without search functionality (using a standard symmetric encryption scheme, denoted as $\mathsf{Enc}_K(\cdot)$ with some secret key K). In the following, we use "DJ" and "Chef" as short abbreviations for "job: DJ" and "job: Chef", respectively, omitting the "job" field name.

We next explain *token derivation* (Figure 1) and *server*

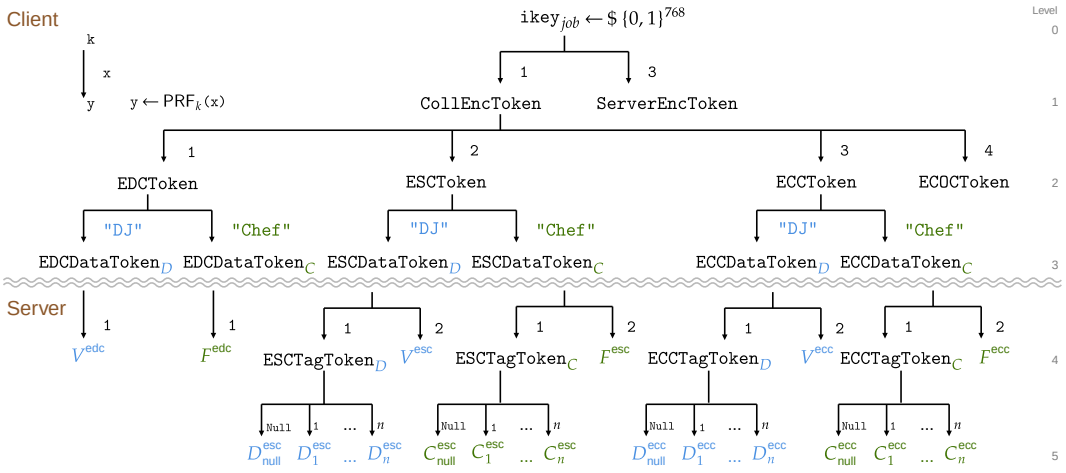

Figure 1: Simplified QE Token Derivation.

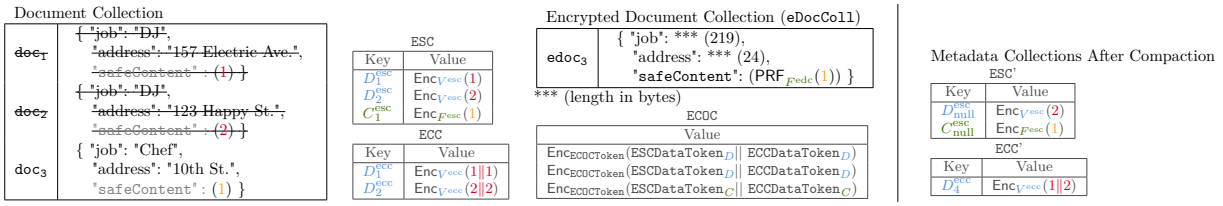

Figure 2: QE Example: server state, including metadata data collections and encrypted document collection after inserting $\texttt{doc}_1, \texttt{doc}_2, \texttt{doc}_3$ and deleting documents $\texttt{doc}_1, \texttt{doc}_2$, then finding "job": "DJ".

state (Figure 3), both fundamental to QE operations. In Figure 1, the $\approx$ separator divides the token derivation process carried out by the client in the upper section and the server in the lower section. The server state is composed of three metadata collections (ESC, ECC, and ECOC) and the encrypted document collection eDocColl. Each document/metadata collection can be seen as a dictionary, with entries being pairs of dictionary keys and values. In our example, the client will insert documents $\texttt{doc}_1, \texttt{doc}_2$ (with "DJ"), $\texttt{doc}_3$ (with "Chef"), subsequently delete $\texttt{doc}_1, \texttt{doc}_2$ (i.e., documents with "DJ"), and finally find documents with "DJ". Figure 3 shows the server state after these operations.

We continue with our example, examining six subprotocols of QE: Init, Connect, Insert, Delete, Find, and Compact. Referring to Figure 1, we represent QE's PRF-based token derivation process, using an arrow that points from some (upper-level) token k to a (lower-level) token y, with the PRF input x noted next to the arrow. This notation is equivalent to $y \leftarrow \mathsf{PRF}_k(x)$.

Init. The client generates a master secret key msk, and stores it either locally or using a cloud key management service (KMS). For field name "job" that supports equality searches, the client generates an index key $\texttt{ikey}_{\text{job}}$ uniformly at random. This key, $\texttt{ikey}_{\text{job}}$, is stored in a dedicated document collection called keyVault. The keyVault collection is encrypted using msk and sent to the server.

Connect. To make queries, the client connects to the server, downloads the keyVault, decrypts it, and keeps $\texttt{ikey}_{\text{job}}$ in memory throughout the active connection.

Insert. The client is connected, retaining $\texttt{ikey}_{\text{job}}$. It then derives CollEncToken and ServerEncToken, using $\texttt{ikey}_{\text{job}}$ with hard-coded values 1 and 3, respectively (to ensure key separation), as shown from level 0 to level 1 in Figure 1. The client further derives EDCToken, ESCToken, ECCToken, and ECOCToken using values 1, 2, 3, and 4 (cf. level 1 to level 2 in Figure 1).

When inserting $\texttt{doc}_1$ with "DJ", the client generates tokens $\texttt{EDCDataToken}_D$, $\texttt{ESCDataToken}_D$, $\texttt{ECCDataToken}_D$ using $\texttt{EDCToken}, \texttt{ESCToken}, \texttt{ECCToken}$ as PRF keys, respectively, and "DJ" (under "job") from $\texttt{doc}_1$ as PRF input (cf. level 2 to level 3 in Figure 1). The client sends $\texttt{EDCDataToken}_D$, $\texttt{ESCDataToken}_D$, $\texttt{ECCDataToken}_D$, and ECOCToken to the server. When inserting $\texttt{doc}_2$ (also with "DJ"), the client sends the same tokens; when inserting $\texttt{doc}_3$ (with "Chef"), the client generates and sends $\texttt{EDCDataToken}_C$, $\texttt{ESCDataToken}_C$, $\texttt{ECCDataToken}_C$, and ECOCToken (cf. levels 2 and 3 in Figure 1).

(ESC) After receiving the tokens from the client, the

server adds entries to ESC. We start by describing the values stored in ESC before encryption. When inserting $\texttt{doc}_1$ with "DJ", the server initializes a counter $\texttt{ctr} \leftarrow 1$, and stores 1. When inserting $\texttt{doc}_2$ with "DJ", the server increments $\texttt{ctr} \leftarrow 2$ and stores 2. This counter is referred to as *document counter*, indicating the number of document insertions that have occurred with "DJ". When inserting $\texttt{doc}_3$ with "Chef", the server initializes another counter $\texttt{ctr}' \leftarrow 1$, and stores 1. To encrypt these counter values, the server generates encryption tokens V^{esc} for $\texttt{doc}_1$ and $\texttt{doc}_2$ (with "DJ"), and F^{esc} for $\texttt{doc}_3$ (with "Chef"), referring to level 3 to level 4 in Figure 1.

The server also needs to determine the dictionary keys used to insert these encrypted document counter values into ESC. To generate the dictionary keys, it first derives tokens: $\texttt{ESCTagToken}_D$ for $\texttt{doc}_1$ and $\texttt{doc}_2$; and $\texttt{ESCTagToken}_C$ for $\texttt{doc}_3$ (cf. level 3 to level 4 in Figure 1). Next, for $\texttt{doc}_1$ with "DJ", it derives dictionary key D_1^{esc} using $\texttt{ESCTagToken}_D$ and $\texttt{pos} = 1$ as PRF input (cf. Figure 1); and inserts $(D_1^{\text{esc}}, \mathsf{Enc}_{V^{\text{esc}}}(1))$ to ESC.

For $\texttt{doc}_2$ with "DJ", the server needs to insert into ESC with a dictionary key different from D_1^{esc}. To do this, the server determines a fresh pos value. This is done by finding $\texttt{pos}_{\max}$, the maximum value of pos used so far in relation to "DJ" in ESC. This is implemented using binary search on some bounded range in QE. At this point, the server recovers $\texttt{pos}_{\max} = 1$. Decrypting ESC's entry at $\texttt{pos}_{\max} = 1$, the server gets the maximum document counter value $\texttt{ctr}_{\max} = 1$ that has been used before with "DJ". For this new insertion, $\texttt{doc}_2$, the server sets $\texttt{pos} \leftarrow 2, \texttt{ctr} \leftarrow 2$, derives D_2^{esc} with $\texttt{pos} = 2$, and adds $(D_2^{\text{esc}}, \mathsf{Enc}_{V^{\text{esc}}}(2))$ to ESC.

Similarly, for $\texttt{doc}_3$ with "Chef", the server adds $(C_1^{\text{esc}}, \mathsf{Enc}_{F^{\text{esc}}}(1))$ to ESC.

(eDocColl) Using the document counter value previously inserted into ESC as the PRF input, the server relies on the EDC token (cf. Figure 1) to produce a PRF output. This PRF output is then added to the encrypted document being inserted to eDocColl. Its purpose is to help the server locate this exact document based on the document counter value, while protecting the counter value in the encrypted document from a snapshot adversary. This specialized PRF output is referred to as safeContent in QE.[3]

In our example, beginning with token derivation, the server derives V^{edc} for $\texttt{doc}_1$ and $\texttt{doc}_2$; and F^{edc} for $\texttt{doc}_3$ (cf. level 3 to level 4 in Figure 1). The server then generates safeContent values, namely $\mathsf{PRF}_{V^{\text{edc}}}(1)$ for $\texttt{doc}_1$, $\mathsf{PRF}_{V^{\text{edc}}}(2)$ for $\texttt{doc}_2$, and $\mathsf{PRF}_{F^{\text{edc}}}(1)$ for $\texttt{doc}_3$, and adds each to encrypted documents $\texttt{edoc}_1$, $\texttt{edoc}_2$, and $\texttt{edoc}_3$, respectively.

[3]The safeContent field contains a list of values, with the list length being equal to the number of indexed fields (in our example, there is only one value).

(ECOC) The ECOC metadata collection stores ESC and ECC tokens used in document insertion (cf. level 3 tokens in Figure 1), encrypted using ECOCToken. We expand on this discussion in the Compact sub-protocol.

Delete. We will now demonstrate the deletion of documents $\texttt{doc}_1, \texttt{doc}_2$ with "DJ", using the "Find-And-Delete" method implemented in QE. Assuming that the client is connected and retains $\texttt{ikey}_{\text{job}}$, the server has already obtained document counter values 1 and 2 associated with "DJ" from ESC (through Find with "DJ"). The server then uses the EDC token to compute safeContents with 1 and 2; and subsequently uses them to locate and delete encrypted documents $\texttt{edoc}_1$ and $\texttt{edoc}_2$. After that, the server adds entries to ECC to record the document deletions associated with 1 and 2. These values are stored in the format of contiguous range $x \| y$, with x and y being the start and the end points. As a result, $1\|1$ and $2\|2$ are to be stored, and they are encrypted using V^{ecc} (cf. level 3 in Figure 1). Similarly to ESC insertions, the server uses $\texttt{pos} \in \{1,2\}$, and derives the associated dictionary keys: it generates D_1^{ecc} for $\texttt{doc}_1$, D_2^{ecc} for $\texttt{doc}_2$; and then adds $(D_1^{\text{ecc}}, \mathsf{Enc}_{V^{\text{ecc}}}(1\|1))$, $(D_2^{\text{ecc}}, \mathsf{Enc}_{V^{\text{ecc}}}(2\|2))$ to ECC (cf. Figures 1 and 3).

Find. The client is connected, retaining $\texttt{ikey}_{\text{job}}$. To find encrypted documents with "DJ", it generates $\texttt{EDCDataToken}_D$, $\texttt{ESCDataToken}_D$, and $\texttt{ECCDataToken}_D$ (cf. level 0 to level 3 in Figure 1), and sends them to the server. The server proceeds with the token derivation for ESC and ECC (cf. level 3 to level 4 in Figure 1). With these tokens, in ESC, it applies the same procedure as in ESC insertion, finding the maximum value of pos, $\texttt{pos}_{\max} = 2$, maximum counter value $\texttt{ctr}_{\max} = 2$. By enumerating from 1 to $\texttt{ctr}_{\max}$, the server obtains document counter values 1 and 2 for all inserted documents with "DJ". In ECC, it finds the maximum $\texttt{pos}_{\max} = 2$ associated with the number of deletions associated with "DJ", and then decrypts ECC entries at $\texttt{pos} \in \{1,2\}$ to obtain 1 and 2. These are deleted document counter values associated with "DJ". After removing the deleted set $\{1,2\}$ from the inserted set $\{1,2\}$, the server returns an empty result to the client.

If a find query on "Chef" were performed by the client instead, similar operations would yield document counter value 1. The server would then compute safeContent with 1 as the PRF input using token V^{edc} as the PRF key, locate $\texttt{edoc}_3$, and return it to the client.

Compact. The sizes of the metadata collections ESC, ECOC, and ECC grow linearly with the number of document insertions and deletions. Since these sizes can become quite large,[4] QE implements a Compact procedure to reduce

[4]In our experiments, described later, ECOC reached 1GB in size when only inserting around 650K records.

them. We now describe this Compact procedure continuing with our running example. Assuming the client is connected and retains $\texttt{ikey}_{\text{job}}$, it generates ECOCToken, initiates Compact, and sends ECOCToken to the server.

Previously, during document insertion, we omitted the description of the tokens stored in ECOC. We now provide this information: $\mathsf{Enc}_{\texttt{ECOCToken}}(\texttt{ESCDataToken}_D \| \texttt{ECCDataToken}_D)$ is added twice during the insertion of $\texttt{doc}_1$ and $\texttt{doc}_2$; $\mathsf{Enc}_{\texttt{ECOCToken}}(\texttt{ESCDataToken}_C \| \texttt{ECCDataToken}_C)$ is added when inserting $\texttt{doc}_3$ (cf. Figure 3).

Continuing the Compact process, the server decrypts the ECOC entries sequentially with ECOCToken. It first obtains $\texttt{ESCDataToken}_D \| \texttt{ECCDataToken}_D$ (from inserting $\texttt{doc}_1$ with "DJ"). With $\texttt{ESCDataToken}_D$, the server compacts "DJ"-related entries in ESC. It first identifies the maximum document counter value 2 stored for "DJ" in ESC (using the same approach described in document insertion). Next, it encrypts and inserts only the maximum document counter value 2 into ESC, removing individual entries 1, 2 as they can be recovered using 2 by enumeration. This maximum counter value 2 is then encrypted using F^{edc}, and is stored at a special position null in ESC. Specifically, ESC now only stores $(D^{\text{esc}}_{\texttt{null}}, \mathsf{Enc}_{V^{\text{esc}}}(2))$ for "DJ". With $\texttt{ECCDataToken}_D$, the server compacts "DJ"-related entries in ECC. Entries $1\|1$ and $2\|2$ are regrouped as a single contiguous range $1\|2$. ECC now only stores $(D^{\text{ecc}}_4, \mathsf{Enc}_{V^{\text{ecc}}}(1\|2))$ for "DJ".[5] The server then processes the second entry, obtaining $\texttt{ESCDataToken}_D \| \texttt{ECCDataToken}_D$. Since the server has already performed the compaction subroutine using the same tokens, this ECOC entry is skipped. Finally, the server gets the third entry $\texttt{ESCDataToken}_C \| \texttt{ECCDataToken}_C$ (from inserting $\texttt{doc}_3$ with "Chef"). By performing similar operations, the server "compacts" ESC by removing the previous entry at $\texttt{pos} = 1$ with maximum counter value 1, and inserts $(C^{\text{esc}}_{\texttt{null}}, \mathsf{Enc}_{F^{\text{esc}}}(1))$ to ESC. No compaction is performed on ECC for "Chef", as no documents with "Chef" have been deleted.

The server has now completed the compaction process on ESC and ECC, using every ESC and ECC token stored in ECOC. Finally, the server can clear ECOC. Figure 3 also shows the state of the metadata collections after compaction.

It is worth emphasizing that the lack of technical documentation for QE has posed significant challenges in our analysis. We managed to understand QE's inner workings through access only to source code and presentations.[6]

[5]The positional value used for this entry is 4 due to a small complication: a placeholder record is inserted at position 3.

[6]https://github.com/mongodb/mongo; https://www.youtube.com/watch?v=0TuCB1pSWZE

3 System Integration of QE

MongoDB builds QE on top of native MongoDB operations, and integrates it as a system component. Thus, QE inevitably interacts with multiple system components of MongoDB. Notably, we focus on its comprehensive logging system. In particular, two types of logs produced by the system, both stored on the server's hard drive: the mongod log file (referred to as the query log queryLog), and operation log opLog.

For better understanding, we provide simplified outputs of queryLog and opLog in Figure 2. These outputs are generated by the execution of the QE example we discussed before.[7]

In queryLog at verbosity level 1,[8] QE records all server operations that incur write/read changes to the server state (e.g., the metadata collections and encrypted document collection) using transactions. For example, when inserting $\texttt{doc}_1$, transaction 101 records dictionary key D^{esc}_1, used for ESC insertion, followed by identifier $\texttt{edoc}_1$, used for inserting encrypted document to the encrypted document collection eDocColl. Inserting $\texttt{doc}_2$ results in transaction 105. Similarly, it records ESC dictionary key D^{esc}_2 and identifier $\texttt{edoc}_2$. We omitted the transaction for inserting $\texttt{doc}_3$ in queryLog for simplicity, which records dictionary key F^{esc}_1 and identifier $\texttt{edoc}_3$. Finally, transaction 503 records the find operation, incurring reads on ESC, with dictionary keys D^{esc}_1 and D^{esc}_2.

In comparison, opLog only records write changes to server state and does not capture any information related to the read operations. For simplicity, the provided opLog omits the document insertion part. Instead, we focus on the write transactions, incurred from the compaction process on ESC. Recall that this compaction process involves running a compaction subroutine on every unique inserted field value ("DJ" and "Chef"). Specifically, the maximum document counter values for "DJ" and "Chef" are kept, and the remaining individual entries are removed. In our example, transaction 211 records the write operations, occurring during the compaction subroutine on "DJ", and transaction 212 records those corresponding to "Chef".

4 Leakage Extraction from Logs

We now proceed to discuss our method for extracting certain *frequency* information related to the (indexed) field values from either queryLog or opLog.

In our example, this (extracted) frequency information

[7]These logs are only for illustration purposes; e.g., we use txnid to denote the transaction ids, which are different from what are used in the actual logs; in opLog, "i" and "d" are used to denote "Insert" and "Delete", respectively.

[8]The maximum level is 5.

Figure 3: Simplified Logs: `queryLog` and `opLog`.

includes the fact that two unique values have been inserted under "job"; and that for "job", $\texttt{edoc}_1$ and $\texttt{edoc}_2$ share the same field value, while $\texttt{edoc}_3$ has a different field value from the other two. The extracted frequency information will be used later in our inference attack.

Our leakage extraction process has two steps. (1) Identifying which ESC tokens are for the same field values; this information can be extracted by exploiting either `queryLog` or `opLog`. (2) Mapping the information extracted in step 1 onto encrypted documents. We demonstrate this process using the same log examples as before. We refer to a (logging) transaction with $\mathsf{txnid} = x$ as transaction x.

(Step 1 with `queryLog`.) Let us look at how to identify ESC tokens with the same underlying field values in `queryLog`. For the purpose of this step, it is sufficient to focus on the transactions for the Find operations. In particular, in $\mathsf{txnid} = 503$, D_1^{esc} and D_2^{esc} appear together. Since a Find operation only returns documents containing the same field value, we can deduce that D_1^{esc} and D_2^{esc} must have the same underlying field value. On the other hand, since F_1^{esc} did not appear in $\mathsf{txnid} = 503$, it must have a different underlying field value.

(Step 1 with `opLog`.) We can infer which ESC tokens have the same underlying field values from `opLog` in a similar manner. Consider the `opLog` in Figure 2. In the transactions for $\mathsf{txnid} = 211$, D_1^{esc} and D_2^{esc} appear together again. And this tells us that these two ESC tokens are for the same field value. We can omit $D_{\text{null}}^{\text{esc}}$ in this process as it is an Insert operation due to compaction as opposed to a Delete one; and only the deleted ones are used during document insertion. Similarly, we can identify F_1^{esc} as an ESC token for a different field value.

(Step 2.) Using the extracted information from Step 1 (from either `queryLog` or `opLog`), we can now map the information onto the encrypted documents. This is done by exploring a different segment of the logs.

Concretely, for `queryLog`, we see that the insertion for ESC token D_1^{esc} and encrypted document $\texttt{edoc}_1$ both appear under transaction $\mathsf{txnid} = 101$. This means that D_1^{esc} is for $\texttt{edoc}_1$. Similarly, we can link D_2^{esc} with $\texttt{edoc}_2$ and F_1^{esc} with $\texttt{edoc}_3$ (the latter transaction has been omitted in the example).

Similarly, there are transactions for inserting encrypted documents in `opLog`. These allow us to link the ESC tokens to the encrypted documents they belong to. Due to limitations on space, these transactions are not shown in the `opLog` example in Figure 2.

Tying everything together, the information on which ESC tokens have the same underlying field values, and the knowledge of which ESC token belongs to which encrypted document allow us to obtain a type of leakage. This leakage tells us which of the encrypted documents share the same field values. Formally, we call this leakage *field-value equality leakage*. We can represent it as follows. Let m be the number of encrypted documents; we define this leakage as $\mathcal{L}_{\text{field-value-eq}} : [m] \to \mathbb{N}$. In our example, the field-value equality leakage can be represented as $\mathcal{L}_{\text{field-value-eq}}(1) = 1$, $\mathcal{L}_{\text{field-value-eq}}(2) = 1, \mathcal{L}_{\text{field-value-eq}}(3) = 2$. We expand on the leakage representation in Section 5.

Length Leakage. We can also extract length leakage for individual encrypted fields that arises as a result of QE's unfortunate choice of using AES-CTR$ encryption without any length padding. This is because the byte length of an indexed encrypted field value is equal to the sum of the plaintext length and a constant (for storing content such as fixed-size tokens, fixed at "215" in QE). In fact, the length leakage can be obtained from either `eDocColl` or `opLog`. For example, in Figure 3, in $\texttt{edoc}_3$, the encrypted field value under "job" has 219 bytes, meaning the plaintext length is 4 bytes. We formally represent the length leakage by $\mathcal{L}_{\text{length}} : [m] \times [\tau] \to \mathbb{N}$, with m and τ defined before, and the output is the length of the encrypted field value in bytes.

5 Further Details on QE and Implications for Leakage Extraction

Thus far, we have provided an overview of QE's six subprotocols: Init, Connect, Insert, Delete, Find, using a running example. In this section, we turn our attention to the details of QE omitted so far, and discuss how they

affect our leakage extraction.[9]

Contention Factor. Note that the Insert protocol we described in Section 2 can only support document insertions for documents with the same field value sequentially. This is because the ESC dictionary values depend on how many documents containing that field value has been inserted. So if two clients try to insert documents containing the same field value at the same time, the Insert queries can only be executed one by one, leading to undesired latency. To circumvent the shortcoming, QE introduced *contention factor* between level 3 and level 4 in the token derivation process (cf. Figure 1).

To demonstrate how contention factor helps with the problem described above, consider an example where two clients want to insert a document containing field value "DJ" each. Instead of deriving $\texttt{ESCDataToken}_D$ (level 3) and sending it directly to the server, the first client generates a small random number cf (called contention factor) and computes $\texttt{ESCDataCfToken}_D \leftarrow \mathsf{PRF}_{\texttt{ESCDataToken}_D}(\texttt{cf})$. It then sends $\texttt{ESCDataCfToken}_D$ in place of $\texttt{ESCDataToken}_D$ to the server in the Insert sub-protocol. Similarly, the second client generates a contention factor $\texttt{cf}'$, derives token $\texttt{ESCDataCfToken}'_D$ and sends $\texttt{ESCDataCfToken}'_D$ in its Insert query. By setting a reasonable valid range for the contention factor,[10] the chance of the two clients generating the same contention factor is low. So with high probability, the server can process the two insertions concurrently, thus reducing query latency.

Of course, the Find sub-protocol needs to be modified to support contention factors. We elaborate how it can be done by considering a Find query on field value "DJ". Here, since the client does not know which contention factors it is looking for, it delegates the search over the contention factors by sending $\texttt{ESCDataToken}_D$ and the maximum possible contention factor cf_max to the server. The client also sends $\texttt{EDCDataToken}_D$ and $\texttt{ECCDataToken}_D$ (as in the unmodified Find sub-protocol) to the server. The server can then derive all possible $\texttt{ESCDataCfToken}_D$ tokens and perform search with $\texttt{EDCDataToken}_D$, $\texttt{ESCDataCfToken}_D$ and $\texttt{ECCDataToken}_D$ in the same way as the unmodified Find sub-protocol.

Multiple Indexed Encrypted Fields. In practice, QE supports more than one equality-searchable field per document. This is achieved by using different index keys $\texttt{ikey}_{\text{job}}$ (cf. level 0 of Figure 1) for each field and field value. This complication does not affect how leakage can be extracted from the logs. The rationale behind this is that the Insert sub-protocol processes field insertions sequentially; as a result, an attacker can identify the dictionary keys used for each field by associating them with the order of field insertions in the logs.

Effects on Leakage Extraction. For the complication due to multiple indexed encrypted fields, we generalize the $\mathcal{L}_{\text{field-value-eq}}$ representation (cf. Section 2) to accommodate τ indexed encrypted fields. Let m be the number of encrypted documents; we define $\mathcal{L}_{\text{field-value-eq}} : [m] \times [\tau] \to \mathbb{N}$. Suppose now we also index the second "address" field in our example, then $\tau = 2$, and we can write the field-value equality leakage for the first indexed field "job" as $\mathcal{L}_{\text{field-value-eq}}(1,1) = 1, \mathcal{L}_{\text{field-value-eq}}(2,1) = 1, \mathcal{L}_{\text{field-value-eq}}(3,1) = 2$.

As an abuse of notation, we let $\mathcal{L}_{\text{field-value-eq}}(i)$ denote the tuple of enumerated field values for (encrypted) document i. Let 1,2,3 be the enumeration of the second "address" field, as this information is distinct for all documents. Then, using this notation, we can write $\mathcal{L}_{\text{field-value-eq}}(1) = (1,1), \mathcal{L}_{\text{field-value-eq}}(2) = (1,2), \mathcal{L}_{\text{field-value-eq}}(3) = (2,3)$.

We continue discussing how the contention factor mechanism affects leakage extraction. Suppose $\texttt{doc}_1$, $\texttt{doc}_2$, and $\texttt{doc}_3$ are now inserted into ESC with tokens generated with distinct cf. With queryLog, we can still extract the field-value equality leakage using the same method described in Section 2. This is due to the correctness requirement of Find protocol, which returns all encrypted documents with the same indexed field value, even if they are inserted with different cf-dependent tokens; at the same time, queryLog records all dictionary keys (generated with different cf values) used in this Find query.

However, with opLog alone, we now cannot learn that $\texttt{edoc}_1$ and $\texttt{edoc}_2$ have identical field value. This is because, the compaction process now only groups encrypted fields inserted with tokens generated using the same field value *and* cf value, while $\texttt{edoc}_1$ and $\texttt{edoc}_2$ are inserted with tokens derived using different cf values. Hence, we define $\mathcal{L}_{\text{field-value-cf-eq}} : [m] \times [\tau] \to \mathbb{N}$ to represent this type of equality leakage, referred to as *field-value-contention-factor equality leakage*. In this example, $\mathcal{L}_{\text{field-value-cf-eq}}(1,1) = 1, \mathcal{L}_{\text{field-value-cf-eq}}(2,1) = 2$, $\mathcal{L}_{\text{field-value-cf-eq}}(3,1) = 3$.

We provide the pseudocode for leakage extraction in Figures 7 and 8 in Appendix B. It is important to emphasize that the leakage extraction methods in our implementation are far more complex than what we have described. This is because raw logs are "messy" (See Appendix F). For example, we need techniques to obtain a unique transaction id for use in leakage extraction, and ensure that our leakage extraction methods remain memory-efficient when dealing with large logs.

[9] Full syntax and pseudocode are provided in the full version of our paper.

[10] The default range for the contention factor is 0 to 4, inclusive.

6 Adversarial Models

As briefly discussed in Section 1, we consider a snapshot adversary with access to all of the data stored in the encrypted database — this include three metadata collections (`ESC`, `ECC` and `ECOC`), the encrypted document collection and logs (`queryLog` and `opLog`). We devise two attacks, one for each log. These attacks demonstrate how the log can be exploited to reveal sensitive information about the encrypted documents.

6.1 Snapshot Adversary with Encrypted Document Collection and `queryLog`

Attack Setting. Recall that `queryLog` is used to store database events. At verbosity level 1 and above, details of queries appear in the log. We consider an attacker who has access to a snapshot of the encrypted document collection and `queryLog`. The attacker here can be a database administrator, an external attacker who manages to gain access to the encrypted database, or an attacker who steals the hard drive from the server.

Attack Goal. With a copy of the encrypted document collection and `queryLog`, the goal of the attacker is to recover the field values in the encrypted document collection.

Attack Requirements. The attack requires a sufficient number of queries to be made by the client. Here, sufficiency depends on the distribution of the field values and the query distribution. In Section 7.2, we show that 300 uniformly random queries on each field of the American community survey 2013 dataset (6 fields in total) enables the attacker to recover between 40% and 80% of the field values (see Figure 4 in Section 7.2).

6.2 Snapshot Adversary with `opLog`

Attack Setting. The `opLog` stores all write operations (but not read operations) on the database. The `opLog` must always be enabled in the current implementation of QE. Even if this log were to be made optional in QE, in production deployments, `opLog` is typically necessary because the database would need to be instantiated as a replica set[11] and `opLog` is required for database maintenance. In this paper, we investigate the security impact of a snapshot attack where an adversary manages to obtain a copy of `opLog`. In practice, this adversary can be a database administrator, an attacker who gains access to the encrypted database (if the connection to the database is unauthenticated or the access keys are leaked), or an attacker who steals the hard drive from the server.

Attack Goal. With a copy of `opLog`, the goal of the attacker is to recover the field values in the encrypted document collection.

Attack Requirements. The attack requires a Compact operation to take place on the database. We emphasize that the compaction procedure is inevitable in the design of QE, as the scheme incurs significant storage blow-up in the size of metadata, and it is advised to perform compaction every time the metadata reaches 1GB.[12] On the other hand, the attack does *not* require any query to be made on the database.

7 Snapshot Attack with `queryLog` or `opLog`

In this section, we show how the information leakage from `queryLog` and `opLog` can be exploited to reconstruct the plaintexts in the encrypted document collection. We start with an overview of our attack technique in Section 7.1, and follow this with an empirical evaluation of our attack in Section 7.2.

7.1 Inference Attack

We begin by showing how we model the distribution the field values in the document collection. We then derive the *likelihood function* used in our attack. Finally, we show how the likelihood function can be used to find the most likely assignment between the identifiers in the leakage and the unencrypted field values by using simulated annealing.

Field value distribution. We represent a document with $\texttt{doc} = (\texttt{val}_1, \ldots, \texttt{val}_\tau)$, where $\texttt{val}_i$ is a particular field value in the i-th field. We can view the tuple of field values as a realisation of a random variable $\mathcal{T}$, where the support of the variable T is the Cartesian product of the set of possible field values across all fields. Without loss of generality, we assume that the tuple of field values t_i has probability p_i to occur in the document collection every time a new document is added. Then, given $n_1, \ldots, n_l$ as the number of occurrences of tuples $t_1, \ldots, t_l$, we can compute the probability of observing

[11] That is, alongside a primary server, at least one other server instance maintains the same database to cope with emergencies such as a temporary power outages; `opLog` contains a list of write operations and is essential for ensuring that the databases maintained in the replica set are kept in sync.

[12] More specifically, it is the size of the `ECOC` collection. Currently, compaction is triggered manually, but it is planned to be automated by the server in a future release. See https://www.mongodb.com/docs/manual/core/queryable-encryption/reference/limitations/#manual-compaction.

the document collection as

$$\Pr[(n_1,\ldots,n_l)] = C \cdot \prod_{i=1}^{l} p_i^{n_i}, s$$

where C is a normalization constant that depends on $n_1,\ldots,n_l$.

We define *marginal field value tuple distribution* on an index set $I = \{i_1,\ldots,i_s\}$ as the probability

$$\Pr_I[(n_1,\ldots,n_{i_s})] = \sum_{x_j, j\in[\tau]\setminus I} \Pr[(x_1,\ldots,n_1,\ldots,n_{i_s},x_\tau)].$$

As an example, if the document collection contains two fields and $I = \{1\}$, then the marginal distribution on I is essentially the field value frequency of first field. More formally, it can be computed as

$$\Pr_I[(n_1)] = \sum_{x_2} \Pr[(n_1,x_2)].$$

Reduced Tuple. Let I be an index set (i.e. a set of integers), we use $t[I]$ to denote the reduced tuple of t indexed by I, that is, the i-th component is in $t[I]$ if and only if i is in I. For example, if $I = \{1,2,3\}$, then $t[I] = (t_1,t_2,t_3)$.

Identifier-field value Assignment. Recall that after extracting the leakage $\mathcal{L}_{\text{field-value-eq}}$ (and $\mathcal{L}_{\text{field-value-cf-eq}}$), we know which of the documents contain the same field values (and field value with contention factor, respectively), but we do not know the exact value of the field values. For this reason, the field values are enumerated by identifiers. Our goal in the attack is to map these identifiers to actual field values. This map $\mathbf{P}$ is formally defined as $\mathbf{P} : [\tau] \times \mathbb{N} \to \mathbb{N}$ where τ is the number of fields in the document collection, and $\mathbf{P}(i,j) = k$ means that the j-th identifier (in either field-value-contention-factor equality or field-value equality leakage) in the i-th field is assigned to the k-th field value.

For field-value-contention-factor equality leakage, the assignment is many-to-one, meaning that there are multiple identifiers assigned to the same field value. For field-value equality leakage, the assignment is one-to-one, meaning that each identifier is assigned to a unique field value.

Likelihood function. The formalism above allows an attacker to compute the probability of observing the leakage given a particular assignment. However, the attacker is interested in the *likelihood* $\mathbf{L}$ of an assignment given the observed leakage $\mathcal{L}$. To compute the latter, we use Bayes' rule as follows:

$$\begin{aligned}\mathbf{L}[\mathbf{P} \mid \mathcal{L}] &\propto \Pr[\mathbf{P}] \cdot \Pr[\mathcal{L} \mid \mathbf{P}] \\ &= \Pr[\mathbf{P}] \cdot \Pr[\mathsf{DB} \mid \mathsf{DB}(i,j) = \mathbf{P}(j, \mathcal{L}(i,j)) \forall i,j].\end{aligned}$$

We assume that the assignments are uniformly distributed so $\Pr[\mathbf{P}]$ do not have any real contribution in the likelihood. Regarding $\Pr[\mathcal{L} \mid \mathbf{P}]$, we can apply the assignment $\mathbf{P}$ on the leakage $\mathcal{L}$ to obtain a guess of the plaintext document collection DB. The probability can then be calculated with what we have described in "Field Value Distribution".

Let I be an index set, then, similar to marginal plaintext tuple distribution, we define marginal likelihood as

$$\mathbf{L}_I[\mathbf{P} \mid \mathcal{L}] \propto \Pr[\mathbf{P}] \cdot \Pr_I[\mathcal{L} \mid \mathbf{P}].$$

Simulated Annealing. Recall that the goal of the attacker is to find the most likely assignment given the leakage and auxiliary distribution. We have just established how likelihood score can be calculated for any assignment. However, the number of possible assignments is exponential in size and it is impossible for the attacker to compute the likelihood score for every possible assignment and pick the best one. To achieve the latter, we propose to use simulated annealing [32].

On a high-level, (standard) simulated annealing works as follows. The algorithm is initialised with a *temperature* T and a random assignment $\mathbf{P}$. Then, for each iteration, the temperature T is reduced according to a *cooling schedule* $\mathsf{cooling}()$. A new assignment $\mathbf{P}'$ is generated by using a neighbourhood generation algorithm $\mathsf{neighbour}()$. This new assignment is supposed to be almost identical to the old assignment except for a small number of differences. A likelihood score $s' = \mathbf{L}(\mathbf{P}' \mid \mathcal{L})$ is computed on the new assignment and is compared to the likelihood score s computed on the old assignment. If the score improves, we replace the old assignment by the new one unconditionally. Otherwise, the new assignment is accepted with a small and decreasing probability specified by the algorithm $\mathsf{acceptProb}()$ which takes as input the old likelihood score s, the new likelihood score s' and the temperature T. After the set number of iterations have been completed, the algorithm returns the assignment that it is currently holding and terminates.

Adapting simulated annealing to our attack. Applying simulated annealing naively to the above likelihood function turns out to be ineffective at finding the most likely assignment. This is because the search space is too large for the assignment to converge properly (it grows exponentially with the number of fields). To tackle the problem, we use a technique similar to that in [4].

The idea is that before running simulated annealing on multiple fields, we run it on the individual fields so that we have a better starting point for the algorithm. We find that simulated annealing works the best when the neighbourhood algorithm is constrained to changing at most 6 identifier-field value pairs in the assignment

(across 3 fields, 2 pairs in each field).

In terms of the score function, we use the marginal likelihoods over the fields that have been changed by the neighbourhood function. This allows the algorithm to ignore the other fields and produce a more informative likelihood function over the chosen fields. The pseudocode of our attack is provided in Appendix C.

7.2 Empirical Evaluation

In this section, we present an empirical evaluation of the attack described above.

Experimental Data. We use American community survey (ACS) from 2013 as the target dataset in our experiments. Detailed information about the dataset and how it is processed can be found in Appendix D.

We pick six fields from ACS as the target fields in our experiments, namely race (RAC3P), state (ST), place of birth (POBP), place of work (POWSP), class of worker (COW) and occupation (OCCP). In the original ACS dataset, all of the fields are numerically encoded, which means there is no length leakage to be exploited. To demonstrate the security impacts of length leakage, we run additional experiments where the field values are replaced with the actual plaintexts and their length leakage is given to the attacker.

To investigate the effect of size of the database on our attack, we run separate experiments for 30K, 300K and 3M documents sampled uniformly randomly from the whole dataset. Due to scalability issues with QE, we decided to generate leakages from the dataset through simulation, also discussed in Appendix A.

Auxiliary Data. We use ACS 2012 as our auxiliary data. Since ACS samples different households every year, ACS 2012 provides a good approximation for ACS 2013.

Parameters for Our Attack. As for the parameters of our attack, we run simulated annealing on the individual fields for 10^7 iterations when the leakage comes from `opLog` and 5×10^7 iterations when the leakage comes from `queryLog`. We then run simulated annealing across all fields (the number of fields picked in the neighbourhood algorithm is uniform random between 1 and 3) for 10^4 iterations when the leakage comes from `opLog` and 2×10^4 iterations when the leakage comes from `queryLog`. These parameters allow us to complete each attack within a day.

Accuracy Metrics. We use two accuracy metrics to measure the performance of our attack. Both metrics are field-based. The first metric is the percentage of correctly guessed plaintexts in a field with respect to the unique identifiers in the field-value equality leakage or field-value-contention-factor equality leakage for that field. This metric disregards the frequencies of the underlying plaintexts and it focuses on the raw performance of the attack. The second metric is the percentage of correctly guessed plaintexts in a field with respect to the actual encrypted documents. This metric gives weights to the frequencies of the plaintexts and it reflects the amount of privacy loss on the encrypted document collection due to our attack.

Experiment Overview. For our attack on `opLog`, we use datasets of size 30K, 300K and 3M documents. For each choice of the dataset size, we generate 100 independent encrypted databases through simulation and extract from them field-value-contention-factor equality $\mathcal{L}_{\text{field-value-cf-eq}}$ and length leakages $\mathcal{L}_{\text{length}}$ with the procedures detailed in Sections 4 and 5. This gives us 100 inputs to our attack with $\mathcal{L}_{\text{field-value-eq}}$ (by removing the contention factor) and another 100 inputs to our attack with $\mathcal{L}_{\text{field-value-cf-eq}}$ and length leakage.

For our attack on `queryLog`, we use datasets of size 30K, 300K and 3M documents. For each choice of the dataset size, we generate 100 independent encrypted databases through simulation. For each encrypted database, we run 100, 300 or 500 random queries (either uniform or Zipf) on *each* field and generate field-value equality leakage through simulation. We also generate length leakage from the encrypted database directly. This gives us 100 inputs to our attack with field-value equality leakage only and another 100 inputs to our attack with field-value equality leakage and length leakage for each query size and query distribution.

Query Distribution. We consider two distributions of queries, namely the uniform distribution and a Zipf distribution for the experiment above. For the uniform distribution, we simply draw 100, 300 or 500 field values uniformly randomly for each field and use them as the field values for the queries.

For the Zipf distribution, we generate the queries as follows. Focusing on one of the fields, suppose that the field values are $\texttt{val}_1, \ldots, \texttt{val}_k$. We randomly permute the field values to obtain a new order, say $\texttt{val}_{i_1}, \cdots, \texttt{val}_{i_k}$. We treat $\texttt{val}_{i_1}$ as the most likely field key to be queried and $\texttt{val}_{i_k}$ to be the least likely field key to be queried, modelling the query frequency of the field values to be:

$$\frac{j^{-1}}{\sum_{n=1}^{k} n^{-1}}.$$

for field value $\texttt{val}_{i_j}$. We draw 100, 300 or 500 field values from the said distribution and use them as the field values for the queries.

Experimental Results. For `queryLog`, we report our experimental results on 3M documents and 300 queries in Figure 4. For results on other query sizes, see Appendix E.

As the figures are quite complicated, we provide the following guide for readers. Each figure shows six groups of experimental results, one for each field. For each field, we show four results, for two accuracy metrics in two different experimental settings. The results for the experiments without length leakage from the field values are shown in the left pair and the results for the experiments with length leakage from the field values are shown in the right pair. Within the pairs, the plaintext recovery rate by the number of unique field value identifiers is shown on the left and the plaintext recovery rate by the number of documents is shown on the right. As we use random queries in our experiments, these queries do not necessary touch all field values in the database (hence, the attacker does not observe the corresponding leakage in `queryLog`). The proportions of field values touched by the queries are shown as white bars in the figures. This allows one to read off the relative plaintext recovery rates of our attack in addition to absolute plaintext recovery rates. For example, in the setting of 300 uniform queries on each field on 3M documents (top figure in Figure 4), for the experiment without length leakage, the attacker observe 75.2% of the field values in the field POBP on average. It is able to recover 12.0% of the field values. This corresponds to $12.0/75.2 = 16.0\%$ relative plaintext recovery rate.

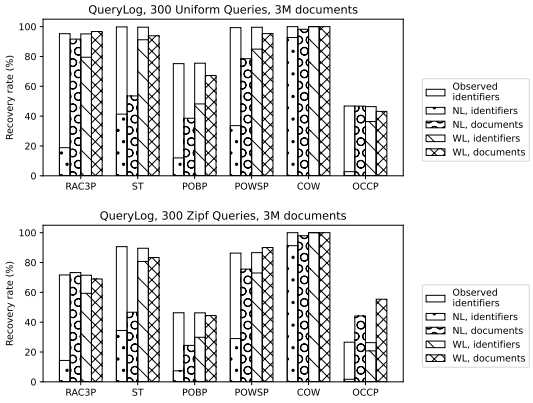

Figure 4: Experimental results on `queryLog` with 3M documents and 300 random queries on each field. "NL" indicates "no length leakage"; "WL" indicates "with length leakage".

Our attack has lower recovery rate on the field values when length leakage is not given. This is not surprising as the attacker has not observed all the (encrypted) field values yet (per the white bars), and it makes it hard for the attacker to recover the field values. Even then, the recovered field values covers the majority of the documents, indicating that QE offers very little protection for data privacy. Furthermore, our attack performs significantly better when length leakage is accessible by the attacker.

We observe that the performance of our attack on different fields varies significantly. This is because the difficulty of performing inference is different for different fields. For example, the field state (ST) is relatively easy to infer as it can only take one of 51 possible values. 40% of the states have a reasonably distinct distribution as compared to other states, which allows our attack to recover them easily. On the other hand, our attack only managed to recover less than 20% of place of birth (POBP). This is because there are 215 possible place of birth in total and our target database has very skewed distribution of POBP. In particular, being the third most frequent field value for POBP, "Texas" occurs in 6.06% of the documents. This makes "Texas" very easy to identify from a frequency analysis perspective. Furthermore, people born in Texas are likely working in nearby states (shown in the auxiliary information), which allows the attacker to use correlation between different fields (ST and PoBP) to improve the accuracy of the recovery further. On the other hand, for minorities born in foreign countries, say Cyprus, the frequency of the field value is too low and its correlation with other fields is too weak to provide any meaningful statistical information for recovery.

For `opLog`, we report our experimental results on 3M documents in Figure 5. For results on other database sizes, see Appendix E.

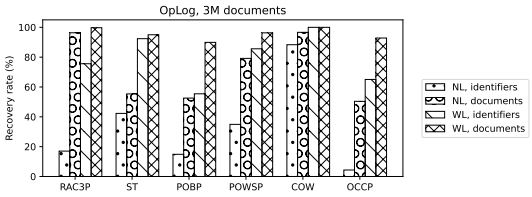

Figure 5: Experimental results on `opLog` against 3M documents. "NL" indicates "no length leakage"; "WL" indicates "with length leakage".

Our attack on `opLog` has similar performance as compared to our attack on `queryLog`. We observe that the presence of length leakage has a significant positive impact on the recovery rate. Another interesting observation is that even though the leakage of `opLog` is more "noisy" due to the presence of contention factor, our experimental results are not worse than those using `queryLog`. This indicates that contention factor (a relatively small one, e.g. 4, as used in QE) offers limited protection against leakage-abuse attacks.

8 Countermeasures

MongoDB's Plan. During the disclosure process, we asked MongoDB about their plan for deploying countermeasures to our attacks. They said that they are still

actively developing QE and did not give us a concrete plan for their deployment of countermeasures. We have tested our attacks up to tag `r6.2.1-rc1`[13] (released on 17.02.2023) and confirmed that they still work.

On one hand, we understand MongoDB's position as there are many uncertainties in the development of QE and it does not make sense to try to "hit a moving target". On the other hand, we believe that the vulnerabilities we have identified should be addressed before general availability (GA) as their presence defeats the purpose of QE otherwise.

Challenges in designing a countermeasure. We consider three natural countermeasures to our attacks below. While these countermeasures do defeat our attacks, we consider them to be either hard to implement (not aligned with the interest of MongoDB) or that they hinder the usability of the database system.

Disable the logs. The most naive countermeasure to our attacks is to simply not use the logs. For `queryLog`, while it is acceptable to not enable it in most cases, the log contains important information for debugging purposes. For that reason, `queryLog` needs to be supported by QE, and the best possible defence against our attack is to implement access control over who can turn on or access `queryLog`. For `opLog`, disabling the log renders QE completely unusable in practical deployments. This is because `opLog` is essential in synchronising different servers in a replica set, which in turn is needed to ensure high data availability.

Perturb/batch the log entries. One less aggressive countermeasure is perturbing or batching the entries of the logs. Although this countermeasure can certainly confuse our attacks, it may cause usability problems with the logs as well. In particular, if the query events in `queryLog` are not logged in order or are batched, it will be difficult for an engineer to use the log for debugging. Similarly, if the write events in `opLog` are shuffled and/or batched, it will be much harder to synchronise the servers within a replica set. Given the complexity of logging and synchronisation systems, it may take MongoDB several person-years of efforts to implement such a change.

Encrypt the logs. Finally, we consider the possibility of encrypting the logs. There are two main ways this can be implemented. In the first approach, the key used to encrypt the logs is stored on the server and the server uses it to encrypt the new entries of the logs as they are generated. This will not make QE secure against a standard snapshot adversary, as the key would be visible to such an attacker.

In the alternative approach, the key used to encrypt the logs is stored on the client and made available to the server on request. There are two main problems with this approach. Firstly, this would be a significant architectural change requiring significant recoding.

Secondly, modern databases only dumps log events to disks whenever the load on the server is low. This means that the client may need to remain online for a long time to send the log encryption key to the server.[14]

9 Conclusions

In this section, we discuss what went wrong with the QE design process and what might be learned from this. Our discussion is necessarily speculative given the limited information available in the public domain.

On the security notion. None of the security models in the SSE/STE literature [5,6,8,9,13,14,26,31] includes leakage from logs. This may explain why MongoDB apparently overlooked them in any internal security analyses they may have conducted. Indeed, the only discussion of the sensitivity of logs in the SSE/STE context is in [18]. Our inference attack concretely demonstrates the need to include logs in formal security analyses of SSE/STE schemes intended for actual deployment.

On the gap between theory and practice. On MongoDB's website for QE, it is stated clearly that "*... data remains secure in-transit, at-rest, in memory,* ***in logs****, and in backups*".[15] On the other hand, if logs were included in the security model, then it would be impossible to formally prove the security of QE with respect to a snapshot adversary (because of our attacks). This leads us to speculate that there may have been miscommunication between MongoDB cryptography researchers and engineers, leading to different expectations of security.

This in turn hints at useful lessons for different actors. Cryptographers need to understand the relevant infrastructure (e.g. whether the system generates logs, and if so, what is exactly in the logs) before building cryptographic protocols on top of it. Engineers need to carefully interpret what cryptographers mean by their security guarantees, and be prepared to challenge those guarantees when they do not match reality. Above all, cross-domain communication is key in building a secure system, especially a complex one like QE.

[13] https://github.com/mongodb/mongo/releases/tag/r6.2.1-rc1

[14] It is possible for the client to send the log encryption key with the query itself, but how this key should be temporarily stored by the server only creates another key management challenge.

[15] https://www.mongodb.com/products/queryable-encryption

References

[1] Public Use Microdata Sample (PUMS). https://www.census.gov/programs-surveys/acs/microdata.html.

[2] G. Amjad, S. Kamara, and T. Moataz. Breach-resistant structured encryption. *PoPETs*, 2019(1):245–265, Jan. 2019. doi:10.2478/popets-2019-0014.

[3] G. Amjad, S. Patel, G. Persiano, K. Yeo, and M. Yung. Dynamic volume-hiding encrypted multimaps with applications to searchable encryption. Cryptology ePrint Archive, Report 2021/765, 2021. https://eprint.iacr.org/2021/765.

[4] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment*, 11(11):1715–1728, July 2018. doi:10.14778/3236187.3236217.

[5] R. Bost. $\Sigma o\phi o\varsigma$: Forward secure searchable encryption. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1143–1154. ACM Press, Oct. 2016. doi:10.1145/2976749.2978303.

[6] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 1465–1482. ACM Press, Oct. / Nov. 2017. doi:10.1145/3133956.3133980.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 668–679. ACM Press, Oct. 2015. doi:10.1145/2810103.2813700.

[8] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*. The Internet Society, Feb. 2014.

[9] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1038–1055. ACM Press, Oct. 2018. doi:10.1145/3243734.3243833.

[10] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 577–594. Springer, Heidelberg, Dec. 2010. doi:10.1007/978-3-642-17373-8_33.

[11] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 79–88. ACM Press, Oct. / Nov. 2006. doi:10.1145/1180405.1180417.

[12] Datanyze. Databases Market Share Report | Competitor Analysis | MySQL, Microsoft SQL Server, Microsoft Access. https://www.datanyze.com/market-share/databases--272.

[13] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS 2020*. The Internet Society, Feb. 2020.

[14] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In S. Capkun and F. Roesner, editors, *USENIX Security 2020*, pages 2433–2450. USENIX Association, Aug. 2020.

[15] M. George, S. Kamara, and T. Moataz. Structured encryption and dynamic leakage suppression. In A. Canteaut and F.-X. Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 370–396. Springer, Heidelberg, Oct. 2021. doi:10.1007/978-3-030-77883-5_13.

[16] P. Grubbs, A. Khandelwal, M.-S. Lacharité, L. Brown, L. Li, R. Agarwal, and T. Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In S. Capkun and F. Roesner, editors, *USENIX Security 2020*, pages 2451–2468. USENIX Association, Aug. 2020.

[17] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 315–331. ACM Press, Oct. 2018. doi:10.1145/3243734.3243864.

[18] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why Your Encrypted Database Is Not Secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 162–168, Whistler BC Canada, May 2017. ACM. doi:10.1145/3102980.3103007.

[19] Z. Gui, K. G. Paterson, and S. Patranabis. Rethinking searchable symmetric encryption. Cryptology ePrint Archive, Report 2021/879, 2021. https://eprint.iacr.org/2021/879.

[20] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, Feb. 2012.

[21] F. Kerschbaum and A. Tueno. An efficiently searchable encrypted data structure for range queries. In K. Sako, S. A. Schneider, and P. Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 344–364. Springer, 2019. doi:10.1007/978-3-030-29962-0_17.

[22] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 644–655. ACM Press, Oct. 2015. doi:10.1145/2810103.2813651.

[23] S. Oya and F. Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In M. Bailey and R. Greenstadt, editors, *USENIX Security 2021*, pages 127–142. USENIX Association, Aug. 2021.

[24] S. Oya and F. Kerschbaum. IHOP:Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In *USENIX Security 2022*, pages 2407–2424. USENIX Association, 2022.

[25] S. Patel, G. Persiano, K. Yeo, and M. Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 79–93. ACM Press, Nov. 2019. doi:10.1145/3319535.3354213.

[26] S. Patranabis and D. Mukhopadhyay. Forward and backward private conjunctive searchable symmetric encryption. In *NDSS 2021*. The Internet Society, Feb. 2021.

[27] R. Poddar, T. Boelter, and R. A. Popa. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, 2019. URL: http://www.vldb.org/pvldb/vol12/p1664-poddar.pdf, doi:10.14778/3342263.3342641.

[28] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1341–1352. ACM Press, Oct. 2016. doi:10.1145/2976749.2978401.

[29] Z. Shang, S. Oya, A. Peter, and F. Kerschbaum. Obfuscated access and search patterns in searchable encryption. In *NDSS 2021*. The Internet Society, Feb. 2021.

[30] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society Press, May 2000. doi:10.1109/SECPRI.2000.848445.

[31] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*. The Internet Society, Feb. 2014.

[32] P. J. van Laarhoven. *Simulated annealing theory and applications*. Kluwer, 1987.

A Leakage Simulation

We made the decision to switch to using simulated leakage in place of leakage from real executions because of the scalability limitations of QE. Consider the 2013 ACS dataset containing 3 million records with six encrypted field names. For a single experiment, with default settings, inserting the 3 million ACS records into MongoDB took at least 4 days (with `queryLog`) or 2 days (without) on an 8-core machine. Parallelizing across cores did not help as the bottleneck is intrinsic to the throughput of disk read/write.

Notation and Convention. For any vector or ordered set $\mathbf{v}$, we let $\mathbf{v}[i]$ denote the i-th element and let $|\mathbf{v}|$ denote the number of elements in $\mathbf{v}$. Given number of indexed encrypted fields $\tau \in \mathbb{N}$, we define encrypted field header by $\texttt{eFieldHeader} = \{(\texttt{fieldName}_i, \texttt{keyId}_i)\}_{i\in[\tau]}$, where $\texttt{fieldName}_i \in \{0,1\}^*$ and $\texttt{keyId}_i \in \{0,1\}^{128}$. With the `eFieldHeader` fixed, we use $\texttt{doc} = (\texttt{val}_1, \ldots, \texttt{val}_\tau)$ as a short-hand abbreviation of $((\texttt{fieldName}_1, \texttt{val}_1), \ldots, (\texttt{fieldName}_\tau, \texttt{val}_\tau))$, where $\texttt{val}_i \in \{0,1\}^*$, and $\texttt{doc}[\texttt{fieldName}_i] = \texttt{val}_i$, for all $i \in [\tau]$, representing a JSON-like document natively supported by MongoDB.

Simulating the Leakage. We describe briefly how we simulated the leakage and refer the details to the pseudocode provided in Figure 6. For preparation, we extract a list of unique field values across all field names from the ACS 2013 dataset, and sort those by alphabetical order. For additional length leakage, we compile a length look-up map `lenMap` using sorted unique encoding and the length of their associated text value. When simulating the field-value equality leakage, we first generate a random permutation and apply it to the (sorted) unique field values. Then we generate the leakage map by examining every document, compiling a unique tuple. When

```
GenSimulatedLeakage(docColl, eFieldHeader, lenMap, cf_max) :
{(fieldName_i, keyId_i)}_{i∈[τ]} ← eFieldHeader
for i from 1 to τ do
    v ← ExtractSortedUniqueFieldValues(docColl[fieldName_i])
    (v^i_R, rmap^i_v) ←$ GenRandomPermutation(v)
    (I^i_R, rmap^i_I) ←$ GenRandomPermutation([|v| × (cf_max + 1)])
end for
id_⊥ ← |v| + 1
L_field-value-eq, L_field-value-cf-eq, L_length ← ()
ctr ← 0
for every doc in docColl do
    idList_v, idList_vc ← ()
    for i from 1 to τ do
        v ← doc[fieldName_i]
        cf ←$ [0, cf_max]
        id_vc ← rmap^i_I[v.Index(v) × (cf_max + 1) + cf]
        idList_vc[i] ← id_vc
        if v = ⊥ then
            id_v ← id_⊥
            L_length[id_v] ← 215
        else
            id_v ← rmap^i_v[v_R.Index(v)]
            L_length[id_v] ← lenMap[v] + 215
        end if
        idList_v[i] ← id_v
        idList_vc[i] ← id_vc
    end for
    L_field-value-eq[ctr] ← idList_v
    L_field-value-cf-eq[ctr] ← idList_vc
    ctr ← ctr + 1
end for
return L_field-value-eq, L_field-value-cf-eq, L_length
```

Figure 6: GenSimulatedLeakage algorithm.

simulating the field-value-contention-factor equality leakage, we increase the size of the random permutation to take into account the contention factor, and parse the plaintext documents, for each field insertion, select a random contention factor.

In the pseudocode as in Figure 6, we assume there exists an ExtractSortedUniqueFieldValues algorithm, which takes as input a list of values and returns a set under alphabetical order; a GenRandomPermutation algorithm that takes as input a set $\mathbf{v}$, and returns $\mathbf{v}_R$ which equals to $\mathbf{v}$ permuted under some random permutation $\texttt{rmap}_v : [|\mathbf{v}|] \to [|\mathbf{v}|]$. We abuse the notation and use $\texttt{docColl}[\texttt{fieldName}_i]$ to denote all field values under some $\texttt{fieldName}_i$ from docColl.

Correctness of the Simulated Leakage. The correctness of the simulated length leakage $\mathcal{L}_{\text{length}}$ straightforwardly follows from the fact that the same length leakage map is used in leakage extraction and simulation.

Regarding the field-value equality leakage: the equivalence of the simulated leakage and the real leakage can be verified by asserting the exact match of frequency information on the unique tuples versus the plaintext distribution.

Verifying the equivalence of field-value-contention-factor equality leakage is trickier since the contention factor is randomly generated on each field value insertion. Due to the scalability issues that we discussed in Section 7, we cannot gather enough statistics for the dataset with 3M records. Instead, we check the consistency between the number of unique field-value-contention-factor identifiers appearing in the real leakage, and the simulated ones across the field names (since for $\texttt{cf\_max} = 4$ and 3 million documents, all unique identifiers will be used with high probability).

B Pseudocode of Leakage Extraction

We include the pseudocode of leakage extraction with queryLog and opLog in Figures 7 and 8, respectively. We omit the parsing and optimization details.

In the ExtractQueryLeakage algorithm in Figure 7, for simplicity, we only include the case where after insertions, for each i, there are exactly k_i find operations on unique field values for $\texttt{fieldName}_i$. However, it can be easily extended to cope with the case where some field values are queried more than once: suppose for the j-th find operation with some field name, we are trying to assign the inserted documents with some id^* by examining the field-value-equality matrix M; we find some $M(x, j) = 1$, but $\mathcal{L}_{\text{field-value-eq}}(x, j)$ is not zero, already assigned with some id, then we abort this assignment and continue with the next find operation. The id^* will be reused for the subsequent find operation with the same field name.

C Pseudocode of Our Inference Attack

The pseudocode of the modified simulated annealing algorithm is shown in Figure 9 and the pseudocode of our attack is shown in Figure 10. For simplicity, we do not include length leakage in the pseudocodes. In practice, if length leakage is given to the attacker too, the initial assignment will be picked such that all identifiers and the field values they are assigned to have the same (unencrypted) length; the new identifier-field value pairs generated in the neighbourhood subroutine neighbour() must satisfy that relation too. In addition to what we have described above, the other subroutines we choose for simulated annealing are as follows. We use $0.995T$ as our cooling scheme and $\exp\left(\frac{s'-s}{T}\right)$ as the acceptance probability.

D Experimental Data and Statistics

Overview. As in [4], we use anonymized American Community Survey (ACS) microdata at the person level [1]

ExtractQueryLeakage(queryLog) :

Extract $(\mathsf{txnid}_i, \mathtt{doc}_i)_{i\in[m]}$ from queryLog for inserting documents to eDocColl.
Extract $(\mathsf{txnid}_i)_{i\in[m+1,u]}$ from queryLog for find operations on eDocColl.
Extract every used ESC dictionary key set S_i associated with txnid_i for all $i \in [u]$.
Construct a $u \times u$ field-value-equality zero matrix M.
for i from 1 to u **do**
  for j from 1 to u **do**
    if $S_i \cap S_j \neq \emptyset$ **then**
      $M(i,j) \leftarrow 1$
      $M(j,i) \leftarrow 1$
    end if
  end for
end for
Assign every i-th find operation under some $\mathtt{fieldName}_j$, a unique id $\in [k_j]$, $i \in [m+1,u]$.
for i from $m+1$ to u **do**
  Suppose the i-th find operation under $\mathtt{fieldName}_y$ indexed by $id \in [k_y]$.
  for x from 1 to m **do**
    if $M(i,x) = 1$ **then**
      if $\mathtt{fieldName}_y \notin \mathtt{doc}_i$ **then**
        $\mathcal{L}_{\text{field-value-eq}}(x,y) \leftarrow k_y + 1$
      else
        $\mathcal{L}_{\text{field-value-eq}}(x,y) \leftarrow id$
      end if
    end if
  end for
end for
return $\mathcal{L}_{\text{field-value-eq}}$

Figure 7: ExtractQueryLeakage algorithm.

ExtractCompactLeakage(opLog, cf_max) :

Extract $(\mathsf{txnid}_i)_{i\in[m]}$ from opLog for inserting documents to eDocColl.
Extract $(\mathsf{txnid}_i)_{i\in[m+1,u]}$ from opLog for the compaction procedure.
Extract every used ESC dictionary key set S_i associated with txnid_i for all $i \in [u]$.
Construct a $u \times u$ field-value-contention-factor equality zero matrix M.
for i from 1 to u **do**
  for j from 1 to u **do**
    if $S_i \cap S_j \neq \emptyset$ **then**
      $M(i,j) \leftarrow 1$
      $M(j,i) \leftarrow 1$
    end if
  end for
end for
Assign every i-th compaction subroutine for some $\mathtt{fieldName}_j$, a unique $id \in [(k_j+1) \times (\mathtt{cf\_max}+1)], i \in [m+1,u]$.
for i from $m+1$ to u **do**
  Suppose the i-th compaction subroutine under $\mathtt{fieldName}_y$ is indexed by $id \in [(k_y+1) \times (\mathtt{cf\_max}+1)]$.
  for x from 1 to m **do**
    if $M(i,x) = 1$ **then**
      $\mathcal{L}_{\text{field-value-cf-eq}}(x,y) \leftarrow id$
    end if
  end for
end for
return $\mathcal{L}_{\text{field-value-cf-eq}}$

Figure 8: ExtractCompactLeakage algorithm.

in our experiments; specifically, we use ACS 2023 as the target dataset encrypted by QE for recovery and ACS 2022 as the auxiliary data. Since QE supports neither range query nor floating-point types, we select six text-based fields listed in Table 1, which were also studied in [4]. Note that the ACS 2012 and ACS 2013 datasets are both random samples of microdata from their respective years, and the data of the same persons are not necessarily included in both.

Data Processing. In the provided ACS 2013 raw data sets, we use `ss13pusa.csv` and `ss13pusb.csv` that contain the ACS 2013 microdata samples across the US, and extract the six attributes (also as field names) listed in Table 1, and export them to JSON files. These are all text-based fields, and the encoding length of the raw field value for each field name is the same. To demonstrate the risk of additional length leakage allowed by QE's selection of AES-CTR$ mode, we also compile the length leakage map `lenMap` using the codebook available in [1] that maps the encoded values to the length of actual text values.

General Statistics. The ACS 2013 dataset contains 3,132,795 records, and the ACS 2012 dataset contains 3,113,030 records. We take the empty string into account as one field value for computing the statistics shown in Table 1. In addition, we also plot the frequency information of the field values for every field name in Figure 11, where the x-axis represents the rank of the most-frequent (top-30) field values, and y-axis represents their frequency (i.e., counts) on a logarithmic scale.

Combined with Table 1, Figure 11 illustrates the data distribution and reveals how it affects the accuracy of our attacks. For example, the detailed racial profile RAC3P is skewed relative to the rest, with the top-30 most frequent values dominating the data distribution. It explains why even though the unique recovery rate (by unique field id) is not high for RAC3P, the recovery rate at document-level is close to 100% for all leakage profiles.

Detailed name	Encoded name	Encoded length	# Unique field values	# Unique lengths
Type III Race	RAC3P	3	100	56
State	ST	2	51	12
Place of birth	POBP	3	215	29
Place of work	POWSP	3	60	16
Class of work	COW	1	10	9
Occupation code	OCCP	4	480	91

Table 1: Statistics for ACS 2013.

Storage Overhead. Consider the 2013 ACS dataset with 3 million records with 6 encrypted field names. The

SimulatedAnnealing($\mathbf{P}, \mathcal{L}, I, rMax, \texttt{cf\_max}, \text{iter}_{\text{max}}$) :

$T \leftarrow T_0$
for i from 1 to iter_{max} **do**
  $T \leftarrow 0.995T$ ▷ *Cooling scheme*
  $r \leftarrow\$$ a random integer between 1 and $rMax$ (inclusive)
  $I_{\text{local}} \leftarrow\$$ r random elements from I (without replacement)
  $\mathbf{P}' \leftarrow\$ \text{neighbour}(\mathbf{P}, I_{\text{local}}, \texttt{cf\_max})$
  $s \leftarrow \mathbf{L}_{I_{\text{local}}}(\mathbf{P}, \mathcal{L})$
  $s' \leftarrow \mathbf{L}_{I_{\text{local}}}(\mathbf{P}', \mathcal{L})$
  if $\text{acceptProb}(s, s', T) > \text{rand}(0,1)$ **then**
    $\mathbf{P} \leftarrow \mathbf{P}'$
  end if
end for

neighbour($\mathbf{P}, I, \texttt{cf\_max}$) :

$\mathbf{P}' \leftarrow \mathbf{P}$
for i in I **do**
  $id_1 \leftarrow\$$ a random identifier in the i-th field of $\mathbf{P}$
  $\texttt{val}_1 \leftarrow\$$ a random (plaintext) field value in the i-th field
  if There are $\texttt{cf\_max}$ identifiers assigned to $\texttt{val}_1$ **then**
    $id_2 \leftarrow\$$ a random identifier that is assigned to $\texttt{val}_1$
    $\texttt{val}_2 \leftarrow \mathbf{P}(i, id_1)$
  end if
  $\mathbf{P}'(i, id_1) \leftarrow \texttt{val}_1$
  if id_2 exists **then**
    $\mathbf{P}'(i, id_2) \leftarrow \texttt{val}_2$
  end if
end for

acceptProb(s, s', T) :

return $\exp\left(\frac{s'-s}{T}\right)$

Figure 9: Pseudocode for Simulated Annealing

Attack($\mathcal{L}, \tau, \text{iter}_{field}, \text{iter}_{all}$) :

$\mathbf{P} \leftarrow\$$ a random assignment between the identifiers in $\mathcal{L}$ and the field values
for i from 1 to τ **do**
  $\text{SimulatedAnnealing}(\mathbf{P}, \mathcal{L}, \{i\}, 1, \text{iter}_{\text{field}})$
end for
$\text{SimulatedAnnealing}(\mathbf{P}, \mathcal{L}, \{1, \ldots, \tau\}, 3, \text{iter}_{\text{all}})$
return $\mathbf{P}$

Figure 10: Pseudocode for our attack.

size of plaintext dataset is 264 MB. After encryption and compaction using QE, the total storage is 8.03 GB, a 30 times increase. This is broken down into a 6.19 GB encrypted document collection and a 1.84 GB compacted metadata ESC (with ECOC dropped). The accumulated (compressed) opLog is approximately 10 GB and 29 GB when exported, and the size of query log is about 497 GB.

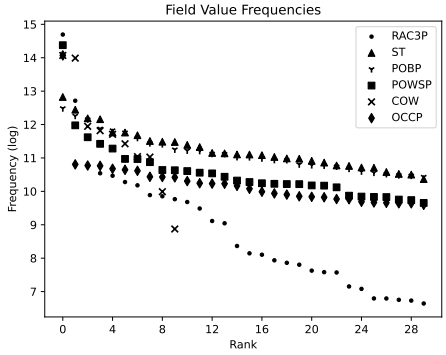

Figure 11: Frequencies of field values.

E Additional Experiments

In this section, we present additional experimental results. The experimental results on opLog with 30K documents and 300K documents can be found in Figure 12. The experimental results on queryLog with 3M documents, and 100 and 500 queries can be found in Figure 13.

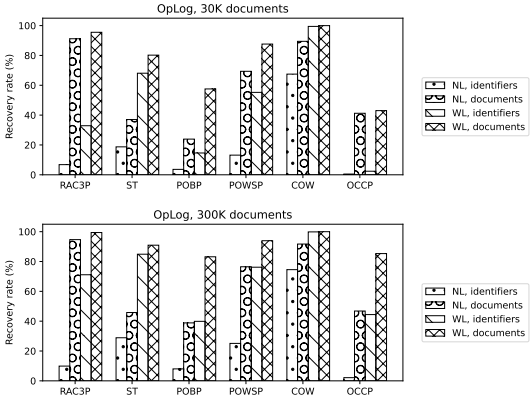

Figure 12: Experimental results on opLog with 30K and 300K documents.

The attack follows the expected behaviour: it works better when more queries are made and when the database is larger; the plaintext recovery rate on different fields vary greatly due to having different field value distributions.

For the attacks against opLog and queryLog, we observe higher plaintext recovery rates when the database contain more documents. This is to be expected as larger databases contain more meaningful statistical information. Furthermore, for the attacks against queryLog, we see that 100 queries (either uniform or Zipf) are not enough to recover field values in some fields such as POBP. This is mainly because the number of queries is not large enough to cover a large proportion of field val-

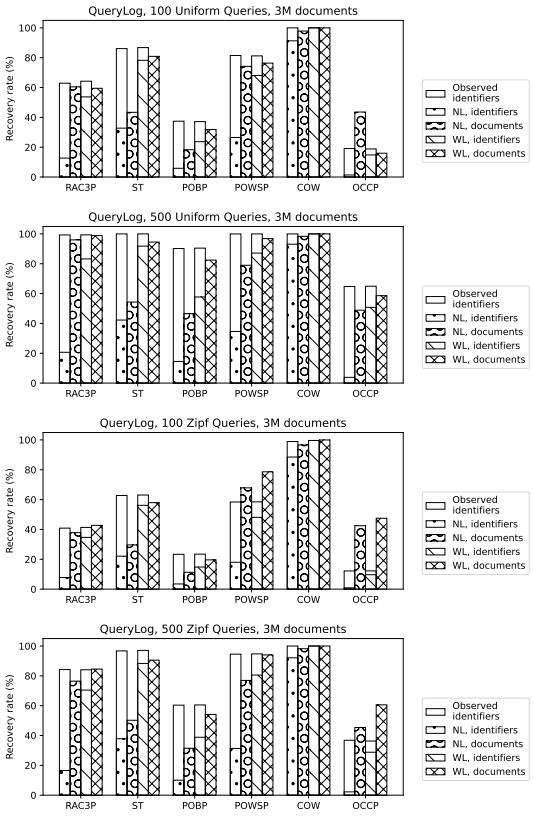

Figure 13: Experimental results on `queryLog` with uniform and Zipf query distributions and 100 and 500 queries on each field.

ues. On the other hand, our attacks perform noticeably better when 500 queries are issued per field. This can be explained by the fact that the queries cover significantly more field values.

F Sample Output of `queryLog` and `opLog`

We include raw output samples of `queryLog` (Figure 14) and `opLog` (Figure 15) after inserting a small set of documents in the processed ACS 2012 dataset. This `queryLog` sample output records a find operation: line 14 indicates the find operation was performed on `ESC`, with `ESC` dictionary key listed at line 18 under "base64" field name. In `opLog`'s sample output, at line 2, "op": i, indicates this is an insertion operation to the `ESC` collection (specified at line 3), with `ESC`'s dictionary key listed at line 13, and inserted value listed at line 19.

```
{
  "t": {
    "$date": "2022-10-01T23:11:46.417+02:00"
  },
  "s": "I",
  "c": "COMMAND",
  "id": 51803,
  "ctx": "FLECrud-3",
  "msg": "Slow query",
  "attr": {
    "type": "command",
    "ns": "acspum.enxcol_.2012.esc",
    "command": {
      "find": "enxcol_.2012.esc",
      "filter": {
        "_id": {
          "$binary": {
            "base64": "zwRc/i0NKENwvZoK1rRF32WORy2
                nVgK4c1jY6cyB8ns=",
            "subType": "0"
          }
        }
      },
      "singleBatch": true,
      "lsid": {
        "id": {
          "$uuid": "b54443de-6079-4291-912b-108d0eae
              38b5"
        },
        "uid": {
          "$binary": {
            "base64": "47DEQpj8HBSa+/TImW+5JCeuQeRkm
                5NMpJWZG3hSuFU=",
            "subType": "0"
          }
        },
        "txnNumber": 2,
        "txnUUID": {
          "$uuid": "e0f24e1e-f6eb-4fe0-ba18-ac278006
              8f0b"
        }
      },
      "txnNumber": 0,
      "autocommit": false,
      "$db": "acspum"
    },
    "planSummary": "CLUSTERED_IXSCAN",
    "keysExamined": 0,
    "docsExamined": 2,
    "cursorExhausted": true,
    "numYields": 0,
    "nreturned": 0,
    "queryHash": "740C02B0",
    "planCacheKey": "740C02B0",
    "queryExecutionEngine": "classic",
    "reslen": 237,
    "locks": {},
    "readConcern": {
      "level": "local",
      "provenance": "clientSupplied"
    },
    "storage": {},
    "protocol": "op_msg",
    "durationMillis": 0
  }
}
```


Figure 14: Raw queryLog.


```
{
  "op": "i",
  "ns": "acspum.enxcol_.2012.esc",
  "ui": {
    "$binary": {
      "base64": "KjjJhamdQQGWJ5WAoN0pLg==",
      "subType": "04"
    }
  },
  "o": {
    "_id": {
      "$binary": {
        "base64": "dNwnzydHRFS/kOoeX4GwJa0g6JEoMTYP3
            +1F0p6Ayos=",
        "subType": "00"
      }
    },
    "value": {
      "$binary": {
        "base64": "omWtaSSemsfK1x3WjdVq+mGwqRBGK6X9
            PmkKvhM77Yg=",
        "subType": "00"
      }
    }
  },
  "o2": {
    "_id": {
      "$binary": {
        "base64": "dNwnzydHRFS/kOoeX4GwJa0g6JEoMTYP3
            +1F0p6Ayos=",
        "subType": "00"
      }
    }
  },
  "stmtId": 0
},
}
```


Figure 15: Raw opLog.