OpAwesome: The Good, the Bad, and the Fuzzy in the Secure Database Landscape

Sophie Hilgard, Wilson Qin

May 29, 2018

1 Introduction to Secure Databases

1.1 Why do we need Secure Databases?

Cloud data storage and computation are increasingly important for both personal users and corporations, driven by a few industry trends: the quantity of data increasing, the cost of cloud storage falling, and users increasingly expecting to access their data from anywhere. However, several high profile data breaches over the past few years have made it apparent that storing data on a vulnerable cloud is no different than handing that data directly to the adversary. At one extreme, users could securely encrypt the entirety of the data they would like to store, but this prevents the cloud service provider from doing any computation other than storing the data and shipping it back to the user - on this front cryptography systems based on Fully Homomorphic Encryption (FHE) are still currently performance prohibitive. Secure databases attempt to bridge the gap between fully functional data storage and fully secure data storage. Current state of the art secure databases claim to be within $5X$ latency of commercial plaintext databases on small datasets with support for most query types [Fuller et al., 2015].

1.2 Design Space Elements for Secure Databases

Threat Model Assumptions Defining the security of protected databases depends on the assumptions being made about which parties are and are not trusted and the actions available to the adversary. In many models, including most of the analysis and discussion that follows, we assume that the client and querier are the same party (the data owner) and that the server is untrusted. This requires primarily protecting the data at rest on the server. That is, we assume that the adversary cannot issue queries or see the results returned from a query but can see all of the data in its current state on the server at any point in time. We additionally assume that this adversary has access to some outside resources, such as potential column names or approximate distributional information, either from a stale version of the database or a related database obtained elsewhere.

In an alternative threat model with a trusted server but untrusted query eavesdropper, we instead attempt to limit the information that can be gained from data in motion. Even stronger models assume that the querier can itself be compromised, allowing insertions or modifications on the database (but typically not arbitrary select/project queries, as we may often assume that queriers are limited to data for which they have specific keys, an important topic in database protection which we will not discuss here).

The security of the database is then modeled in terms of leakage, the quantity or type of information that can be extracted from the database by the adversary. Fuller et al. [2017] describe the levels of database leakage as structure, identifiers, predicates, equalities, and order in increasing order of severity. Structure of the database is generally assumed to be known to the adversary, so structure here refers to the structure of individual entries, which could be masked with padding if desired. Leakage of data object identifiers reveals nothing else about the object itself but allows adversaries to recognize when an item returned by a previous query is queried again. Predicate leakage allows adversaries to additionally gain some information about some quality of the data object through the structure or knowledge of a query clause that resulted in returning that object. Equality and order leakage are the basis of most attacks in the secure database literature: they allow adversaries to determine which data objects share the same plaintext value and the ordering of those plaintext values, respectively. Eventually, leakage samples could lead to the adversary learning the distribution of sensitive attributes.

Design Parameters and Tradeoffs The design decisions that make secure databases different from standard database designs are related to tradeoffs between security and some measure of performance.
Performance measures include:

- computation (on server and client)
- data movement (I/O's across memory hierarchy, disk, and network)
- space (memory and persistent storage on server and client)
- query latency

The primary question being asked is: what is the query processing model? Which parts of query processing should take place at the client and server respectively? Once this has been determined, we must also ask what auxiliary structures (encryptions, precomputations, indices) are necessary to ensure performance while maintaining security.

Computation at the client is generally more secure and requires less cryptographic resource use, as the client is trusted and could therefore use traditionally optimized query processing techniques for operating over plaintext. Computation at the server is either as secure (assuming a CPA-secure encryption scheme) or dramatically less secure and generally requires more computationally intensive techniques for cryptography requirements. For example, secure range queries on encrypted data require full sequential scans instead of metadata-informed range queries that can be done on plaintext; partially homomorphic

addition requires significantly more operations than regular plaintext addition. However, if the time and space of the server is assumed to be cheaper than the computation and space of the client, this can be a worthwhile tradeoff.

One example of a common query processing design decision concerns how much batch processing to support - for instance, does the system answer search queries one-at-at-time versus answering queries in batches with a shared scan approach to query processing. For analytics database engines it can be perfectly appropriate to delay a single query, as long as every batch of k queries runs together on one pass through the data, if it improves overall system throughput. Thus, a user's expectations can also shape what cryptographic treatment of data is supportable.

2 Legacy Secure Database Systems

2.1 CryptDB

In [Popa et al., 2011], the authors design and implement CryptDB, the first widely-adopted cryptographic database, built on top of the Postgres DBMS with the idea of maintaining as much of the original SQL functionality as possible while providing minimum security guarantees. The system works by using modern SQL engines as-is over encrypted values which are mapped from plaintext values sent by the client through a CryptDB proxy server and CryptDB UDFs. CryptDB is able to support a wide variety of selection and projection operations, as well as minor computation and notably supports both updates and joins, making it unique in this field.

However, CryptDB is far from offering full CPA security. In the most extreme case, if the proxy server is compromised, then all data at that location could be exposed in the clear, due to predicate evaluation happening at the proxy. In the more likely scenario of a compromised SQL engine, security guarantees will depend on the type of encryption used to secure a given field, as we will see below.

2.2 What CryptDB does well

In the interest of supporting traditional SQL operations over encrypted data, the CryptDB system encrypts each column with several encryptions of increasing security, in what they call "onions" [Popa et al., 2011]. Several of the encryption schemes provide CPA-security against a compromised server and therefore have no leakage with respect to equality or order. These are RND, a randomized cipher encryption using AES/Blowfish with a random initialization vector (IV); Pallier for homomorphic addition; ElGamal for homomorphic multiplication; DET, a deterministic AES/Blowfish encryption, when applied to unique values; and SEARCH, a scheme based on [Song et al., 2000], when applied to exact string matches.

CryptDB begins to encounter leakage concerns when it must rely on its less-secure encryption schemes, DET for non-unique values, which will leak equality, SEARCH for substring matching, which leaks substring equality (number of duplicate words), ADJ-

JOIN, which leaks equality, and OPE (order preserving encryption), which leaks equality and order.

[Naveed et al., 2015] details the empirical weaknesses of the DET and OPE encryption schemes in the CryptDB system through inference attacks on OpenEMR datasets. While Naveed et al. [2015] make some valid points, which we investigate below in the limitations of CryptDB, the authors of CryptDB argue in [Popa et al., 2015] that access to the encryptions as used in the paper would never be available if the system were used correctly.

The main argument of Popa et al. [2015] concerns the action of tagging columns as 'sensitive', in which case a properly configured CryptDB system should refuse to use any of the weaker encryption schemes on such sensitive attributes. In this case, the attacks of [Naveed et al., 2015] would be avoided. The authors further argue that many of the columns on which the inference attacks are performed do not actually exist in OpenEMR or would not be queried in actual settings as they are in the attack paper.

Additionally, the attacks are heavily dependent on the quality of auxiliary datasets. If such rich, well-matched datasets do not actually exist, CryptDB would perform much better than in this example. While the likelihood of this attack model is difficult to codify, it remains a challenge to solutions like CryptDB to take into account some measure of the availability of such datasets.

2.3 Attacks against and Weaknesses of CryptDB

The attacks used against CryptDB in [Naveed et al., 2015] are attacks of a "general server adversary" against data at rest on the server. While the authors detail four different specific attacks, they all fall under the general category of Frequency Analysis, in which the frequency of deterministically encrypted or order-preserving ciphertexts are compared to either the frequency of known data demographics or the order of a known plaintext message set. Below we describe the attacks and improvements that could be made to CryptDB to increase security.

2.3.1 Attacks against CryptDB

The four attacks used against CryptDB in [Naveed et al., 2015] are as follows:

- Frequency analysis: this attack uses an auxiliary dataset with columns that are expected to be similarly distributed to columns in the target dataset. By matching the frequency of occurrences in the plaintext dataset to a deterministically encrypted dataset, the ciphertexts can be matched to their plaintext counterparts. The success of this attack depends on the correlation between the auxiliary and target dataset.

- ℓ_p-optimization: this attack is simply a more sophisticated version of the above frequency analysis, in which histogram values can be matched with varying norm distance metrics rather than simply on count.

- Sorting attack: the worst case of this attack on OPE-encrypted columns happens when every possible value of the message space is sorted in an encrypted column.

This is notably a problem because a DBMS uses sorted intermediate results for answering queries with range joins. This attack combined with knowledge of the message space, reduces to an easy task of mapping each ciphertext to its plaintext value.

- Cumulative attack: this attack seeks to be slightly more sophisticated than the above, allowing for missing values by building an empirical cumulative density function of ciphertexts and selecting the plaintext values that minimize the difference between this empirical CDF and an auxiliary plaintext column which is expected to have a similar distribution.

The success of these attacks is entirely dependent on the ciphertexts leaking equality or ordering. Below we address alternatives to the DET and OPE encryption schemes that maintain all or some of the functionality while increasing security.

2.3.2 Alternatives for DET

With respect to deterministically encrypted columns, there is an easy fix, which the CryptDB authors seem to rightly argue would have been used if any of the deterministically encrypted columns had been considered sensitive. Instead of using AES/Blowfish to encrypt a given plaintext, they could instead use SEARCH, which still allows for exact matches and is CPA-secure without indexing, but is more space-intensive.

SEARCH works by using the schemes based on pseudorandom functions and pseudorandom generators developed in [Song et al., 2000]. The authors allow for exact matching (with the possibility of some false positives) using roughly the following construction:

- For any "word" (here, a value of a column, assumed to be padded to some length n, an encryption of the word itself $X_i = E_{k''}(W_i)$ is used as input to a pseudorandom function $f_{k'} : \{0,1\}^n \to \{0,1\}^\lambda$ with some random key k' to generate a new key k_i for each W_i.

- A pseudorandom generator is used to create a stream of values $S_1, , S_\ell$, each of length $n - m$ for $m < n$ where ℓ is the number of values to be encrypted.

- These are used as inputs to a new pseudorandom function family keyed by the keys generated above $f_{k_i} : \{0,1\}^{n-m} \to \{0,1\}^m$ and then concatenated with the S_i to form $T_i = \langle S_i, f_{k_i}(S_i)\rangle \ \forall i$.

- The ciphertext is then computed as the XOR of this value, which will be indistinguishable from random uniform due to the security of the PRG and PRF, and the encryption X_i of the original W_i, leading to a provably CPA-secure ciphertext $C_i = T_i \oplus X_i$.

To search for a value W_i, the client sends $X_i = E_{k''}(W_i)$ and $k_i = f_{k'}(X_i)$. The server can now do a sequential scan to find if there are any data points for which $C_i \oplus X_i = \langle s, f_{k_i}(s)\rangle$.

In reality, the X_i must be split into its first $n - m$ and last m bits to allow for decryption, but this construction provides the general intuition. We can see that there is

some probability $\approx \ell/2^m$ of a false match, as with some probability, the psuedorandom function evaluation for a different key k_i' matches the pseudorandom function evaluation for the true key k_i. In the CryptDB attacks, access pattern attacks were not considered, but we note that if it is a concern that the server knows where matches occur, this parameter m can be made smaller, such that there are more false positives, again leading to more client-side computation but greater security.

2.3.3 Alternatives for OPE

Traditional OPE Order-preserving encryption is more difficult to secure while maintaining the properties that make it desirable. Some recent OPE constructions have created schemes that are more robust to query-based attacks but seem unlikely to help in the case of a compromised server. Modular OPE (here in conjunction with monotone minimal perfect hash functions, a function family which maps every value in a known message space to its ordered index in that message space), studied in [Boldyreva et al., 2011], adds a predetermined random value to all order-preserving encryptions, making it more difficult for an adversary who sees queries to determine the specific value or value window of a given result set. However, this encryption scheme still suffers from the same inference attacks as non-modular OPE with respect to distance windows (that is, inferring an estimate of the distance between two values in a given result set) in a query setting and provides little added protection against the type of CDF attacks seen in [Naveed et al., 2015] (one can imagine simply rotating the ciphertext space until the frequencies find a good fit. While this is certainly an added layer of complexity, it is not a convincing security measure for privacy protection.)

Similarly, Popa et al. [2013] create a unique mutable OPE scheme based on balanced B-trees, in which inherently unordered ciphertexts are stored in order of their corresponding plaintexts in a B-tree and accessed through the encoding of their root-to-leaf paths. The scheme must be mutable as rebalancing the B-tree requires changing a limited number of these encodings on each insertion. The authors prove this to be the first OPE encryption scheme that reveals no additional information beyond the ordering of the plaintexts, however in the compromised server setting this again does not protect against frequency or CDF attacks.

Approximate or Keyword-based OPE We note that the leakage of [Popa et al., 2013] is the least that can be achieved with strict OPE and is still insecure in our attack model of an untrusted server. This implies that methods must be used which either require client-side computation or do not fit the traditional linear-time scan OPE construction. Di Crescenzo and Ghosh [2015] develop an approach for secure range queries based on storing for each value v in the message space the 'upper-rank', the number of items in the message space less than or equal to v and the 'lower-rank', the number of items in the message space strictly smaller than v. This is called the rank database and is used to access corresponding values of a payload database. When initializing and querying these databases, a random shift is additionally applied to the ranks, further obscuring the data accesses. If these databases are suitably encoded, the frequency-based attacks

above are no longer possible. However, when using only a client-server protocol (no third party computation) the server time complexity of this scheme is linear in the size of the database, which is generally unacceptable.

Another approach is "bucketization", which trades off some degree of client computation for added security. At the extreme of client computation, the server could ship the entire securely encrypted column or database for a range query to the client and perform the operation there with no leakage. On the other extreme, using OPEs that reveal order allow all range filters to happen at the server but have high information leakage. Bucketization index approaches allow for intermediate solutions in which a range query returns some false positives to the server in return for added security.

In general, an index is created which partitions a new index column into some set of M buckets corresponding to the values of a column. Then the column itself can remain securely encrypted (e.g. by randomized AES), and range queries that would normally operate over the column operate instead over the bucketed index. If the number of buckets is less than the number of values in the message space, this will necessarily result in some additional records being returned to the client. Hore et al. [2004] provide an algorithm which achieves maximal entropy per balanced bucket while ensuring that the client only processes at most k times more data than it would have in the ideal situation with as many buckets (in which case the buckets would have items in strict order and any given value would be contained in a single bucket). Their algorithm, *controlled diffusion*, begins with this ideal-performance M-bucket partition and creates a new M-bucket partition by probablistically moving values from one bucket to another to increase the entropy of each bucket. However, it limits the number of buckets k that any one value can reside in so that range queries including that value will never increase client workload by a factor more than k. Empirically, the authors show that the extra work performed by the client is often much less than k times the ideal workload, as some ranges will experience fewer false positives than the worst case. (For example, requesting the full message space would clearly result in no false positives.)

2.3.4 Other areas in which CryptDB could be improved

Steady State Onions As mentioned briefly above, CryptDB uses "onion" layers to encrypt higher leakage encryptions with more secure encryptions. However, once a higher level encryption is removed to allow for operations on DET or OPE, the column is left in the less-secure state as its "steady state onion". While any decryption of a sensitive column to DET or OPE can be problematic if we assume a compromised server, if we prefer to minimize the amount of time the server is in this state while still preventing encryption "thrashing", where we might re-encrypt a column only to have to decrypt it soon after, CryptDB could use statistics on historical query distribution to intelligently decide when re-encryption is worth the computational effort.

User responsibility In [Popa et al., 2015], the authors place a good deal of responsibility on the users to properly tag sensitive columns. Either the authors of [Naveed et al., 2015] explicitly ignore these instructions, or the program could do more to provide

leakage warnings based on potential query profiles. For example, it should be possible to output a warning along the lines of "with the specified query and sensitivity profile, the chosen encryption scheme is likely to reveal x to an adversary with access to y additional information, would you like to proceed?"

More explicit warnings of this type are easier for a user to understand and could be followed up with alternative solutions, for example: "Using a partitioned index instead of order-preserving encryption reduces the risk of inference attacks by x and is expected to increase client computation time by at most y. For full data privacy we recommend performing order operations at the client." Because timestamps are usually considered to have high entropy and low sensitivity, timestamps could likely be recommended in many scenarios to filter results to a smaller result set for client computation.

3 Optimizations in Legacy Databases

Aside from the leakage concerns detailed above, CryptDB also fails to support most analytical queries and is not optimized to minimize space and computation costs. Authors of the original CryptDB work have since explored extensions to the old approach, two of which we address below.

3.1 Monomi

Monomi, developed in [Tu et al., 2013], uses the CryptDB system directly but acts as a database designer and query optimizer, selecting the appropriate encryptions for each column based on a proposed query workload and space constraint. To allow for additional queries to be answered on the server-side, Monomi also selects specific multi-attribute quantities to precompute and store encrypted. For example, Pallier encryptions allow only addition while ElGamal encryptions allow only multiplication, so if we expect the query workload to involve adding two columns and multiplying by another, it may make sense to do this computation in plaintext at the client once and store the computed values encrypted on the server. In many cases, the encryption schemes prevent full computation of a query at the server, and in these situations Monomi optimizes to do as much as possible at the server before shipping the results back to the client to finish the computation.

Given a dataset and a representative set of queries, Monomi proceeds in three steps to choose the optimal database structure:

- For each query, Monomi determines what encryption scheme(s) for every column (original or derived) would allow the query to be executed at the server.
- Monomi then uses a query execution planner to determine for each query and possible database structure (based on combinations of the encryption schemes determined above) how much computation can be done at the server and how much will be pushed to the client.

- Monomi then uses a cost model to choose the encryption scheme that allows for the lowest cost in terms of a combination of server computation, data transfer, and client computation.

This process can also be extended using Integer Programming to select the best encryption subject to additional constraints. In the paper, the authors limit this constraint to total database space, but additional variables and constraints could be added to the problem to restrict server computation, client computation, data movement, or leakage.

3.2 Arx

Arx extends CryptDB by directly addressing CryptDB's prior issues related to joins over DET and OPE, using custom encrypted indexes. While Arx's primary storage engine is still a DBMS (MongoDB) for directly storing encrypted base data, they introduce new custom encrypted data structures for the purpose of selection (searching for pointers to qualifying records in the underlying DBMS). In this way, Arx separates two discrete query processing responsibilities: select operations run in the custom indexes, and the fetching qualifying records happens in the DBMS.

Arx supports two custom tree index variants (Arx-Eq and Arx-Range). Arx-Range aims for a history-independent data structure design and uses Garbled Circuits (GC) at each node for tree traversal to avoid an interactive selection process. GCs are chained together, so that a path from root node to leaf can be visited in a single request. Much like a B+tree, the custom index guarantees that point queries (equality matches) take a logarithmic number of node accesses. However, Arx-Range must destroy and recreate each node after each visit, for security purposes (multiple fence keys are constants baked into each GC). This strict protocol, when coupled with the history independence properties, means that a value's ciphertext cannot be distinguished solely by its position(s) in a data structure, a dramatic improvement over OPE.

Meanwhile, Arx-Eq addresses the previous leakage problems of using deterministic encryption schemes on non-unique attributes, by reducing encryption schemes of non-unique values to the encryption of unique values ($EQUnique$). Each index specifies that the client maintains a mapping table for creating indistinguishable ciphertexts, via disambiguation of duplicated occurrences of an attribute value. The mapping table assigns each duplicate a sequence number of its occurrence. Therefore, given two different occurrences of duplicated values v, they must have sequence numbers m and n, such that $m \neq n$; they no longer collide as duplicates in a new hash space H: $H(EQUnique(v), m) \approx H(EQUnique(v), n)$.

The authors also note that Arx indexes can be extended with Segment-Tree like properties, where summary aggregate information over all child nodes (such as counts, sums, etc) can be stored within each node as well. This is an example of precomputation being used to accelerate aggregation queries instead of a lengthier traversal and summations over Pallier ciphertexts. Since for any desired range, simply querying the covering set of ancestor nodes is sufficient to collect an aggregate such as partial-sums.

4 State of the Art Custom Secure Databases

We recap problems with the legacy systems approach to designing secure databases, and also dive into two notable examples on lines of cryptographic work that focus on leakage reduction and custom secure data structure design.

Inherent Problems of Legacy Systems Many of the state of the art experimental implementations of secure databases have moved away from building on top of legacy systems [Fuller et al., 2017]. Legacy systems are inherently limited in their structure and computation methodologies, leading to necessarily higher leakage than custom systems built on secure multiparty computation techniques.

By relying on an underlying standard DBMS, legacy systems assume the DBMS exposes no practical security weaknesses (usually when reasoning about an adversary with server snapshot access). However, in practice this is hard to ensure, as systems design aims to strike a tradeoff between Consistency, Availability and Partition-Tolerance (as per Brewer's popularized CAP theorem) Gilbert and Lynch [2002].

Even common database settings being switched on, could break many security assumptions. For instance, maintaining write-ahead-logs (WAL) allows for determining the order of transactions for the purpose of replay during database recovery - thus leaking ordering of ciphertexts to the adversary who, worse, may already have a priori knowledge of the access pattern (e.g. adversary knows when a in-order bulk insertion is happening). On the other hand, diagnostic features such as enabling query statistics allows for database administrators to debug the data schema for underlying causes of slow queries. Diagnostics can leak historic timing information to an adversary on the untrusted server, as a common default is to store the diagnostic data in plaintext.

Grubbs et al. [2017] highlights additional practical vulnerabilities. We take special note that it is accordingly hard to isolate an adversary with server snapshot access from historical data pertaining to the workings of the DBMS itself. Otherwise practical configurations of a DBMS could actually upgrade a snapshot adversary to a quasi-persistent adversary by leaking the adversary a retroactive knowledge in the snapshot. Below we discuss some unique custom solutions with provably less leakage than CryptDB and other legacy systems.

Kamara-Moataz 2016 The system developed in [Kamara and Moataz, 2016] is unique in that it does not build a cryptographic system on top of existing DBMS models. Rather, the authors come up with a unique plan based on structural encryption, in which they create several encrypted maps out of the original tables and then use structured operations over these maps to recover the elements corresponding to select, project, and cross product operations. Information leakage is limited to access patterns, result set sizes, and some information about attributes, preventing the attacks based on equality and ordering detailed in our discussion of CryptDB. However, the system has not actually been implemented, and the necessity of the maps created upon initialization prevents later updating of the database.

Blind Seer The Blind Seer (BLoom filter INDex SEarch of Encrypted Results) system developed in [Pappas et al., 2014] uses secure two-party computation based on Yao's garbled circuits and Naor-Pinkas oblivious transfer to traverse an index structure consisting of encrypted keyword-based bloom filters. Each bloom filter at a node is hashed with all keywords (for example "name:JEFF") on which the client might filter for all of its children. Thus, the bloom filter will return a positive result at any node which is the ancestor of a leaf record which satisfies the Boolean query against which the bloom filters are checked. The evaluation at each server-supplied index node is the product of a secure function evaluation of a client-supplied circuit that represents an SQL-query as the conjunction of individual Boolean keyword queries. Due to the security of the function evaluation, no information is leaked except the eventual result, 1 or 0, and the resulting tree traversal path. If the bloom filter returns a positive result at some node, the protocol proceeds by checking every child of that node as well. If the bloom filter returns a negative result, there are no matching results in the corresponding sub-tree.

Using this single index for the entire database, Blind Seer can evaluate a wide range of queries including arbitrary Boolean queries, ranges, and negation queries (which are often not supported by secure databases). However, insertion is not well-optimized, based on a server-held array log that is occasionally flushed to the index, requiring a full rewrite of the structure. Additionally, joins are not supported, as the index for the entire database must be stored as a single structure.

5 Ideas for the Future of Secure Databases

Recent developments since CryptDB suggest a rich design space for custom index structures. Continued focus in this area has shown structures and access patterns that derive their security from a growing diversity of cryptographic primitives spanning Garbled Ciruits, Private Information Retrieval, Oblivious Transfer, and far more as seen briefly above [Fuller et al., 2015].

Naturally, economics of computation and storage drive the tradeoffs of where query processing gets done and where data is stored. We identify an opportunity for storing encrypted base data on the untrusted cloud server as remote storage continues to get cheaper, and notice a sweet spot for maintaining lightweight index structures on the trusted client, if we allow for some disk or memory usage at the client side.

6 opAwesome and Future Work

We rely on the insight that in the distributed computing environment today, access patterns over a single ciphertext that is the encryption of an entire dataset leak far less information than similar access pattern over ciphertexts for every datum per attribute for the same dataset. However, the main problem with encrypting large blocks of data is twofold:

- Query processing steps could lose granularity of access - single value ciphertexts can

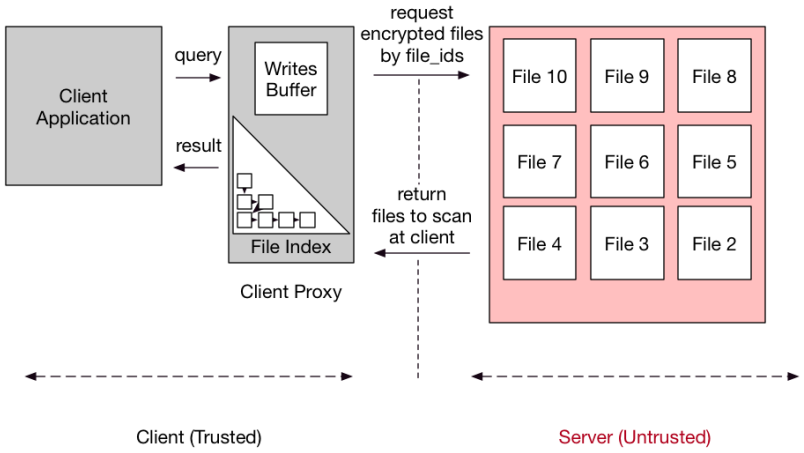

Figure 1: opAwesome is a LSM-Tree structure with remote files, which maintains a lightweight index structure at the trusted client, and delegates encrypted base data storage to the untrusted server.

be compared easily for equality, however a ciphertext of a file is far less useful for comparison (but harder to distinguish for an adversary).

- Large encrypted data blocks need to be transferred between the untrusted parts of the system and the trusted computing base. This can present a major I/O cost, especially on the network.

To this end, we describe our preliminary work on opAwesome, a secure key-value database which has a primary index that is a modification of a Log Structured Merge (LSM) Tree. Our design seeks simplicity with guarantees of CPA security, by treating entire files as ciphertexts. We trade off larger sizes of network communications, and trade off space and computation between the trusted client and untrusted server.

6.1 Division of Labor: Trusted Client and Untrusted Server

opAwesome works across a trusted client and untrusted server. The client's responsibility is to buffer new write queries and maintain a file index which contains metadata about files (such as file key ranges, and encryption instances used). The untrusted server is tasked mainly with receiving and storing these immutable files created by the client.

6.1.1 Files

New file creation is the responsibility of the client. A file can be created by new writes, or as the result of several old files being merged together (to consolidate repeated writes to the same key). The contents within it are no longer mutable when it reaches the untrusted server. The only time a file's contents will be touched and re-written is if that file later returns to the client as a participating file of a client-run merge operation, which we discuss later.

The untrusted server is tasked with receiving and storing these immutable files created by the client. At any point, the client could request a file back. To keep read accesses (and ensuing size of server-client communications) bounded, we seek to parameterize the responsibility of merge operations between the client and the server as a tuneable design option.

Files are typically at least a multiple of disk-page size large to make the network communication worthwhile. State of the art embedded key-value stores (non secure) keep files between 2MB and 160MB by default. Effectively this means for small keys and values, write and merge patterns must repeat themselves exactly (same keys, values, ordering, and over long periods) before the ciphertext of an entire file would repeat. We note that this would present a problem if a malicious user with use of the trusted client colludes with the server adversary to control the client's write buffer and file contents. On any given file, CPA-security is guaranteed against a snapshot adversary, because any keys and IVs are held by the client itself, as long as the client uses a strong deterministic encryption scheme (which we make the default).

6.2 File Size Considerations

Because files are by default encrypted at the entire file level, the smallest unit of remote server access is one file transfer over the network between server and client. Therefore, tuning of file size is an important consideration because it also affects the granularity at which reads, writes and merges can happen.

6.3 Log Structured Merge (LSM) across local-remote context

For the client file index, opAwesome leverages a modified Log Structured Merge (LSM) Tree to create a division of labor on three crucial operation types O'Neil et al. [1996]. The three tasks that occur in the database are user reads and writes, and system merge operations.

6.3.1 Read Queries

For reads, the client-side file index allows local select operations, producing a pruned list of file pointers to retrieve from the remote server as a separate fetch operation.

The encrypted files, returned from the server to the client as the result of a fetch operation of a read query, are then decrypted accordingly by the client for query evaluation, before the final results are returned to the user.

6.3.2 Write Queries

For writes, the client-side has a writes buffer which is serialized, encrypted and committed to the server as a file after the buffer fills. The file index, is updated accordingly to indicate the existence of this new file on the server side.

We do not discuss delete queries as their own class of query, but in practice it is a special case of a write, except that on a delete, there is no associated value to set a key to.

6.3.3 System Merges

The third operation type are periodic merges triggered by the system as a background process. The goal of the merge is for the system to keep the read amplification (ratio of total bytes scanned compared to relevant bytes) and network communication size bounded. We note that without merges, total dataset size at the file server can balloon, because a key can be the target of multiple writes as the associated value is written (including deletes). Merges are triggered by the client, because the client has a full overview of the file metadata from the local file index.

6.3.4 Client choice: different file, different encryption scheme

In our preliminary demo, we allow the client to specify CPA-secure deterministic encryption schemes, and setup various instances even across cipher types. As illustrated in 2, 3, 4, opAwesome supports storing different files under different encryption formats on the untrusted server.

6.4 Progress

We plan to demonstrate that opAwesome can be used with any strong deterministic encryption scheme. Furthermore we are investigating a process for opAwesome to tune its data access patterns to maintain CPA-security across the presence of both unique keys and non-unique keys. Currently opAwesome only requires lightweight indexing at the trusted client, but we also are exploring support for helper data structures like Bloom Filters to accelerate point queries on the untrusted server. One future direction is to delegate more opAwesome merging operations to run securely in the background of the untrusted server (where compute is much cheaper), however this requires more exploration.

While the preliminary design for opAwesome only supports a single server and single client, we note that opAwesome has future flexibility to integrate with other distributed cryptographic schemes.

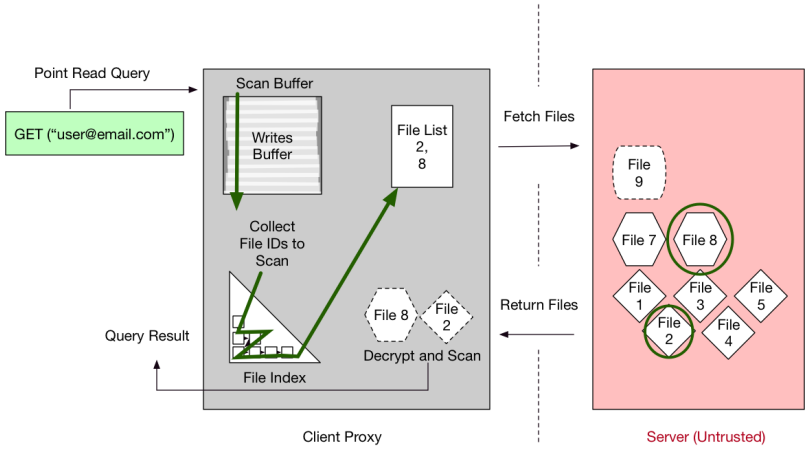

Figure 2: For reads, the client-side file index allows local select operations, producing a pruned list of file pointers to retrieve from the remote server as a separate fetch operation. The encrypted files returned from the server are then decrypted accordingly on the client for query evaluation, before the final results are returned to the user.

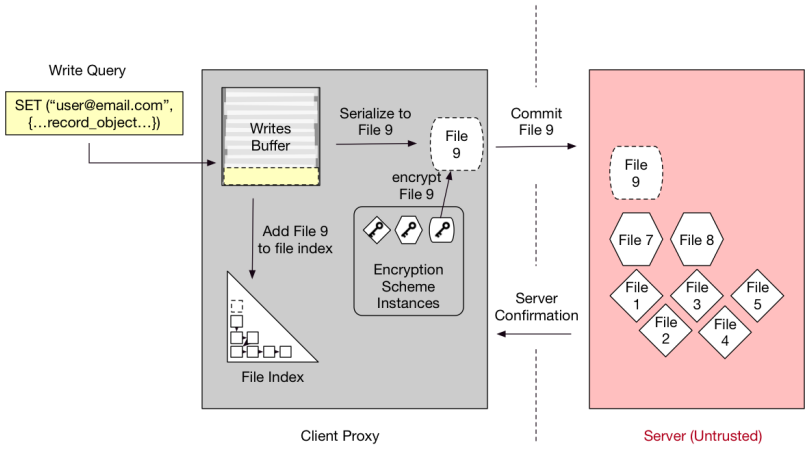

Figure 3: For writes, the client-side has a writes buffer to take initial write queries. After the buffer is full, it is serialized, encrypted and committed to the server as a file. The file index, is updated accordingly to indicate the existence of this new file on the server side.

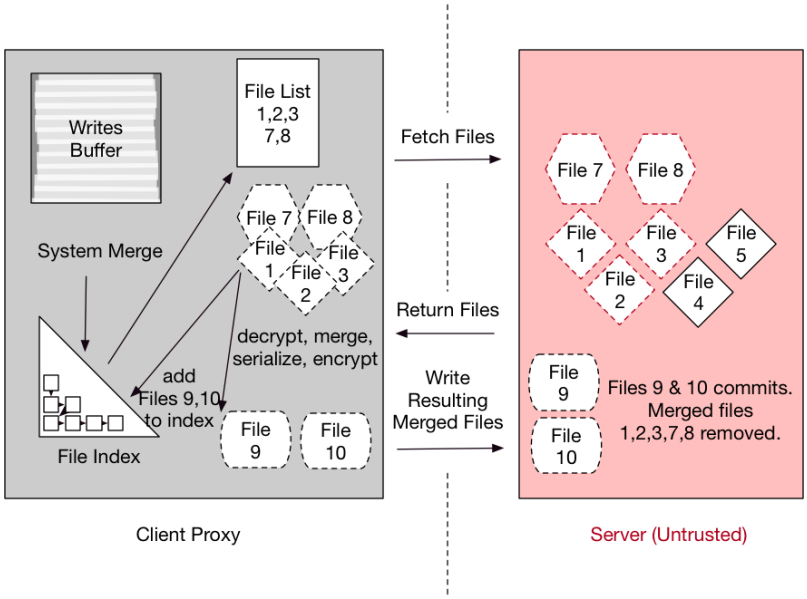

Figure 4: Periodic merges help to bound number of files and space for both server and client. Merges are opportune time for client to switch old file contents (Files 1-3,7,8) to new files under new encryption schemes (Files 9, 10).

References

Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011.

Giovanni Di Crescenzo and Abhrajit Ghosh. Privacy-preserving range queries from keyword queries. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 35–50. Springer, 2015.

Benjamin Fuller, Darby Mitchell, Robert Cunningham, Uri Blumenthal, Patrick Cable, Ariel Hamlin, Lauren Milechin, Mark Rabe, Nabil Schear, Richard Shay, et al. Security and privacy assurance research (spar) pilot final report. Technical report, MIT Lincoln Laboratory Lexington United States, 2015.

Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected database search. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 172–191. IEEE, 2017.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2): 51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL `http://doi.acm.org/10.1145/564585.564601`.

Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 162–168, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103007. URL `http://doi.acm.org/10.1145/3102980.3103007`.

Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 720–731. VLDB Endowment, 2004.

Seny Kamara and Tarik Moataz. Sql on structurally-encrypted databases. *IACR Cryptology ePrint Archive*, 2016:453, 2016.

Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996. URL `http://dl.acm.org/citation.cfm?id=230823.230826`.

Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 463–477. IEEE, 2013.

Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan. Guidelines for using the cryptdb system securely. *IACR Cryptology ePrint Archive*, 2015:979, 2015.

Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, volume 6, pages 289–300. VLDB Endowment, 2013.