ZeroDB whitepaper[a]

M. Egorov[†] and M. Wilkison[‡]
ZeroDB, Inc.
(Dated: January 25, 2016)

ZeroDB is an end-to-end encrypted database that enables clients to operate on (search, sort, query, and share) encrypted data without exposing encryption keys or cleartext data to the database server. The familiar client-server architecture remains, but query logic and encryption keys are pushed client-side. Since the server has no insight into the nature of the data, the risk of a server-side data breach is eliminated. Even if adversaries successfully infiltrate the server, they will not have access to the cleartext data.

ZeroDB provides end-to-end encryption while maintaining much of the functionality expected of a modern database, such as full-text search, sort, and range queries. Additionally, ZeroDB uses proxy re-encryption and/or delta key technology to enable secure, granular sharing of encrypted data without exposing keys to the server and without sharing the same encryption key between all users of the database.

I. INTRODUCTION

Given the cost, performance, and scalability benefits, outsourcing of on-premise infrastructure to cloud environments is accelerating. However, concerns over data security and the frequency of high-profile data breaches hinder cloud adoption in highly-regulated industries such as financial services. While strong encryption can alleviate many of these concerns (by guaranteeing data confidentiality), once data is encrypted, it is no longer usable. Several strategies for computing on encrypted data have recently emerged that seek to provide the benefits of encryption while preserving the functionality of encrypted data.

These techniques typically fall into two broad categories: (1) homomorphic encryption schemes that enable server-side computation directly over ciphertext and (2) trusted modules that are assumed to be inaccessible by adversaries [1]. These two approaches can be further deconstructed into (1) fully homomorphic encryption (FHE) and

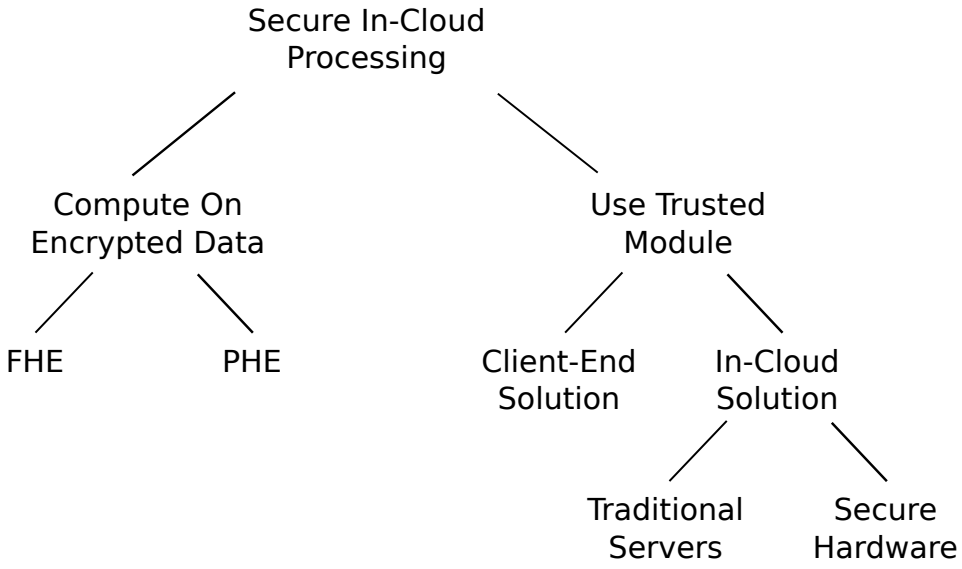
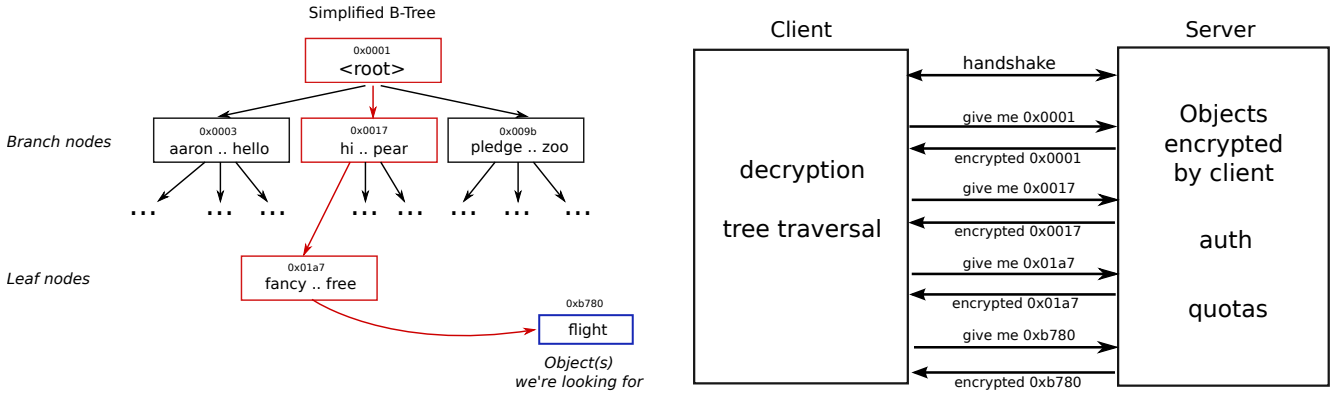


FIG. 1: Taxonomy of approaches to secure in-cloud processing. Computing directly on encrypted data can be done in a fully homomorphic (FHE) or partially homomorphic manner (PHE). Trusted Modules (TM) include client-end and in-cloud solutions.

[a] This whitepaper describes planned functionality but not necessarily the current state of ZeroDB

[†] michael@zerodb.io
[‡] maclane@zerodb.io



(a) Encrypted index traversal example (simple keyword search)

(b) Sequence of client requests for traversal of the example index

FIG. 2: Search protocol for equality query using an example of keyword search

partially homomorphic encryption (PHE) and (2) client-end solutions and in-cloud solutions, resulting in the taxonomy show in Fig. 1.

The most high-profile of these include Gentry's seminal paper on homomorphic encrytion [2], Popa's CryptDB [3], which executes SQL queries over encrypted data using a collection of efficient SQL-aware encryption schemes [4], and Cipherbase, a comprehensive database system that provides strong end-to-end data confidentiality through encryption and secure hardware. Smart provides a more comprehensive overview of the latest research and private sector efforts to achieve data encrypted in-use [5].

II. QUERY PROTOCOL

The basis of the end-to-end encrypted query protocol is as follows. The client interacts with the server during the execution of a query over a series of round trips. An encrypted index is stored on the server as a B-Tree, and the client traverses this index remotely to retrieve the necessary encrypted records. The index consists of buckets which are encrypted before being uploaded to the server and only decrypted client-side.

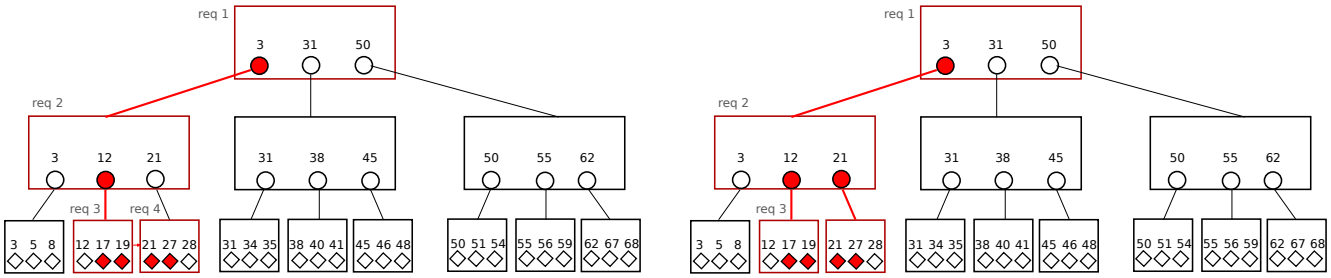
A. Equality query (using an example of keyword search)

In ZeroDB, indexes are structured as B-Trees Fig. 2a. A B-Tree consists of buckets, each of which can be either a root, branch, or leaf node. The leaf nodes of a tree point to the actual objects being stored. Thus, searching the database is a simple tree traversal.

In order to make the database end-to-end encrypted yet still capable of performing queries, the client encrypts the buckets (at the time of creation or modification). The server, which stores the buckets, never knows the encryption key used. The objects referenced by the leaf nodes of the B-Tree indexes are also encrypted client-side. As a result, the server does not know how individual objects are organized within the B-Tree or whether they even belong to an index at all. Since ZeroDB is encryption agnostic, probabilistic encryption can be used so that the server cannot even compare objects for equality.

When a client performs a query, it asks the server to return buckets of the tree as it traverses the index remotely Fig. 2b. Buckets can be cached client-side so that subsequent queries do not make unnecessary network calls.

The server is responsible for data replication, multi-version concurrency, object locking, user authentication, quotas, etc. The client performs encryption/decryption and query logic.



(a) Query searching for objects with a property $16 \leq weight$, $limit = 4$ (takes 4 requests w/o cache)

(b) Query which fetches *all* objects with $16 \leq weight \leq 27$ (takes 3 requests w/o cache)

FIG. 3: Range queries

B. Range queries

When data are organized in B-Trees, doing range queries is easy. Take an example having records *Record* with an integer property *weight*. Data pointers to *Record* objects are placed in a B-Tree sorted by *weight*.

Two different types of range queries could be performed. One is when we want a small subset of data in the beginning of the range (limit query). In this case, we find a pointer to the beginning of the range Fig. 3a, then incrementally fetch subsequent buckets if the range occupies more than one, then bulk-fetch objects themselves by their pointers.

The other case is when we want to get *all* objects in the range (select *). We download subsection of B-Tree matching the range query level-by-level in a logarithmic number of steps as in Fig. 3b. To do that, we start with the root bucket. Then, we download all the child buckets which match the range at once, repeating until we get to the leaf nodes. After that we (optionally) bulk-fetch all the objects which match the range query at once. This is done in a way similar to prefetching all trees (Section II D 1). Selecting all objects in a range reveals the approximate number of objects in this range to the server (the range and field names remain secret).

C. Complex queries (multi-keyword search, multiple conditions)

So far we have described simple queries. But queries may contain multiple conditions at the same time. Depending on the number of objects matching each condition and desired security properties we can use different approaches. Making a query with an "or" condition requires simply zip-joining two sorted datasets, so we do not consider that to be problematic. Instead, let us consider a query where we select objects matching the condition $(v_1 = a)\,\&\,(v_2 = b)$ and ordered by v_3.

1. *Prefetch approach*

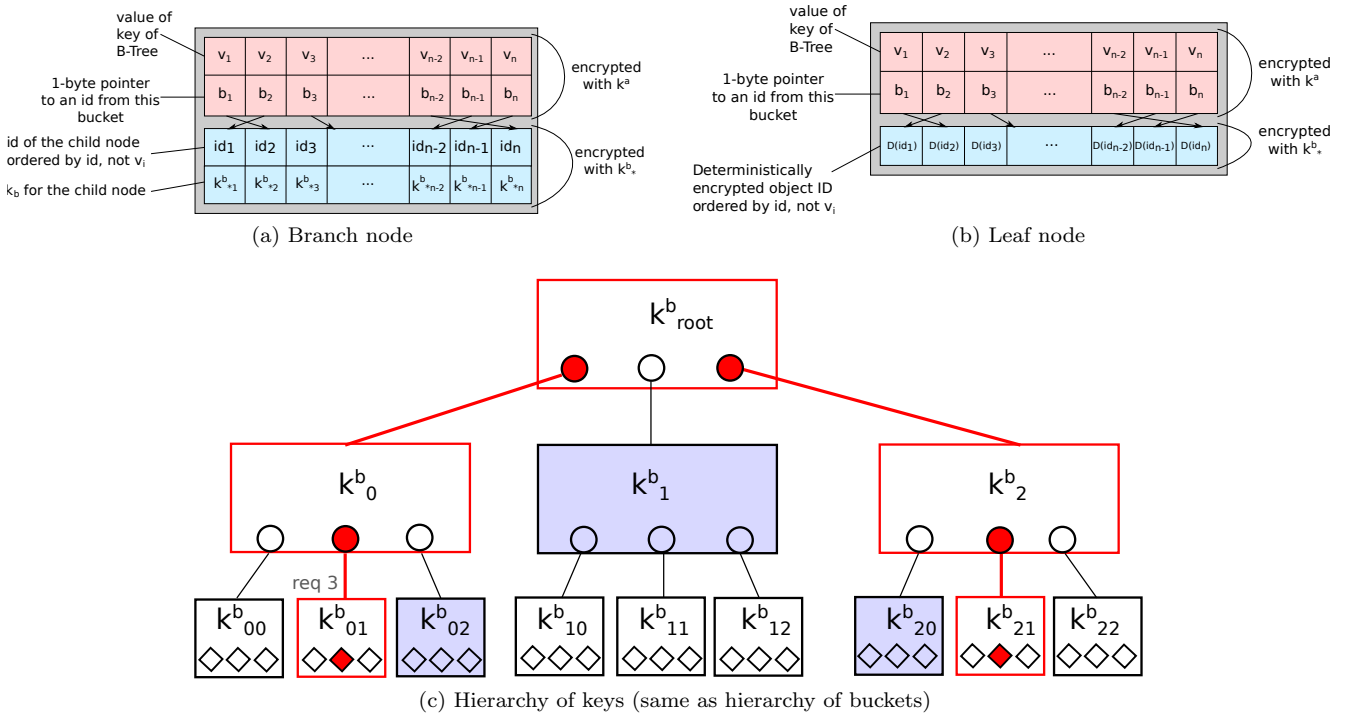
In case the number of items with $v_1 = a$ is small, it could be worthwhile to download the entire subset of object IDs matching this condition. This could be practical in a multi-keyword search when one word is rare.

We can use the following indexes for this kind of query:

$$\begin{aligned}&\mathrm{BTree}(v_1 \to \mathrm{TreeSet}(ids)),\\&\mathrm{BTree}(v_2 \to \mathrm{TreeSet}(ids)),\\&\mathrm{BTree}(id \to v_3).\end{aligned}$$

First, we estimate which condition has the smallest number of matching elements. We can do so by fetching "contours" of the B-Trees (corresponding to smallest and largest elements of the range respectively) in order to determine the height of the tree and approximate distance between the smallest and largest elements, knowing the average size of a bucket. This only requires H requests between client and server.

We prefetch a TreeSet for the most lightweight condition (Sec. II D 1). Then we bulk-search (Sec. II D 2) these IDs in the larger TreeSet (and if we do not find an ID in the leaf node, we drop it).



(a) Branch node

(b) Leaf node

(c) Hierarchy of keys (same as hierarchy of buckets)

FIG. 4: B-Tree structure which supports server-side set operations. In order for server to do these operations, the client allows it to see deterministically encrypted object IDs $D(id_i)$. The server can count the number of occurrences of each $D(id_i)$, and those values which are repeated the same number of times as the number of conditions in the "and" query are the ones the client wants to be returned.

After that, we want to sort the small subset of fetched IDs by v_3. We do a parallel traversal of the third B-Tree and find which value of v_3 corresponds to each ID.

This approach reveals the number of elements matching condition $v_1 = a$ to the server, although the condition itself remains unknown.

2. *Preorder approach*

The prefetch approach could be slow and reveal too much information to the server. So, whenever possible, values are pre-ordered in the index. It works in the following way.

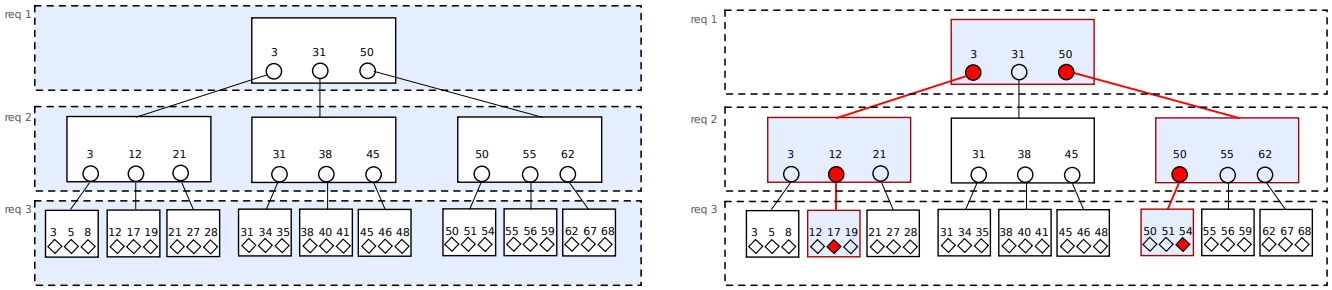
The composite index to make a query selecting objects matching the condition $(v_1 = a)\,\&\,(v_2 = b)$ and ordered by v_3 is:

$$\mathrm{BTree}((v_1, v_2) \to \mathrm{BTree}(v_3 \to \mathrm{TreeSet}(ids))).$$

For this query, we find a B-Tree (or multiple B-Trees) corresponding to the conditions and lazily traverse them in order of v_3. This makes limit orders much more performant and does not leak information about possible dataset size to the server.

3. *Doing set intersection server-side*

The performance bottleneck of many complex queries (such as multi-keyword search or queries with a product of multiple conditions) is set intersections. For multi-keyword search for example, doing set intersections server-side would help significantly when each keyword has too many pages associated with it (which is possible for extremely large datasets).



(a) When a tree (or sub-tree) is small, it can be fully pre-fetched to the client

(b) When multiple branches of the tree needs reading/updating, tree traversal can be done in parallel

FIG. 5: ZeroDB-specific optimizations for working with B-Trees includ pre-fetching a tree or finding multiple values in a number of steps equal to the tree height

Let us consider a query where $v_1 = a$ and $v_2 = b$. In order to do server-side set intersection, we reveal deterministically encrypted IDs $D(id_i)$ of all objects matching each of these conditions to the server (Fig. 4). The server can see the frequency of each ID, which is equal to the number of conditions satisfied for this ID. The server sends those deterministically encrypted IDs which satisfy all conditions (two conditions in our example) to the client.

In order to do that, buckets are split into two sections (Fig. 4 (a), (b)). The first (top) section is encrypted with key k^a, which is known to the client but never known to the server. The second (bottom) section is encrypted with a key k^b_{*i}. Keys from k^b family are hierarchical: child-keys k^b_{*i} are derived from a keyk^b_* which can decrypt the bottom section of a node as well as the bottom sections of all its child nodes (Fig. 4c). The keys can be derived as $k^b_{*i} = \mathrm{SHA256}(k^b_* + i)$.

The client traverses the B-Tree along the "contour" of the range (red at Fig. 4c). It provides keys k^b_i of buckets which are fully within the range and in-range deterministically encrypted object IDs $D(id_i)$ to the server. The server, using this information, is able to determine all children of those buckets and the $D(id_i)$ associated with them. After adding the $D(id_i)$ given by the client, the server can count how many times each value is repeated. Values which are repeated the same number of times as the number of conditions in the "and" query are returned to the client.

In this case, the server can determine the total number of records matching any of the given conditions n_1 (the conditions themselves are unknown to the server), the number of records matching exactly two of the conditions n_2, etc. until the maximum number of conditions. The server also learns "parent-child" relationships between some of the buckets. The server does not learn the order of the objects and it does not learn which conditions the client is applying.

4. Related objects

ZeroDB is based on ZODB, which uses object references instead of joins [6], similar to MongoDB's database references [7]. In order to keep everything consistent, ZODB fires events at commit time [8] to keep back-references and indexes up to date. In ZeroDB (as in ZODB), we can create indexes on attributes of related objects, much as we create normal indexes.

D. Optimizations specific for remote client

In most cases, all the query logic in ZeroDB happens client-side (e.g. client and storage are separated by a network channel with high latency). Therefore, we use two primitives specific for this architecture.

1. Bulk-fetching small (sub)trees

When a tree (or sub-tree) is small enough, it could be more performant to fetch the entire tree to the client. We do not need to do that bucket-by-bucket, as one would do with an index stored on a local hard drive, but in a logarithmic number of steps.

In order to do that, we fetch the root first Fig. 5a. The client unpacks the root bucket and fetches all its children in one query. Then it unpacks all the child nodes and learns the IDs of their children. This process continues until the entire B-Tree is fetched. Thus, the number of queries is equal to the height of the B-Tree, H, proportional to the logarithm of its size, $\log S_{\text{ix}}$.

When we fetch a tree, we implicitly show the number of objects it contains, S_{ix}, to the server. Based on the size of the read data, the observer would be able to infer a number close to S_{ix} and associate it with the bucket IDs just read. Therefore, this technique should be used as rarely as possible, combined with reading other data at the same time or with oblivious RAMs [9–12]. The latter would prevent the observer from learning bucket IDs.

2. *Parallel tree traversal*

Often queries involve several values of the same field (or different fields). For example, indexing a new document containing 100 words could be an expensive operation since it would involve $100\,H$ requests to the server, making the performance defined by client-server latency very poor.

In order to fix that, we traverse the B-Tree in parallel Fig. 5b. We first fetch the root of the tree. Then, we fetch only those child-buckets of the root which are relevant to the values in our query. Then, we fetch only the relevant children of those, etc. This way, we do tree traversal for all the necessary values in H steps.

When we do parallel traversal, the server would be able to see access patterns but it would not be able to distinguish access patterns belonging to each of the values individually.

III. SHARING DATA

If there are multiple users of the same encrypted dataset, there are serious shortcomings with sharing the encryption key with all parties. Since the architecture of ZeroDB allows using any encryption algorithm, we can use proxy re-encryption [13, 14] or delta-keys [15, 16] to share data with other users (third parties) of the same dataset. These algorithms allow the third party to hold an encryption key other than that used to encrypt the data, while still being able to do the same queries as the data owner, until the server revokes their key. We can also granularly share subsets of data, by combining these sharing algorithms with ZeroDB's query functionality.

Sharing an entire dataset is done as follows. Suppose user A owns the dataset and wants to share it with user B. In the case of proxy re-encryption algorithms [13, 14], user A has a key pair $priv_a/pub_a$ and user B has a key pair $priv_b/pub_b$. All objects and buckets are encrypted with random content-encryption keys cek_i (individual for each object) using a block cipher algorithm (AES256/GCM by default). The cek is encrypted with key encrypting key $priv_a$. So, we store $e_a(cek_i) = \text{encrypt}(priv_a, cek_i)$ and $c_i(obj) = \text{encrypt}(cek_i, data)$.

When user A wants to give user B access to the entire dataset and indexes, he constructs a re-encryption key r_{ab} such that the server can transform content-encryption key as $e_b(cek_i) = \text{transform}(r_{ab}, e_a(cek_i))$. Re-encryption key r_{ab} is given to the server, along with a revocation time. While the server has this re-encryption key, it transforms CEK's of encrypted objects $e_a(cek_i) \to e_b(cek_i)$ as client B requests the objects. The server can revoke the key by removing r_{ab} and by throttling queries. Client B is now able to do tree traversal, in the same manner as client A.

A. Granular access permissions

Let us first exemplify granular sharing of data in a scheme where A shows encryption keys to B and A shares M objects selected by a range query. It is possible to share data granularly in such a way that A has to only send B a number of keys proportional to $\log M$ rather than M.

In order to do that, keys k^a must form a tree hierarchy similar to the one we had for set intersections (Fig. 4c). Client A traverses the tree along the beginning and end of the range. It shares keys for those buckets with B giving B the ability to do tree traversal within this range (buckets with blue background in Fig. 4c). However, there may be buckets which are partly within the range. Client A passes key-reference pairs for buckets which belong to the contour of the range (there should be no more than $2\,s_b$ such pairs, where s_b is the maximum bucket size). The set of keys giving access to the in-range fraction of the index and the in-range contents of the buckets from contour of the range are given to B, encrypted with his public key.

The described sharing mechanism suffers from the fact that B is given all the information to decrypt objects within the range at once. In order to make this access revokable, we need to use proxy re-encryption. However, client A should give hierarchical rights to re-encrypt the data to the server rather than to client B. E.g. the server can only

re-encrypt objects within the range given a set of re-encryption keys r_{ab}. It is possible to do this using a conditional proxy re-encryption scheme (C-PRE) [17–20].

Usually such proxy re-encryption schemes support giving a product of multiple equality conditions. The conditions which we apply would be object IDs in the bucket hierarchy, starting from the top. For example, C-PRE encryption of a bucket in the second layer of the tree, b_1, would give the server ability to derive re-encryption keys for all of its children, b_{1*}. This gives the server ability to re-encrypt all buckets within the range specified by A, yet only requires A to calculate a number of keys proportional to the log of the number of records shared.

IV. SECURITY ANALYSIS

ZeroDB is agnostic to the encryption algorithm used. By default, we use authenticated strong encryption (AES-256 in GCM mode). Nothing is decrypted on the server unless we choose to use server-side set intersections (Sec. II C 3).

Using deterministic or order-preserving encryption visible to the server could be detrimental to the security of the database in certain cases [21]. However, even traversal of B-Trees leaves the possibility for an attacker to collect access statistics and infer the data in the database [22]. We provide a security analysis and an example of the number of queries necessary for such an attack.

A. Threat model

We adopt a threat model similar to the one used for the security analysis of CipherBase [3]. We introduce a *strong adversary*, who has unlimited observational power and can view the contents of memory and disk at every moment of time. A *weak adversary* can observe only a snapshot of memory and disk at one particular moment of time. We assume that both strong and weak adversaries are passive (honest but curious) and do not return wrong results to the client.

B. Data confidentiality

In ZeroDB, all contents of the database and indexes are stored as encrypted objects. Therefore, a weak adversary does not learn anything except for database size (including the number of fields, etc), even during queries. So, ZeroDB provides *semantic security* against a weak adversary.

A strong adversary, on the other hand, is able to observe access patterns and collect statistics of those over time. So, we provide *operational security* against a strong adversary.

First, we consider search queries which just require tree traversal. For example, limit orders for queries which do not involve pulling a large subset of the index to the client. For simplicity, let's assume that all we have in a database index is a B-Tree of customer surnames. An active adversary is able to acquire access statistics of the index leaf nodes when searches of, say, 10 people with a given last name are performed. However, the amount of statistics should be large enough to be able to distinguish different records in the database. If we assume that probability of a given surname appearing in a given query is the same as in the U.S. Census 2000 data [23] and that keywords are distributed across queries according to a Poisson distribution, an adversary needs to observe two keywords at least N_1 and N_2 times, such that $|N_1 - N_2| > \sqrt{N_1 + N_2}$. The fraction of last names which can be revealed by a strong adversary from a query given some number of queries is shown on Fig. 6. For example, the surnames in 10% of new queries can be inferred by a strong adversary after observing a million queries (although surnames not touched by queries are not revealed). Unlike CipherBase [3], our approach does not leak information about the order of the elements. By observing access patterns, the server can only figure out which buckets are leaf nodes and which are branch nodes.

In cases where we need to prefetch a large chunk of a B-Tree (Sec. II C 1), we immediately reveal information about the size of the sub-dataset to the server. Of course, this is not desirable. To avoid this, we can download adjacent records in the same query so that the downloaded size is always approximately constant and equal to that of the largest sub-dataset. This way our security guarantees are similar to those for tree traversal. In order for this to work we need to unfold nested trees into one tree (Sec. V B).

The security guarantees of server-side set intersections (Sec. II C 3) are weaker. One query leaks the sum of the sizes of the datasets involved in the query and how many elements satisfy any c conditions from the query where $1 \leq c \leq n$, where n is the number of conditions in the query. The server can also assume that objects themselves (which are not stored in the index) will be read after the client obtains the result of the query and those can then be linked to deterministically encrypted object IDs. "Fake" deterministically encrypted object IDs can help to not open the number of elements in sub-datasets, but this method still shows the equality graph to the server.

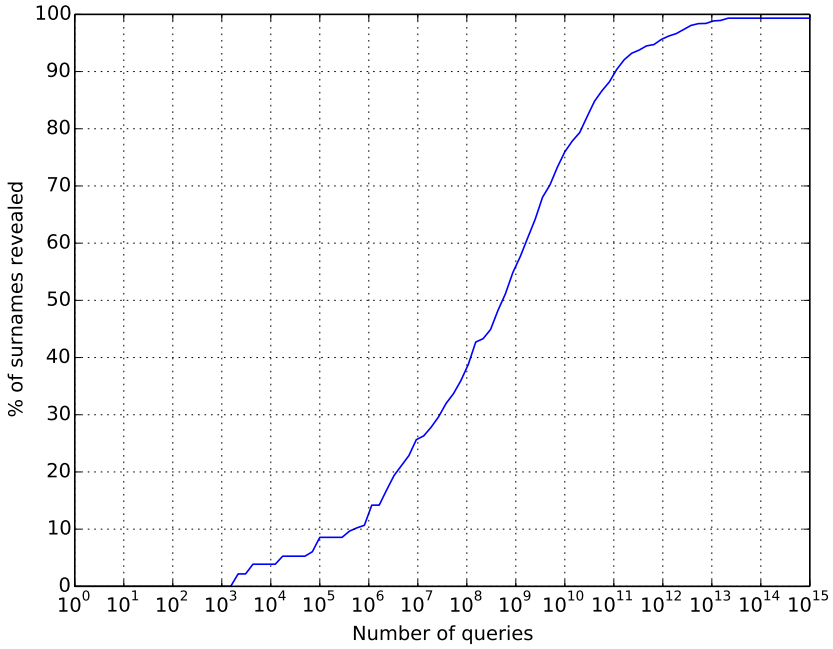


FIG. 6: Fraction of queries where last name can be inferred by a passive observer depending on the number of queries made to the database

We can significantly improve security guarantees by using ORAM techniques. They allow us to hide access patterns by writing back data after reading it. Several algorithms are available, though the most promising ones are Burst ORAM [10] and oblivious data structures [12]. Still, these require writing back to the database about 20 times the amount of data read (although these numbers improve every year). It is also possible that hiding access patterns in a multi-cloud or fully decentralized arrangement has quite good performance properties [11], which is especially interesting with decentralized storage layers such as Storj.io [24] and IPFS [25].

V. PERFORMANCE

A. Optimizing bucket size

ZeroDB requires several interactions between client and server in order to return query results. Thus, both connection bandwidth b and time of round-trip between client and server τ, determine the performance of queries. Moreover, there is an optimal bucket size s_b for any given parameters.

Suppose we have an index in the form of a B-Tree of size s_i bytes. The size of one record in the bucket is s_r bytes. Apart from encryption one can apply compression. The compression rate is defined as the ratio of initial bucket size to final (compressed + encrypted) bucket size is c. The number of requests necessary to perform a query with an empty cache is equal to the height of B-Tree h, though we will also consider the case when we are allowed to cache up to s_c bytes.

A good approximation for B-Tree height is [26]:

$$h = \log_m n,$$

where $m = s_b/s_r$ is the number of records in one bucket, $n = s_i/s_r$ is the total number of records in the index, assuming the best and the worst case heights are close to each other. Thus:

$$h \approx \frac{\ln s_i - \ln s_r}{\ln s_b - \ln s_r}.$$

When we are allowed to use caching, we cache predominantly the top of the B-Tree with an average cached height approximately equal to the height of a B-Tree with same number of elements as the number of cached elements:

$$h_c \approx \frac{\ln s_c - \ln s_r}{\ln s_b - \ln s_r}.$$

The average number of interactions between client and server per query will be $k = h - h_c$. Thus, the time which this interaction takes is:

$$\Delta t = k\tau + k\frac{s_b}{cb} = (h - h_c)\left(\tau + \frac{s_b}{cb}\right) = \frac{\ln s_i - \ln s_c}{\ln s_b - \ln s_r}\left(\tau + \frac{s_b}{cb}\right).$$

We seek to make queries as fast as possible (i.e. to minimize time Δt necessary for a query to complete) seeking for the optimal bucket size, s_b. The largest possible bucket size is equal to the size of the index and is obviously not efficient for sufficiently large indexes (it takes too long to download the index). On the other hand, when bucket size is too small, the number of requests is too large and query performance suffers due to latency between client and server. An optimal compromise can be determined by equalizing the first derivative to zero:

$$\frac{\partial \Delta t}{\partial s_b} = 0$$

which yields the optimal bucket size:

$$s_b = \frac{cb\tau}{W\left(\frac{cb\tau}{es_r}\right)},$$

where W is the real part of Lambert W function [27] and $e = 2.718281828\ldots$.

In ZeroDB, the typical size of a record in a branch bucket is $s_r = 29$ bytes, compression rate with zlib $c = 3$. For a typical network bandwidth of $b = 10^6$ (1 MB/s) and roundtrip time $\tau = 0.05$ s, the optimal bucket size is $s_b = 10$ kB when compression is not used, or 8 kB when compression is used. For a typical cache size of 5 MB and index size of 50 GB, one query using this index on average takes $t = .07$ s and client-server exchange of 11 kB of compressed data (without caching that would take 0.17 s and exchanging 24 kB of compressed data), reducing it to an average of 1.3 roundtrips. We can finetune the optimal bucket size on the fly as the connection speed improves over time.

When we need to traverse several indexes at once, we bunch requests for several indexes in one query. For example, the first request for indexes A and B would be "read root of indexes A and B (by object IDs of the roots)". In the past, that would make some heavier queries impractical, but the growth in internet bandwidth according to Nielsen's law [28] alleviates this.

B. Unfolding trees with subtrees into one tree

By default, we often use nested B-Trees of TreeSets. While that yields performance benefits in traditional databases, it is not ideal when working with large latencies, τ. And, more importantly, it has security drawbacks. For example, one cannot easily read adjacent records to hide the size of the dataset from the server.

Consider a B-Tree linking word IDs, *wid*, with TreeSets of object IDs, *oid*. We can instead use a treeset of tuples (wid, oid) which will be unique. The tree will have approximately the same height for all *wid*s. Also, it is easy to pre-set the approximate size of the dataset to download if we don't want to show how many *oid*s we have per *wid*.

VI. CRYPTOGRAPHIC PRIMITIVES USED

Even though the server never decrypts the data, we use encryption in transit (using SSL) in order to prevent third parties from analyzing access patterns. The client authenticates with an X.509 certificate or a self-signed certificate where the private key is derived from a passphrase. When a passphrase is used, the scrypt algorithm is used to derive a private key from the passphrase. Information about the salt is stored server-side and is given before establishing an authenticated encrypted connection.

For symmetric encryption, we use AES256 in GCM mode. The symmetric key is derived from either the passphrase, or the private key in an X.509 certificate.

If proxy re-encryption is used, we use the AFGH algorithm [13] to encrypt a random content key for an AES256 block cipher in GCM mode.

---

[1] A. Arasu, K. Eguro, R. Kaushik, and R. Ramamurthy, in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14 (ACM, New York, NY, USA, 2014) pp. 1259–1261.

[2] C. Gentry, Commun. ACM **53**, 97 (2010).

[3] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy, *Transaction Processing on Confidential Data using Cipherbase*, Tech. Rep. MSR-TR-2014-106 (2014).

[4] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11 (ACM, New York, NY, USA, 2011) pp. 85–100.

[5] N. Smart *et al.*, *Future Directions in Computing on Encrypted Data (Draft)*, Tech. Rep. (2015).

[6] M. Faassen, "A misconception about the zodb," (2008).

[7] "Mongodb documentation: Database references," .

[8] "zope.lifecycleevent," .

[9] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13 (ACM, New York, NY, USA, 2013) pp. 299–310.

[10] J. Dautrich, E. Stefanov, and E. Shi, in *23rd USENIX Security Symposium (USENIX Security 14)* (USENIX Association, San Diego, CA, 2014) pp. 749–764.

[11] E. Stefanov and E. Shi, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13 (ACM, New York, NY, USA, 2013) pp. 247–258.

[12] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14 (ACM, New York, NY, USA, 2014) pp. 215–226.

[13] G. Ateniese, K. Benson, and S. Hohenberger, in *Topics in Cryptology–CT-RSA 2009* (Springer, 2009) pp. 279–294.

[14] B. Libert and D. Vergnaud, Information Theory, IEEE Transactions on **57**, 1786 (2011).

[15] R. A. Popa and N. Zeldovich, "Multi-key searchable encryption," Cryptology ePrint Archive, Report 2013/508 (2013).

[16] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014) pp. 157–172.

[17] J. Weng, R. H. Deng, X. Ding, C.-K. Chu, and J. Lai, in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09 (ACM, New York, NY, USA, 2009) pp. 322–332.

[18] S. Chow, J. Weng, Y. Yang, and R. Deng, in *Progress in Cryptology — AFRICACRYPT 2010*, Lecture Notes in Computer Science, Vol. 6055, edited by D. Bernstein and T. Lange (Springer Berlin Heidelberg, 2010) pp. 316–332.

[19] J. Qiu, G.-H. Hwang, and H. Lee, in *Information Security (ASIA JCIS), 2014 Ninth Asia Joint Conference on* (2014) pp. 104–110.

[20] L. Fang, W. Susilo, C. Ge, and J. Wang, Comput. Stand. Interfaces **34**, 380 (2012).

[21] M. Naveed, S. Kamara, and C. V. Wright, in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (ACM, New York, NY, USA, 2015) pp. 644–655.

[22] M. S. Islam, M. Kuzu, and M. Kantarcioglu, in *Network and Distributed System Security Symposium (NDSS)* (2012).

[23] "Frequently occurring surnames from the census 2000," United States Census Bureau (2014).

[24] "Storj. a peer-to-peer cloud storage network," (2014).

[25] "Ipfs is a new peer-to-peer hypermedia protocol," .

[26] Wikipedia, "B-tree," (2016), [Online; accessed 23-Jan-2016].

[27] Wikipedia, "Lambert W function," (2016), [Online; accessed 23-Jan-2016].

[28] "Nielsen's law of internet bandwidth," (2014).