MiniCrypt: Reconciling Encryption and Compression for Big Data Stores

Wenting Zheng†, Frank Li†, Raluca Ada Popa†, Ion Stoica†, Rachit Agarwal◊
†University of California, Berkeley, ◊Cornell University

Abstract

We propose MiniCrypt, the first key-value store that reconciles encryption and compression without compromising performance. At the core of MiniCrypt is an observation on data compressibility trends in key-value stores, which enables grouping key-value pairs into small *key packs*, together with a set of distributed systems techniques for retrieving, updating, merging and splitting encrypted packs. Our evaluation shows that MiniCrypt compresses data by as much as 4 times with respect to the vanilla key-value store, and can increase the server's throughput by up to *two orders of magnitude* by fitting more data in main memory.

1. Introduction

Many applications today capture an immense amount of private data about their users [30], such as their purchase history, social interactions, and communication. This private data is stored at servers running highly performant data stores [2–4, 6, 12, 14, 17, 23, 25]. Unfortunately, leakage of confidential data from these servers is a significant problem [13].

Applications need data storage systems that are able to preserve data confidentiality and handle big data workloads efficiently. Many big data stores employ compression [12, 24] to significantly increase performance, sometimes by up to an order of magnitude [6, 23]. Compression is effective at providing performance gains because it enables servers to fit more data in main memory, thus decreasing the number of accesses to persistent storage. To protect data confidentiality, a natural solution is to encrypt the data stored on servers [9, 19, 22, 32] and keep the key at the client. Therefore, an ideal system that aims to protect confidentiality and preserve performance should incorporate both encryption and compression into its design.

EuroSys '17 April 23-26, 2017, Belgrade, Serbia
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4938-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064176.3064184

Unfortunately, existing systems choose either compression or encryption – but not both – because there is a *fundamental tension* between encryption and compression. First, compressing encrypted data (which is randomized) is not viable because pseudorandom data is not compressible. Second, while encrypting compressed data works well in some systems, it is problematic in the database setting. Compressing a single row of data typically provides limited compression ratio, while compressing multiple rows together means the server cannot maintain fine-grained access to these rows/attributes and makes it more difficult to maintain correct semantics. There are a range of effective database compression techniques [5, 8, 21] that also permit querying data, but their layouts leak significant information about the data, as we discuss in Section 2.4.

In this paper, we propose MiniCrypt, the first key-value store that achieves the benefits of both compression and encryption. Our system model considers a cloud-hosting context, in which a hosting service (the server) hosts a key-value store, and a company/organization (the customer/client) uses the hosting service. The client has a symmetric key using which it encrypts the values stored on the server. To ensure that MiniCrypt's design is generic and not tied to a specific key-value store, we designed MiniCrypt as a layer on top of *unmodified* key-value stores. This makes MiniCrypt easier to adopt into different key-value stores and enables MiniCrypt to benefit from their performance and fault-tolerance. Our solution leverages only two basic primitives likely to exist in many key-value stores: a sorted index on the primary key as well as a single-row conditional atomic update mechanism.

MiniCrypt starts with an empirical observation on data compressibility trends in key-value stores. It is well known that a better compression ratio can be achieved by compressing more data. However, encrypting large compressed chunks is problematic for key-value stores. Since the server cannot decrypt, data processing has to be done on the client side, thus requiring the client to retrieve a large amount of data even if it only needs to operate on a single row. Our empirical observation is that by compressing together *only relatively few* key-value rows, one can achieve a high compression ratio as compared to compressing the entire dataset. As we discuss in Section 3, we observed this behavior for a wide range of datasets that could be stored in key-value stores, such as data from Github, genomics, Twitter, gas sensors, Wikipedia, and anonymized user data from Conviva (internet-scale video access logs) [1]. For example, our compression experiments on the Conviva dataset showed that compressing 1 row yields a compression ratio of 1.6, compressing 50 rows yields a compression ratio of 4.6, and compressing 8.7 million rows (the entire dataset) yields a compression ratio 5.1. We observed that compressing only a small fraction (0.00057%) of the total number of rows already provides 90% of the maximum compression ratio.

Based on this compression ratio observation, MiniCrypt *packs* together *few* key-value pairs, compresses, and encrypts them using the shared encryption key. Clients now retrieve and update packs instead of individual key-value pairs. However, this simple design runs into a significant challenge: *encrypting packed rows removes the server's ability to manage key-value pairs and to maintain correct semantics* because the server cannot decrypt. In a system without encryption, the server is able to decompress the data to read or update a key-value pair. Maintaining these properties turns out to be a challenging distributed systems problem, which we address through a set of new techniques.

The first challenge results from the fact that the server can no longer serve or update individual keys since it cannot decrypt encrypted packs. For example, how can the server fetch the correct pack based on a given key? What happens when concurrent clients update *different* keys that happen to reside in the same pack? If not carefully designed, concurrent clients may write over each other's updates. Fortunately, these problems can be addressed using systems techniques. First, we provide a simple mapping scheme that allows the server to identify the pack containing the key, while avoiding the overhead of looking up the ID of the pack in a separate server table or index. We also propose an algorithm for preventing blind overwrites based on a lightweight *single-row* synchronization primitive, which is provided by many existing key-value stores. MiniCrypt's writes are slower because each write consists of a read of the encrypted pack, followed by a synchronized write. To alleviate this problem, we provide a separate mode that targets a very common workload – appends-only workload (e.g. time-series data) – and provide a tailored protocol that achieves a write throughput close to that of the underlying key-value store in this mode. We call this new mode the `APPEND` mode, and the original mode of operation the `GENERIC` mode (because it can handle any workload).

MiniCrypt's second challenge is the server's difficulty in managing encrypted key packs. As more rows get inserted, packs might become too large and bottleneck the network. Therefore, MiniCrypt introduces a pack split protocol in the `GENERIC` mode to safely split large packs. We also provide a merge protocol in the `APPEND` mode that helps to merge append key-value pairs into packs and achieve fast write throughput. However, since the server cannot split or merge, pack management must be designated to the clients. Merge and split operations act on multiple rows of data, but many existing key-value stores do not provide sophisticated transactional mechanisms (e.g., most stores do not support transactions over multiple keys). We provide new algorithms to merge and split packs using the same *single-row* synchronization primitive mentioned above.

We implemented MiniCrypt on top of Cassandra [25] without modifying its internals. Our performance evaluation shows that MiniCrypt delivers significant compression, while keeping the data encrypted: for example, on the Conviva dataset, MiniCrypt compresses the data by a factor of 4.3; this ratio is close to the maximum one of 5.1, which can be obtained by compressing all the data together into one unfunctional blob. We show that MiniCrypt can increase the server's throughput significantly as compared to both an encrypted baseline (Cassandra with single row encryption, but without MiniCrypt's compression) and a vanilla version (Cassandra with no encryption) by fitting more data in main memory. For example, on the Conviva dataset, the read throughput increases by up to 100 times (for disk-backed servers) and 9.2 times (for SSD-backed servers) compared to the encrypted baseline, and even up to 6.2 times (for SSD-backed servers) compared to the vanilla version. Moreover, MiniCrypt's packing delivers particularly good performance for range queries, which are common for time-series data: 5 to 40 times faster than the encrypted baseline.

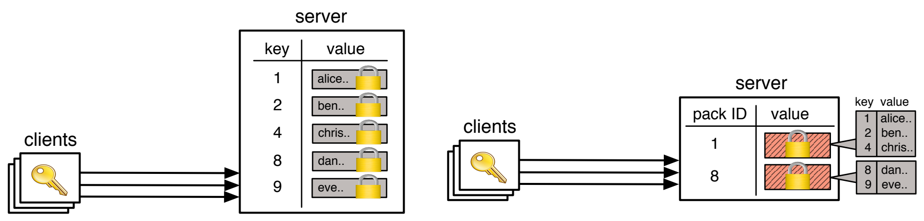
2. Overview

2.1 Model and threat model

We adopt a cloud-hosting model, consisting of two roles: the hosting service (the server, which can be distributed), and the company/organization that uses the hosting service (the customer). The customer consists of one or more client machines within the same trust domain; these share a single encryption key, which is not available to the server. The server hosts the encrypted key-value store, and the clients issue queries to the server.

MiniCrypt protects against an attacker who manages to gain access to the server, can see all server side data (including messages from the clients), and attempts to exfiltrate the data. MiniCrypt considers a passive attacker, which follows the MiniCrypt protocol (e.g., a curious system administrator). In particular, it does not corrupt the data and does not provide incorrect query results. We also assume that the clients are trusted and allowed to see all server-side data (e.g., because the customer is the data owner). In this paper, we are concerned only with protecting the data from a server attacker. As a consequence, we assume that the attacker cannot control any client and hence cannot execute any queries through the client (e.g., insert, delete, get).

Many applications implement finer grained client access control to ensure that certain clients have access to only part of the data. Such access control can be implemented



(a) Typical setup with encryption. (b) MiniCrypt's setup (encryption and compression).

Figure 1: System architecture for a typical encrypted key-value store and MiniCrypt. A lock indicates an encrypted item. Orange indicates a compressed item. MiniCrypt's values consist of compressed and encrypted packs.

in various ways in a MiniCrypt setup and is complementary to MiniCrypt. For example, the customer can add a proxy between the clients and the server and the proxy acts as a MiniCrypt client: the proxy restricts access to queries and query results to the clients. Another way is to let the clients maintain different keys for each group of data with the same access permissions, essentially running a separate MiniCrypt instance for each group. Since pack compression is done on the client side, the client can pack data with different permissions in separate packs. Although both of these designs can be easily integrated into MiniCrypt, client access control is not the focus of MiniCrypt – MiniCrypt is concerned with protecting the data from the cloud provider.

Finally, access patterns (e.g., which keys are being retrieved) or timing attacks are out of scope for MiniCrypt.

2.2 Goals

MiniCrypt has the following design goals. First, MiniCrypt aims to provide end-to-end encryption for the values in the key-value store. Second, MiniCrypt aims to provide significant compression, which promises better performance in many situations. For example, MiniCrypt should achieve higher read throughput than a standard encrypted service without compression, because MiniCrypt can fit more data in memory. Third, MiniCrypt aims to work as a layer on top of *unmodified key-value stores*. This makes it easier to adopt MiniCrypt into different key-value stores and enables MiniCrypt to benefit from their performance and fault tolerance. Finally, MiniCrypt aims to provide eventual consistency guarantees, which is often utilized in big data key-value stores due to its performance.

2.3 System API

MiniCrypt exposes the basic key-value store API, as well as support for range queries. Range queries (in the form of `get (low, high)`) are common for time-series big data [11, 27]. MiniCrypt supports the following API:

Function	**Description**
`get (key)`	returns value associated with `key`
`put (key,val)`	sets the value for `key` to `val`
`delete (key)`	deletes the record with key `key`
`get (low, high)`	returns all the (`key`, `value`) pairs where `low` $\leq$ `key` $\leq$ `high`

2.4 Strawman designs

Compression is a widely studied topic in databases [5, 6, 8, 21]. We briefly discuss two compression strawman designs and show their limitations.

The first approach is utilizing compression techniques that allow queries to be run on the compressed data [5, 6, 21]. Directly adding encryption will leak significant information about the data due to the data layout and the data access patterns. For example, run-length encoding (RLE) [5], commonly used in column-oriented databases, encodes runs of values (a contiguous sequence of the same values) together. Ten consecutive rows with the value "female" are encoded as ("female", 10). One possible way to integrate encryption is to encrypt the run value (e.g., "female") separately, while leaving the run length (e.g., 10) in plaintext. This allows the server to answer a `get (key)` query, during which it can use the run length information to return the correct value. However, this arrangement easily leaks important information such as data frequency. For example, if the column is gender (F/M) or letter grades (A, B, C) and is sorted by this value, the server-side attacker can learn which row has what grade or gender by observing the run lengths that are stored in the plaintext.

Dictionary encoding [8] is another common compression technique. The clients share a compression table that maps uncompressed values to compressed codes. To compress, the clients look up specific values in this table and construct the correct compressed version. To decompress, the clients refer to the same shared table and translate the compressed values back into the correct uncompressed text. The advantage of this design is that it combines compression and encryption without introducing the complexity of packs. However, this approach has significant disadvantages. First, dictionary encoding works well for some columns (such as columns with few distinct values), but it does not work well in general. As an example, we ran this technique on Conviva data and found that, even though the compression rate was very high for some columns, the overall compression ratio was only 1.6. Second, each client needs to use the compression table for both reads and updates. If the table is stored on the server side, each client must do extra reads in order to decompress and compress, which will reduce the read throughput as well as leak significant information through access patterns on this shared table. Storing the table at the client imposes performance overhead. The compression table can be very big: for example, for Conviva, the table was 80% of the entire compressed data. Finally, as data gets modified, the contents of the table change over time. The database must provide a protocol for synchronizing the compression table stored on different clients, adding further complexity. Not updating the dictionary table might result in an out-of-date dictionary table that wastes space storing past encodings.

MiniCrypt aims to be a generalized system that can handle both reads and updates on a variety of data types while providing strong confidentiality guarantees. Our packing technique is independent of workloads, data types, and compression algorithms.

2.5 MiniCrypt's design overview

Figure 1 summarizes MiniCrypt's architecture in comparison to a regular key-value store that provides encryption in a straightforward way. This is a logical baseline for MiniCrypt because it provides similar security. The data in MiniCrypt is stored in packs, where each pack is a group of key-value pairs, compressed together and encrypted. The keys in a pack represent a contiguous range in a sequence sorted on the keys.

Each pack has a packID. We choose the packID to be the smallest key in the pack, and assume there is a sorted index on the packID. This means that to find a given key, we simply need to retrieve the pack with the largest ID smaller than or equal to the key. Each pack also has some metadata information such as a hash of its value and status messages (to be introduced in later sections).

To read a key, a client fetches the corresponding pack, decrypts and decompresses it. To write a key, a client updates a pack. As keys get deleted or inserted, some packs become too small or too large; in this case, MiniCrypt merges or splits them to maintain performance.

Security guarantees. MiniCrypt protects each pack with a strong and standard encryption scheme (AES-256 in CBC mode), which provides semantic security. The clients never send the decryption key to the server. Thus, this encryption protects the values in the original key-value store. The encryption leaks nothing about the contents of the pack, except for the size of each compressed pack. While MiniCrypt reveals the size of a compressed pack, it does not reveal the sizes of the original rows within the pack, which are revealed by the vanilla system. MiniCrypt enables reducing the information leaked by the size of the pack by padding the encrypted packs to a tier of a few possible sizes. MiniCrypt allows the customer to specify the padding tiers, such as small-medium-large or exponential scale, and pads each pack to the smallest tier value that is at least the pack size. This strategy provides a tradeoff between compression and security. Note that, in our threat model, an attacker does not have the ability to issue write queries to the storage through the client, which prevents the attacker from doing injection attacks that exploit pack size.

The keys in a key-value store are often not sensitive (e.g., random identifiers, counters, timestamps). However, if the keys are deemed sensitive, then the packIDs should be encrypted since a packID reveals the lowest key in the pack. Note that the rest of the keys are automatically encrypted by MiniCrypt as part of a pack. MiniCrypt supports packID encryption only in `GENERIC` mode and for a system that does not perform range queries, as follows. MiniCrypt applies a pseudorandom function [20] to the packIDs, keyed using a different symmetric key for each table, and treats that as the new packID. While this is a deterministic encryption scheme, it is essentially as secure as randomized encryption because the keys in a key-value store are *unique*. This determinism allows MiniCrypt to proceed as normal by maintaining a sorted index on the *encrypted* keys and allows the clients to directly query on them. MiniCrypt does not support range queries or `APPEND` mode in this case, because there is currently no encryption scheme that enables ranges while providing semantic security and being efficient in our setting. A number of different schemes for range queries on encrypted data exist, trading off either security or efficiency. For example, order-preserving encryption (OPE) schemes [10, 31] enable efficient range queries on encrypted data in exchange for revealing the order of packIDs to the server. In the rest of the paper, we treat packIDs as unencrypted for simplicity, but our evaluation uses encrypted packIDs. Finally, MiniCrypt does not hide the number of rows/packs in the database or the structure of the database (e.g., number of tables).

2.5.1 The underlying key-value store

To achieve the goal of working on top of unmodified key-value stores, MiniCrypt should not rely on specialized or expensive primitives that are not supported by most key-value stores. For example, most key-value stores do not support transactions covering multiple keys. The few that do (e.g. Redis and Cassandra) support them with limitations and significant performance overhead.

In fact, MiniCrypt can work on top of any key-value store that supports an ordered index on keys and provides a conditional atomic update (update-if) primitive for a single row. The first property enables MiniCrypt to support range queries and to efficiently locate the packID for a given `key`, as discussed in Section 4.1 and Section 4.2. Most key-value stores support an index on the key and many of these enable ordered clustering or range queries on this key.

The second property is a lightweight transactional primitive that executes an update on a single row at the server only if the condition is true. This primitive can be of the form "UPDATE ... IF condition" or "INSERT ... IF NOT EXISTS". This type of transaction is relatively lightweight because (1) it operates on a single row and (2) it contains only one statement as opposed to a multi-statement proce-

Mode	Type of write	Pack op.	Perfor. note
Generic	all types (`get`, `delete`)	split	`put` uses update-if
Append	append, `put`, no `delete`	merge	fast appends

Table 1: Comparison between MiniCrypt's modes, including the pack maintenance operation.

dure. Most key-value stores support such transactions. For example, in Cassandra, these are called *lightweight transactions* [18].

MiniCrypt maintains the semantics of the underlying store for eventual consistency, which is commonly used in the key-value store setting. MiniCrypt also benefits from the fault tolerance and replication mechanisms of the underlying store.

2.5.2 Modes of operation

MiniCrypt provides two modes of operation: GENERIC and APPEND. Table 1 compares the two modes.

The GENERIC mode applies to any application: clients can update, insert, and delete key-value pairs. Writes use the update-if primitive to prevent clients from overwriting each other's updates. When a pack becomes too large, it is split. Since supporting both split and merge at the same time is overly complex, MiniCrypt does not support merging packs in this mode.

The APPEND mode supports applications in which writes are appends of new keys, and there are no deletes. Many big data applications are append-only. For example, a common pattern for big data is time-series data [11, 27], where the keys are timestamps, while values are typically actions or measurements. In this mode, MiniCrypt delivers high performance for appends, essentially as fast as the underlying system. Appends get inserted directly into the key-value store and not into packs. Then, background processes running on clients merge these keys in packs.

3. Key packing

In this section, we empirically justify the observation that compressing a *relatively small* number of key-value pairs together yields a compression ratio that is a significant fraction of the compression ratio obtained when compressing an entire dataset into one unfunctional blob.

The six datasets we examined are: anonymized Conviva time-series data on user behavior, genomics data where each entry consists of an identifier and a partial sequence of the genome, time-series Twitter data consisting of tweets and metadata, time-series data from gas sensors, Wikipedia files, and Github files from the Linux source code. We examined 5 different compression algorithms (bz2, zlib, lzma, lz4, snappy), which provide different tradeoffs in compression ratio and speed. For each pair of dataset and compression algorithm (30 pairs), we plotted the compression ratio against the number of rows in the pack. We first re-format each dataset into key-value pairs, then group the values into packs by adjusting a maximum threshold (in bytes) for each pack and calculate the average number of values present in each pack for a given pack size.

Figure 2 shows the results. Due to space constraints, we include here the graphs for Conviva and genomics with a table summary of the other datasets. The x-axis shows the average number of values, while the y-axis shows the compression ratio. We can see from Figure 2 that the compression ratio grows very fast as the number of rows increases, then quickly plateaus close to the maximum compression ratio achieved when compressing the entire dataset. For example, for Conviva, compressing 1 row yields a compression ratio of 1.6, compressing 50 rows yields a compression ratio of 4.6, and compressing the entire dataset of 1.5 million rows yields a compression ratio of 5.1. Hence, a relatively small number of rows per pack suffices for a significant compression ratio. We explain how MiniCrypt determines the pack size for a given dataset and system parameters in Section 8.3.

As surveyed in [16], there is a sharp tradeoff between compression ratio and the speed of compression/decompression. For example, bz2 and lzma have high compression ratios but poor compression/decompression speed, which affects client latency in MiniCrypt. Considering this tradeoff, we chose to use zlib in MiniCrypt as it achieves both a good compression ratio and good compression/decompression speeds.

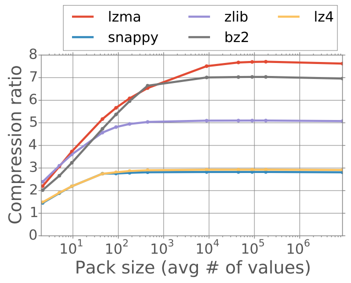
4. Read operations

In this section, we describe read operations in MiniCrypt, which work the same in both the GENERIC and APPEND modes.

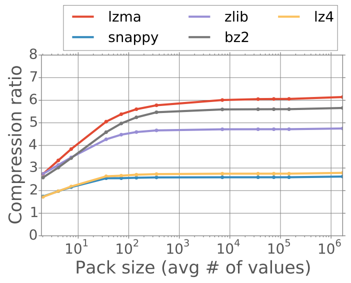
4.1 Get

Since key-value pairs are packed and encrypted in MiniCrypt, clients can only fetch at the granularity of a pack. A question is: how does a client know the packID given to the key of interest? One option is to maintain a table mapping keys to packIDs at the server. This strategy is undesirable because it increases server side space usage and query latency, and is difficult to keep consistent with the main data table during concurrent updates and client failures.

Instead, MiniCrypt enables clients to fetch the correct pack without knowing the packID. Since the packID is chosen to be smaller than or equal to the smallest key in the pack, the pack corresponding to a key k *is the pack with the highest ID from all the packIDs that are at most* k. This query can be run efficiently at the server because the underlying key-value store keeps an ordered index on packID. To find the right pack, the server simply locates k by retrieving the packID immediately preceding it. Once the client receives the result of this query, it decrypts and decompresses the pack. It then scans the content and retrieves the value for



(a) Conviva



(b) Genomics

All Datasets	Total Num of Rows	Avg Value Size	Max Comp Ratio	Pack Size (rows)
Conviva	8.7M	1.1KB	5.1	25
Genomics	1.6M	1.3KB	4.8	14.5
Twitter	2.5M	5.5KB	8.3	4.2
Gas Sensor	4.2M	135B	3.4	75
GitHub	50.8K	11.6KB	4.5	2.5
Wikipedia	202.5K	13.2KB	3.1	1.4

Figure 2: Compression ratios for different datasets. Note the x-axis is log-scale. The table summarizes the trend for each dataset using the zlib compression algorithm. For each dataset, it lists the total number of rows, the average size of the value in each row, the maximum compression ratio achieved (on the entire dataset), and the average number of rows that must be in a single pack to achieve a compression ratio that is $\geq 75\%$ of the maximum compression ratio.

the key k. Figure 3 presents the overall procedure. The hash is a hash of the encrypted pack.

```
1: procedure GET(key)
2:     Fetch data from server:
       SELECT packID, value, hash FROM table
3:            WHERE packID ≤ key
              ORDER BY packID DESC LIMIT 1
4:     Decrypt and decompress value
5:     Return the entire row
```

Figure 3: get pseudocode.

4.2 Range queries

Range queries fit easily into MiniCrypt's design. In fact, for large range queries that touch many keys, MiniCrypt utilizes less network bandwidth than a regular key-value store because MiniCrypt compresses multiple keys together based on the query range. MiniCrypt performs a range query on packIDs using a key range [low, high]. Since a packID indicates the lowest key in the pack, a MiniCrypt client fetches the packIDs in [low, high]. If low is not equal to the smallest packID in the results, then MiniCrypt needs to fetch the pack that potentially contains keys from low to the smallest packID. The packs that contain key low and key high will need to be filtered by the MiniCrypt client as they can contain keys outside of the range [low, high]. The pseudocode for range queries is in Figure 4.

```
1: procedure GET(low, high)
2:     Fetch range from server:
       res ← SELECT packID, value, hash
3:                FROM table
                  WHERE low ≤ packID ≤ high
4:     Decrypt and decompress each value in res
5:     if low < smallest packID in res then
6:         Run get (low) and add to res
7:     Filter out keys not in [low, high] from res
8:     return res
```

Figure 4: get by range pseudocode

5. Writes in the generic mode

In the GENERIC mode, clients can perform any type of write from the API in Section 2.3. MiniCrypt additionally supports pack splitting because a pack may grow too large if there are repeated inserts, and retrieving overly large packs will increase bandwidth usage and hurt performance.

5.1 Put

Writes are more challenging than reads in MiniCrypt. Since the server cannot update an individual key in a pack due to encryption, each client has to retrieve an entire pack to execute a write. The client updates the value of the specific key in the pack, compresses the pack, encrypts it and writes it back to the database. However, if designed naïvely, concurrent clients may overwrite each other's updates if they are all modifying the same pack.

To prevent such contention, we rely on the *update-if* primitive explained in Section 2.5.1. We use this primitive and hashes to ensure that clients do not overwrite changes from each other as follows. Consider a client C1 who wants to update a key in a pack. The client reads the pack and records its hash h. C1 then updates the contents of the pack and issues an *update-if* at the server, specifying that the value should be updated only if the hash of the pack is still h. If the hash is no longer h, it means that another client C2 has recently updated the pack. Thus, C1 should not perform the update because it can overwrite C2's update. Instead, C1 will retry the update by performing a read of the current pack value. Figure 5 presents the overall procedure.

Another possible design that preserves eventual consistency is to do a blind write of the pack without any update-if mechanism. However, a read of the pack is still necessary to decrypt and modify the pack. As we show in the evaluation section, the extra read incurs significantly more cost than the update-if mechanism. Hence, we chose to use update-if because its cost on top of the extra read is not significant and it preserves better the original data store's semantics.

```
1: procedure PUT(key, value)
2:     repeat
3:         (packID, pvalue, h) ← get (key)
4:         if # keys in pvalue > max_keys then
5:             Do split as in Section 5.2. continue
6:         Inside pvalue, set key's value to value
7:         Compress and encrypt pvalue
8:         Compute its hash phash
           ok ← UPDATE table SET value = pvalue,
9:                                  hash = phash
               WHERE packID = packID IF hash = h
10:    until ok
```

Figure 5: put pseudocode

5.2 Split

Pack splitting is useful when there are inserts in the system that cause packs to grow large.

When to split a pack. Whenever a client runs put or delete, the client first checks the size of the retrieved pack after reading the pack. If the pack contains more than max_keys, the client proceeds to split the pack. The parameter max_keys is a system-wide constant, and can be set to $1.5 \cdot n$, where n is the desired number of keys in a pack. The client can proceed with the original operation once a split successfully completes.

How to split a pack. Figure 6 shows the pseudocode for split (packID, pack, h), where pack and the hash h are the retrieved values from reading packID. During split, the client divides the pack by creating a left pack from the first half of the keys (rounded up) and a right pack from the rest of the keys. Note that we require this operation to be deterministic so that every client that reads the same pack will divide the pack in exactly the same way. The client then compresses each pack and encrypts it as usual. It inserts the right pack and then the left pack, both using the *update-if* primitive.

The split procedure is safe in the presence of multiple clients as well as client failures. For example, two clients, A and B, may attempt to split the same pack. These clients will read the same pack, and then split in a deterministic way, resulting in the same left and right packs. If client A inserts the new right pack into the database first, client B will attempt to insert the same right pack. The second insert operation will fail because client A's insert has succeeded.

What if a client fails in the middle of a split operation? As an example, let's assume that a client fails its split right before step 5 of Figure 6. This means there are two copies of the new right pack's rows in the database: one copy in the newly inserted right pack, and one (stale) copy in the original pack. This is still safe because a new client that attempts to read/modify those keys will retrieve the newly inserted right pack instead of the stale copies. Any client that attempts to modify the original pack will execute the split procedure. Note that the new right pack will not be overwritten since we do not delete packs.

```
1: procedure SPLIT(packID, pack, h)
2:     Assemble the right half of the pack: right_pack with
       rightID and hash rh
3:     INSERT INTO table VALUES (rightID,
           right_pack, rh) IF NOT EXISTS
4:     Assemble left_pack with hash lh
       UPDATE table SET value = left_pack, hash = lh
5:             WHERE packID = packID
               IF hash = h
```

Figure 6: split pseudocode

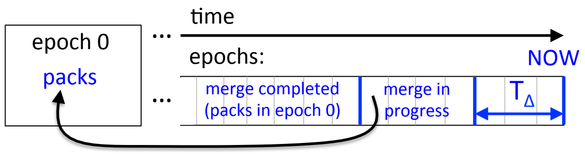


Figure 7: APPEND mode timeline.

5.3 Delete operations

delete is similar to put except that the key is removed from the pack. The ID of the pack does not change even when the lowest key in the pack gets removed. We do not remove packs when they become empty. The protocol for removing an pack is very complex because a split might reinsert a right half of the pack that was deleted.

5.4 Correctness

Recall that our notion of correctness was that MiniCrypt maintains the eventual consistency and liveness of the underlying key-value store. Due to space constraints, we only provide an intuition here.

The algorithm is safe when the operations are read-only. Updates that do not split the packs are also safe because of the hash check. The situation is trickier when there are concurrent updates with split operations. However, any client that issues a put or delete to a pack that has a number of keys greater than max_keys will block, since that client will choose to run split first. This allows concurrent modifications to safely co-exist with split. Furthermore, synchronization mechanisms such as "insert if not exists" and "update-if" ensure that concurrent splits on the *same* pack are safe. A client that is delayed during its split will not overwrite changes made after the split finished (due to other clients) to either of the two resulting packs. Finally, the liveness property is maintained because a client will not be stuck in an infinite loop — at least one live client will succeed to do a put or complete a split.

6. The APPEND mode

In APPEND mode, data is inserted into the system in order of roughly increasing keys. Clients can read keys, but no keys are updated or deleted once a certain time has elapsed since a key's first insertion. This mode fits applications whose writes are appends and enables these writes to be very fast. There are many APPEND mode use cases. For example, a common

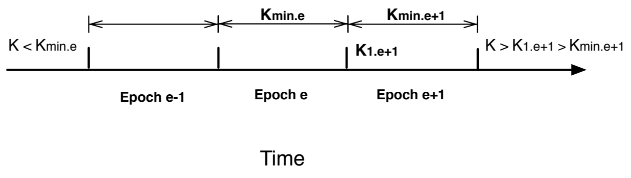


Figure 8: APPEND mode: an illustrated timeline showing the key constraints in each epoch. Let $k_{1.x}$ be the first key in epoch x, and $k_{min.x}$ be the minimum key in epoch x. The epoch time is designed to be large enough to provide guarantees that $k_{1.e+1}$ is less than all keys in epochs $e+2$ and beyond. This implies that $k_{min.e+1}$ is also less than all keys in epochs $e+2$ and beyond. We can also guarantee that $k_{min.e}$ is greater than all keys in epochs less than $e-1$, since $k_{1.e}$ is greater than all keys in those epochs. From this, we can guarantee that all keys between $k_{min.e}$ and $k_{min.e+1}$ occur in $e-1, e$ and $e+1$.

pattern for big data is time-series data [11, 27], where the keys are timestamps while the values are actions or events.

Let us define concretely MiniCrypt's assumption in this setting. First, MiniCrypt assumes that the keys are roughly inserted in order, but not in a perfectly increasing manner. MiniCrypt allows for a time lag in which keys do not appear (to get operations) in increasing order and requires that there is an upper bound on this lag denoted as $T_\triangle$. Let key k be a key inserted at time t_k. The assumption is, for every key k, that no key less than k is inserted beyond $t_k + T_\triangle$; similarly, no key greater than k is inserted before $t_k - T_\triangle$.

What is $T_\triangle$? $T_\triangle$ should be a conservative upper bound on the sum of the relevant time bounds, which include a bound on the time lag in which keys can arrive out of order (which could be due to the client), a bound on the network transfer time (the time it takes for a client request to reach the servers), and a bound on the server side time delay for new updates to be propagated to all available servers.

In the following section, we first present a basic design for the APPEND mode. We follow up with various improvements that can be made on the base design.

6.1 Design

The put operation is slow in the generic mode because MiniCrypt clients perform an extra read per put and employ synchronization (using *update-if*) to avoid overwrites due to concurrent put operations. To enable put to be fast in APPEND mode, MiniCrypt executes a put directly into the database (by compressing and encrypting a single row), and arranges for the clients to merge these inserts into packs in the background. In principle, this is possible because no key is updated or deleted once a certain amount of time has elapsed since it was first inserted, and a new key is not inserted between two such keys. In append mode, put should be very fast compared to regular put operations that do not operate on packs.

The main challenge that we face in APPEND mode is that the merge process can be expensive, but must keep the same rate as puts. Clients must be in charge of merging packs since the server cannot decrypt data. This means that the clients must read and delete a lot of keys, and that multiple clients might attempt to merge the same keys (potentially causing pack overwrites) while leaving other keys unmerged.

MiniCrypt addresses this challenge through a careful design of the system in APPEND mode. First, MiniCrypt groups keys into monotonically increasing *epochs*, which are useful in enabling batch processing. Each insert or update belongs to a single epoch. In many key-value stores, retrieving an entire epoch (e.g., if it is a partition as in Cassandra) and deleting the entire epoch is much faster than performing many server-side operations for reading and deleting each key.

We create a service called the *epoch management* (EM) service to manage epochs and the associated metadata. The EM service maintains a global epoch that is periodically increased. Each epoch is EPOCH seconds long. Clients periodically synchronize their local epochs with the global epoch. We define T_{drift} to be the maximum amount of time any client can be out of sync with the current global epoch. To maintain correctness, we need to make sure that EPOCH $> T_\triangle + T_{drift}$. In practice, T_{drift} is very small compared to EPOCH (we set it to be 10 seconds in our experiments), and we require any new/recovered client to synchronize its local epoch before doing an insertion.

Clients merge at the granularity of epochs in a deterministic way: sort the keys in an epoch and group them in packs of a given size starting with the smallest key. Thus, even if two clients concurrently merge the same epoch, the results are the same. The merged packs are placed in a special epoch 0, and the keys in the merged epochs are eventually deleted. Figure 7 shows the timeline of merge operations.

One way for the clients to merge epochs is to randomly pick an epoch to merge. Even though determinism ensures correctness, this method is inefficient because clients might do wasted work by merging the same epochs. MiniCrypt attempts to avoid having many multiple clients merge the same epoch by allowing the EM service to assign epochs to clients so that only one client is merging an epoch. If a client fails, the EM service will assign new clients to those potentially unmerged epochs.

6.1.1 The EM service

For availability, the EM service runs on the server. The EM does not see the contents of the packs, but only manages information about what client should merge what packs; hence, hosting the EM on the server does not pose a security issue within our threat model.

The EM maintains three pieces of information: the stats table, the clients table, and g_epoch, which we explain below. All this information is stored in the server database, so the EM is essentially a client of the underlying key-value store, and requires no changes to this store.

The `stats` table contains an entry per epoch of the form: epoch ID, client ID (the client that is in charge of merging the epoch), and status (current status of epoch, could be `NOT_MERGED`, `MERGED`, or `DELETED`). When a new client comes online, the client updates the `clients` table on the server by inserting its client ID and its local timestamp. Each client periodically refreshes that timestamp to indicate that it is still alive. The EM service periodically reads the `clients` table to assign clients to epochs, and to make sure that currently active clients are still alive. If a client times out, the EM service will scan through the `stats` table and assign all unmerged epochs with the failed client's ID to new clients.

The EM also maintains `g_epoch`, the current global epoch number, and updates it once every every `EPOCH` seconds.

6.1.2 Put

A `put` operation in `APPEND` mode is simply a single-row insertion of the key-value pair. When a client wishes to execute a `put`, it looks at its locally stored variable `c_epoch` for the current global epoch. The client loosely synchronizes the `c_epoch` variable with the server-side `g_epoch` by periodically querying `g_epoch`. In MiniCrypt, we adjust the period to be short enough so that `c_epoch` is at most one epoch behind of `g_epoch`. The client uses `(c_epoch, key)` as the new key and inserts `((c_epoch, key), value)` into the table, where `value` is compressed and encrypted.

6.1.3 Get

A `get` operation in `APPEND` mode is similar to the `GENERIC` mode `get`. A client first queries epoch 0 using the `GENERIC` mode query method. If the key is not found in the retrieved pack, the client retrieves the `stats` table, which additionally contains the minimum key for each epoch. The client finds the epoch with the largest "minimum key" that is smaller than the queried key. Let this epoch be e. Since the keys can be inserted roughly out of order, the actual record could be in either epoch e or $e-1$ (but not beyond $e-1$ since each epoch is long enough to cover $T_\triangle + T_{drift}$). The client will execute `get` for at most two epochs. Note that due to concurrent merges, one could miss the key if that key is merged right before either of the queries. Therefore, if both reads miss, the client performs an extra read of epoch 0 to attempt to find the key. Note that there are still cases where the key exists but is not found partly due to the fact that the underlying key-value store is itself eventually consistent, so the client must retry after a delay.

6.1.4 Merge

Each client's merge process periodically reads the `stats` table to find epochs that the client is responsible for. Consider a client that is responsible for merging epoch e. Because of the loose epoch synchronization on the client side and the fact that key inserts are also only roughly increasing in order, we cannot take keys from epoch e, order by key, and directly merge them into packs. We still wish to maintain the pack semantics – that each pack uses the minimum key as its pack id, and that all key-value pairs reside in the correct pack. An `APPEND` mode `get` should be able to use a range query to retrieve the correct pack for a query key once that key-value pair has been inserted. Therefore, the merge process begins by reading back all key-value pairs from $e-1$, e, and $e+1$. We use the *minimum key* from epochs $e, e+1$ (denoted $k_{min.e}, k_{min.e+1}$), as markers for deciding which keys to merge: all key-value pairs from $e-1, e, e+1$ with keys that are greater than or equal to $k_{min.e}$ but less than $k_{min.e+1}$ are grouped together and merged. Once the correct key-value pairs are retrieved, the merging process is easy: we simply order every key-value pair by key, then split them into packs based on a pack threshold. These packs are then inserted into epoch 0. After the packs have been inserted, the client updates the `stats` table with a status that the epoch has been merged by setting status to `MERGED`.

Each client also periodically deletes epochs. An epoch e can be safely deleted if its status is `MERGED` and epochs $e-1$ and $e+1$ are either `DELETED` or `MERGED`. After deletion, e's status is set to `DELETED`.

6.2 Fault tolerance for the EM

There need only be one EM machine running at a time because the job of the EM is light. For reliability, we distribute the EM service into multiple replicas. In our design, we assign each server replica an *EM instance*. One of these instances is the master EM, which is in charge of modifying the `stats` table and updating the global epoch. To survive EM master failures, the server contains an entry `EMreplica` identifying which replica is the master EM. The only task of the other EM replicas is to ping the master replica periodically to check if the master is alive.

If a replica believes the master is down, it updates `EMreplica` to designate itself as the master; this update is performed with an update-if, and thus relies on the underlying store's lightweight transactional mechanism to agree on the next replica to run the EM service. Hence, every update to `g_epoch` and the other EM data structures is performed using an update-if. This makes our design safe even if multiple EM replicas believe they are masters. We assume that the EM replicas' clocks are roughly synchronized (using a protocol such as NTP [29]), which means that multiple masters can safely update the global epoch by checking its local timestamp with the timestamp of the `g_epoch`. Since there is a further timing overhead from synchronization, the epoch time needs to be adjusted appropriately. Second, our merge protocol is deterministic, which means that it is safe even if multiple clients attempt to merge the same epoch.

6.3 Correctness

We provide an intuition for the correctness of `APPEND` mode. MiniCrypt provides eventual consistency guarantees because all replicas will eventually reach the same state after updates stop. MiniCrypt's protocols are also correct. The writes are regular single-key inserts, thus inheriting the correctness of the underlying system. The `merge` process preserves correctness for three reasons. First, there are no concurrent `put` and `delete` operations because clients only merge epochs that are two epochs older than `g_epoch`. We carefully choose the epoch time to ensure that the inserts/updates have settled and that the values will not change further (essentially becoming immutable). Second, `merge` operations by two clients on the same epoch result in the same outcome because the merge operation is deterministic. Since the client always reads the `stats` table (executing a read that reads back the *latest* status) before attempting to merge epochs, it will never attempt to merge a partially deleted epoch because the epoch's status is always set to `DELETED` before the actual deletion begins. Finally, the `merge` protocol first inserts keys in epoch 0 before deleting them; `get`-s are not affected because a `get` queries epoch 0 as well as unmerged epochs.

Figure 8 also shows an explanation of our append mode's correctness through an epoch timeline analysis.

7. Implementation

We implemented MiniCrypt in approximately 5000 lines of C++ code on top of an existing key-value store, Cassandra [25]. Cassandra is a widely used open-source key-value storage system that is both scalable and highly available. Our implemented interface does not make *any* internal modifications to Cassandra, and simply calls Cassandra's C++ driver. We used zlib to compress packs and OpenSSL AES-256-CBC to encrypt them.

Cassandra uses consistent hashing to distribute its data. Primary keys in Cassandra consist of a partition key and optional clustering keys. To find a particular piece of data, Cassandra simply hashes the partition key. Cassandra does not support range queries directly on the partition key, but clients can order rows within a partition based on the clustering keys. This allows for range queries within a Cassandra partition. MiniCrypt sorts data in descending order to optimize for get queries.

To support a generic key-value store interface, MiniCrypt makes some small adjustments to fit Cassandra's design. For a key-value pair `(key, value)`, MiniCrypt takes `key` and hashes it to a hash value. Using this hash value, MiniCrypt is able to assign the keys to `N` partitions. The default number of hash partitions is 8, though the user may adjust this parameter. Therefore, the primary key used in Cassandra is a key pair `(part_key, key)` where `part_key = SHA256(key) mod N`. Within each hash partition, the data is ordered by `key`. If a user wishes to perform `get(k)`, MiniCrypt will hash `k` to get a partition number, then use the get operation described in Section 4.1. For range queries, MiniCrypt has to make a range query request to each partition. In addition to a generic interface, MiniCrypt also has the ability to support a compound primary key `(partition key, clustering key)` (similar to what Cassandra supports). The primary key used in this situation is key pair `(part_key, key)` where `part_key = SHA56(partition key) mod N`.

8. Evaluation

Our experiments were conducted on Amazon EC2. All benchmarks were run on a small cluster of 3 c4.2xlarge instances, and each instance has 15 GB of memory and 8 cores. The SSD experiments were run with 2 provisioned SSD drives per instance. The disk experiments were run with 2 magnetic drives per instance. The Cassandra replication factor was set to 3. All benchmarks use the Conviva dataset row values [1], consisting of approximately 1100 bytes of anonymized customer data. All experiments use one 8-byte integer as the key, and one Conviva row as the value. Unless indicated otherwise, all MiniCrypt experiments (in both `APPEND` and `GENERIC` modes) set pack size to 50 rows.

We compare the performance of MiniCrypt with a baseline encrypted client. The baseline embodies a typical encrypted system that gives confidentiality guarantees by encrypting each row individually. This client has similar security to MiniCrypt, but it does not have the compression benefits of MiniCrypt. Nevertheless, we also use the same compression algorithm on each row before encrypting it to give this client an advantage. The compression ratio for single rows is roughly 1.6. Additionally, we compare MiniCrypt with a vanilla Cassandra client that uses no encryption and no compression for the 100% read and range query benchmarks.

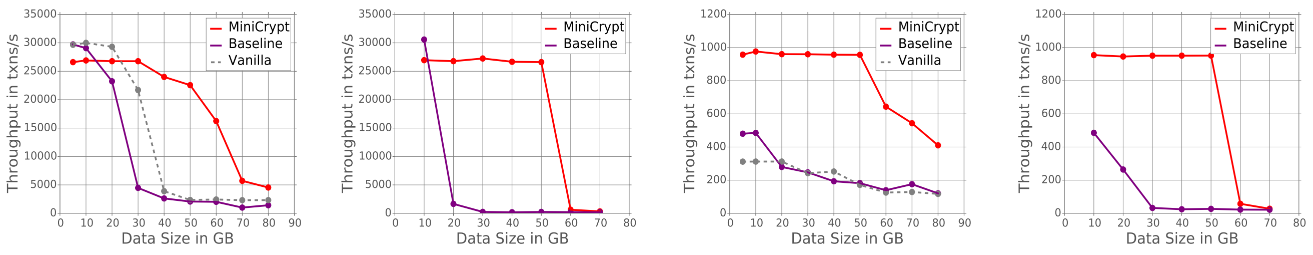
8.1 Read performance

We ran a modified YCSB read workload on a small three-instance cluster. We pre-loaded data into Cassandra and ran a 100% uniform read/range query workload on tables of different sizes. We compare MiniCrypt with a baseline encrypted client that compresses and encrypts each row separately, as well as a vanilla Cassandra client. The system was warmed up for 5 minutes for SSDs, and 10 minutes for disks. After warmup, each experiment was run for 60 seconds.

8.1.1 Point queries

Figure 9 plots the maximum server throughput (achieved by saturating the server with as many clients as possible) against varying overall dataset sizes. The same experiment was run with both SSD and magnetic disks.

We first compare MiniCrypt with the encrypted baseline client. When the dataset's size is small, both MiniCrypt and the baseline client fit the data in memory. The baseline client has a higher maximum throughput than MiniCrypt because the latter is retrieving more data and does extra processing (decompression, decryption) on the client. As the dataset size is slowly increased, the baseline client cannot fit in memory anymore. Once it starts accessing persistent storage for reads, the throughput drops significantly. Because



(a) **(b)** **(c)** **(d)**

Figure 9: Point queries on (a) SSD (b) disk; range queries on (c) SSD (d) disk

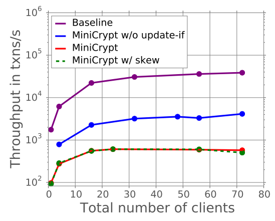


Figure 10: `GENERIC` mode 100% write

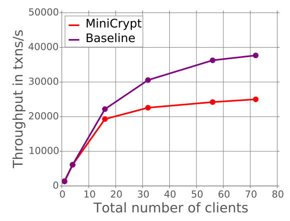


Figure 11: `APPEND` mode 100% write

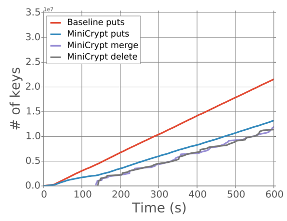


Figure 12: `APPEND` mode 100% write, long run

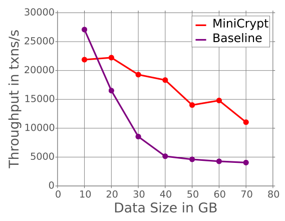


Figure 13: `APPEND` mode 50% read, 50% write

MiniCrypt achieves higher data compression ratio, the same dataset that does not fit in memory for the baseline client still fits in memory for a MiniCrypt client. Thus, MiniCrypt continues to maintain high throughput until the compressed data can no longer fit in memory. In this situation, MiniCrypt is able to achieve roughly *100x* performance gain over the baseline client when the server is backed by disk, and *9.2x* performance gain when the server is backed by SSD. If both clients cannot fit data in memory anymore, MiniCrypt still manages to maintain good performance for a while because a large majority of the data it accesses is still in memory. For larger data sizes, we expect a crossover point where the baseline client becomes better than MiniCrypt because the query overhead starts to be dominated by accesses to persistent storage. MiniCrypt is weak in this scenario because it accesses an entire pack for a single point query. Compared to the SSD graph, the disk graph has much sharper drops. This behavior is expected because disks have a significantly lower read throughput for uniform access than SSDs.

MiniCrypt is also able to achieve roughly *6.2x* performance gain over the vanilla Cassandra client (SSD). The vanilla client's graph is similar to the encrypted baseline's graph, except shifted to the right. Since Cassandra utilizes compression on the server side, it is able to compress plaintext value to a certain extent. However, the compression ratio Cassandra achieves is not as good as that of MiniCrypt. These graphs show that MiniCrypt provides a significant throughput increase for a significant range of data sizes.

To compare latency numbers for both MiniCrypt and the baseline, we ran the same 100% read benchmark on SSD for a MiniCrypt client and a baseline client (both of which are single threaded) on a database size of 5 GB. This latency measurement is advantageous for the baseline client because its data fits in memory. The baseline achieves 1.019 ms per query, and MiniCrypt achieves 1.199 ms per query. MiniCrypt's extra latency gains come from the extra processing on the client.

8.1.2 Range queries

Range queries are very common in many workloads. For example, time series data (such as session logs) are frequently append-only and immutable when inserted. The logs are later retrieved by time range. The Conviva analytical query workload also retrieves customer data within a time range, where the range can be as short as one hour and as long as one week.

MiniCrypt's design makes range queries very efficient because MiniCrypt orders all key-value pairs and groups them into packs. For a point query, the space overhead is (pack size / compression ratio). For range queries (especially large scans that touch many records), the bandwidth overhead is reduced. For example, if the number of records queried is significantly greater than the pack size, the baseline client will have more bandwidth overhead than MiniCrypt (by a factor of C, where C is the ratio of MiniCrypt's pack compression to a single row's compression).

Our range query experiments are based on YCSB's short ranges workload. Each query selects a key k uniformly from the keyspace, and attempts to query all items between $(k - 1000, k]$. Figure 9 shows that MiniCrypt *consistently* experiences a significantly higher maximum range query throughput compared to both the encrypted baseline client and the vanilla client, both when the data fits in memory and when it does not. MiniCrypt is able to achieve up to 5x performance gain over the encrypted baseline client. Note that the vanilla client is slightly slower than the encrypted baseline client in the 100% range query experiment for small database sizes. This is due to the vanilla client being bottlenecked by the network. The vanilla client may achieve compression in memory, but still has to return the result in uncompressed format. The baseline encrypted client is able to achieve some compression for a single row, and can therefore return the result in the compressed format as well. As the size of the database increases, disk becomes more of a bottleneck, and the vanilla client and the encrypted baseline client converge to same throughput.

These experimental observations align with our analysis. Since our range is 1000 records, MiniCrypt is able to achieve much better performance because the compressed data has a higher compression ratio. When data can no longer fit in memory, the performance drops because of disk accesses. The drop is more significant for disk than for SSD.

8.2 Write performance

Generic mode. In `GENERIC` mode, each write has two overheads: an extra read and a lightweight transaction (*update-if*). Figure 10 shows the result of running 100% uniform random writes on a pre-loaded 10 GB database, with the y-axis on a log scale. Each experiment ran for 120 seconds. The baseline client is fast because it is able to execute blind writes. MiniCrypt `GENERIC` mode with update-if is slow, but is mainly dominated by the extra read while the usage of the lightweight transaction introduces further stress to the system. This experiment justifies our decision to use the update-if mechanism because it gives much better guarantees of the system's semantics. Even if we revert the system to do blind writes, each write still requires an extra read, which is where the performance loss is coming from. Figure 10 also shows a skewed workload that is generated using a Zipfian distribution. The Zipfian parameter is set to 0.2 (with 0 being pure Zipfian, and 1 being uniformly random). The skew has almost no effect on the write performance.

Append mode. Our append mode writes increase the performance of `put` by several orders of magnitude. We ran two sets of experiments in MiniCrypt `APPEND` mode: modified YCSB 100% write and 50% read/50% write. Under `APPEND` mode assumptions, all writes are actually inserts where inserted keys are roughly increasing. Each experiment is run for 120 seconds (except for the long 100% write).

Write-only. We start with an empty database for both the encrypted baseline and MiniCrypt. Figure 11 compares the baseline client with MiniCrypt in `APPEND` mode. Compared to Figure 10, MiniCrypt is able to keep up with the baseline client's put speed much better because `put` in `APPEND` mode does not have an extra read and does not use *update-if*. The difference between MiniCrypt's and baseline's throughput is due to the overhead of the merge process. This overhead is not visible when the number of clients is small, but it

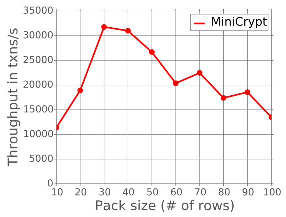


Figure 14: Pack size versus maximum throughput.

appears when the number of clients increases. The merge process has to read back inserted keys, as well as re-insert them (though in a compressed format). This interferes with regular inserts because of both disk reads and extra insertion costs. MiniCrypt settles to about 40% of maximum write throughput achieved by baseline client.

Figure 12 shows the performance of MiniCrypt for a long run (approximately 10 minutes). We scale up the number of clients to 72, which corresponds to the right most data point in Figure 11. This graph plots cumulative number of keys against time. The baseline client line shows cumulative number of keys inserted during the 10 minute run. MiniCrypt has three different lines: "insert", "merge", and "delete". Insert indicates the cumulative number of keys inserted during the run; "merge" is the total number of keys merged from the inserted keys; "delete" is the total number of keys deleted. This graph shows that the merge process is able to keep up with key insertions, albeit at a lower insertion rate than that of the baseline client.

Read/write mix. The read/write mix workload is aimed to emulate one of YCSB's "read-most-recent" workloads, which is a common case when a workload inserts new data. All of the runs were executed on a pre-loaded 70 GB database. We adjust an "interval" parameter that indicates the range of the keys read. For example, an interval of 5 GB will allow the clients to read a uniformly random key from the most recently inserted 5 GB worth of data. Both the baseline client and MiniCrypt were warmed up for 5 minutes before each run. Each experiment runs for 120 seconds.

Figure 13 shows MiniCrypt's performance for a 50% reads and 50% writes workload. Since writes are faster than reads, the baseline's performance settles to a point that is higher than the baseline's performance in Figure 9 (a). The performance of MiniCrypt `APPEND` falls off as the size of the query interval increases because the merge process also needs to read recently inserted values. If more values are read into memory in the normal benchmark, the two processes will interfere with each other.

We also benchmarked the latency of the append MiniCrypt client as compared to the baseline client (again using single-threaded clients) on 5 GB of data. The baseline achieves a latency of 1.103 ms per read query, and 0.718 ms per write query. MiniCrypt achieves a latency of 1.743 ms per read query, and 0.781 ms per write query. The writes are very fast for both clients because both execute appends, while MiniCrypt's read is slower because it may have to do more work if an initial attempt to read misses.

8.3 Determining the pack size

Writing an equation for the optimal pack size is not feasible because there are too many factors that affect this choice. Instead, MiniCrypt provides a tool to empirically determine a good pack size. This tool takes in a representative dataset and workload and can generate a graph of throughput plotted against various pack sizes. MiniCrypt then chooses the pack size that provides the highest throughput. Figure 14 shows running YCSB 100% uniform read workload for 50 GB of Conviva data. In our experiments, we noticed consistently that the optimal pack size was the following: the smallest pack size for which the data fits in memory, namely, $\mathsf{argmin}_n\{\mathsf{comp ratio}(n) \cdot \text{data size} < \text{memory size}\}$, where $\mathsf{compratio}(n)$ is the compression ratio obtained when compressing a pack of size n.

We recommend using MiniCrypt when all or most of the data fits in memory when compressed by MiniCrypt (i.e., fits in memory on each machine), but would not fit in memory without MiniCrypt. We show in the previous sections that there is a significant data size range when this is the case. If a significant fraction of the data does not fit in memory, we do not recommend using MiniCrypt.

8.4 Network bandwidth

MiniCrypt's network bandwidth overhead can be determined by (# of rows in each pack / pack compression ratio). In our experiments, network bandwidth did not become a bottleneck. We expect MiniCrypt to be used in a setting where the network is not the bottleneck.

9. Related work

We now discuss other related work in addition to the strawman designs described in §2.4.

Key-Value Stores. Some key-value stores (*e.g.*, Cassandra [25] and MongoDB [3]) compress and then encrypt the data at rest (in permanent storage). However, the decryption key is available to the server so that the server can decrypt and decompress the data when a client requests a key. This strategy does not protect against a server compromise (e.g., hacker, administrator of the server) because the attacker can get access to the decrypted data by compromising the server-side key. On the other hand, if a client inserts data encrypted with a key unavailable to the server, the compression mechanisms in these systems become ineffective due to the pseudorandom properties of the encryption. In comparison, MiniCrypt provides a significant compression ratio even in this case.

A recent system, Succinct [6] supports compression for a key-value store, while enabling rich search capabilities. However, Succinct does not support encryption. Adding encryption directly on top of Succinct would cause significant data access pattern leakage.

Encrypted databases. A number of recent proposals in databases support queries on encrypted data [7, 32, 33]. However, encryption introduces a significant storage overhead compared to the unencrypted data (e.g., 5 times larger for [32]). These systems do not support compression while executing queries on encrypted data. As presented in the MiniCrypt evaluation, querying on data that does not fit in memory will cause a significant performance hit.

Compressed databases. Compression is a common technique explore in databases [5, 6, 8, 15, 21, 26]. We discuss some simple strawman designs that utilize these techniques in Section 2.4. MiniCrypt is a generalized system that does not rely on a particular compression algorithm – users can choose any algorithm that fits their application requirements.

MiniCrypt differs from these two types of databases in two main ways. First, it focuses on NoSQL stores and does not support the more generic SQL operations. Second, MiniCrypt achieves both data confidentiality through encryption and a significant compression ratio.

File systems. There is previous work on designing encrypted file systems [9, 19, 22, 28] to protect data confidentiality from an untrusted server. One can compress a file before encrypting it. However, as discussed, compressing a single key-value pair alone does not provide good performance.

10. Conclusion

We presented MiniCrypt, the first big data key-value store that reconciles encryption and compression. MiniCrypt makes an empirical observation about data compression trends and provides a set of distributed systems techniques for retrieving, updating, merging and splitting encrypted packs while preserving consistency and performance. Our evaluation shows that MiniCrypt can increase the server's throughput by up to two orders of magnitude.

Acknowledgments

We thank our anonymous reviewers, our shepherd Mahesh Balakrishnan, and Aurojit Panda, for their valuable feedback or discussions. This work was supported by the Intel/NSF CPS-Security grants #1505773 and #20153754, DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, the UC Berkeley Center for Long-Term Cybersecurity, as well as gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] Conviva. `http://www.conviva.com/`.

[2] Memcached. `http://www.memcached`.

[3] Mongodb. http://www.mongodb.org.

[4] Redis. `http://www.redis.io`.

[5] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[6] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, May 2015.

[7] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1033–1036. ACM, 2013.

[8] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 283–296. ACM, 2009.

[9] M. Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, Nov. 1993.

[10] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2009.

[11] K. Challapalli. The internet of things: A time series data challenge. In *IBM Big data and Analytics Hub*, 2014.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[13] P. R. Clearinghouse. Chronology of data breaches. http://www.privacyrights.org/data-breach.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Googles globally-distributed database. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[15] Daniel J. Abadi and Samuel R. Madden and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *ACM International Conference on Management of Data (SIGMOD)*, 2008.

[16] G. Danti. Linux compressors comparison on CentOS 6.5 x86-64: lzo vs lz4 vs gzip vs bzip2 vs lzma. In *Hardware and software benchmark and analysis*, 2014.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazons highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[18] J. Ellis. Lightweight transactions in cassandra 2.0. http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0.

[19] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, pages 131–145. Internet Society (ISOC), Feb. 2003.

[20] O. Goldreich. *Modern cryptography, probabilistic proofs and pseudorandomness*, volume 1. Springer Science & Business Media, 1998.

[21] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 389–400. ACM, 2007.

[22] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, Apr. 2003.

[23] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 60, 2016.

[24] L. King. Why compression is a must in the big data era. In *IBM Big Data and Analytics Hub*, 2013.

[25] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. In *ACM SIGOPS Operating Systems Review, 44(2):3540*, 2010.

[26] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[27] B. Lorica. The re-emergence of time-series. In *O'Reilly Radar*, 2013.

[28] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. Technical Report TR2002–826, NYU Department of Computer Science, May 2002.

[29] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[30] J. Podesta. Findings of the big data and privacy working group review. In *The White House Blog*, 2014.

[31] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 463–477, San Francisco, CA, May 2013.

[32] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[33] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, 2017.