MiniCrypt: Reconciling encryption and compression for big data stores

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

ABSTRACT
More and more applications and web services generate larger and larger amounts of confidential data, such as user and financial data. On one hand, these systems must use encryption to ensure confidentiality, while on the other hand, they want to use compression to reduce costs and increase performance. Unfortunately, encryption and compression are in tension, leading many existing systems to support one but not the other.

We propose MiniCrypt, the first big data 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 in small key packs, together with a set of new distributed systems techniques for retrieving, updating, merging and splitting encrypted packs while preserving consistency and performance. 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 an order of magnitude by fitting more data in main memory.

1 Introduction
Many applications today capture an immense amount of private data about their users [28], such as purchasing history, social interactions, demographic information, and communication patterns. This private data is stored at servers running highly performant data stores [2, 4, 17, 3, 12, 14, 23, 6]. Unfortunately, leakage of confidential data from these application servers remains a significant problem [13].

Hence, such applications need systems that are both efficient at handling big data and able to preserve the data's confidentiality. To improve efficiency, many big data stores employ compression [22, 12] to significantly increase performance, sometimes by up to an order of magnitude [6]. These gains come from the fact that compression enables servers to fit more data in main memory, and helps to decrease both disk and network transfer times. To protect data confidentiality, a natural solution is to encrypt the data stored on servers [30, 9, 19, 21]. The decryption key can be kept on the client's side so that if an attacker (e.g., hacker, server administrator) gets access to all the data on these servers, the attacker cannot decrypt the data. Hence, an ideal system should incorporate both encryption and compression into its design. Unfortunately, as we now explain, existing systems choose either compression or encryption, but not both.

There is a fundamental tension between encryption and compression. First, putting compression on top of strongly encrypted data (e.g. randomized encryption) 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, 20] 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. 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 multiple rows of data. However, encrypting large compressed chunks is problematic for key-value stores because it hurts performance. Since the server cannot decrypt, all 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 benefit from key-value stores, such as data from genomics, Twitter, Conviva, gas sensors, Wikipedia, and GitHub. For example, for a dataset consisting of anonymized user data from Conviva (internet-scale video access logs) [1], we consider a case where only values are compressed. Compressing 1 row yields a compression ratio of 1.6, compressing 25 rows yields a ratio of 4.1, compressing 50 rows yields a ratio of 4.6, and compressing 8.7 million rows (namely, the entire dataset) yields a compression ratio 5.1. This shows that compressing only a fraction (0.0003%) of the total number of rows already provides 80% of the maximum compression ratio.

Our system model assumes a cloud-hosting context, where there are two roles: the hosting service (the server), and the company/organization that uses the hosting service (the customer). There could be multiple client machines within the same trust domain of the customer, and they share a single encryption key, using which they want to hide the data of the customer from the cloud provider. 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 inherit their performance, fault tolerance, and consistency. In particular, 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.

Based on the 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 decompact the data to read or update a key-value pair. Maintaining these properties turns out to be a challenging database problem, which we address through a set of new techniques.

The first challenge is that the server can no longer serve or update individual keys, as it cannot decrypt the pack. To fetch the value for a key, we provide a simple mapping scheme that allows a client 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. Regarding updates, concurrent clients could overwrite each other’s update when trying to update different rows: the reason is that these rows could fall in the same pack. We propose an algorithm for preventing such conditions based on a lightweight single-row synchronization primitive, a primitive which is provided by virtually any existing key-value store. Furthermore, since many big data workloads are mostly appends (e.g., time-series data), we provide a tailored protocol for appends based on a fast pack management protocol, achieving a write throughput close to the throughput of a regular key-value store.

The second challenge MiniCrypt addresses is the server’s difficulty in managing encrypted key packs. As rows get inserted or removed, some packs could become too small or too large, impacting the compression ratio or performance. Hence, these packs need to be split or merged. Pack management is left to the clients, who can fail or act concurrently. 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), making it difficult to resolve these issues using concurrency control. Instead, we provide two new algorithms to merge and split packs using only a lightweight form of transaction for a single row.

We have implemented MiniCrypt on top of Cassandra [23] without modifying the internals of Cassandra. 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 (encrypted Cassandra without MiniCrypt’s compression) and a vanilla version (regular Cassandra with no encryption and extra compression) by fitting more data in main memory. For example, on the Conviva dataset, the read throughput increases by up to 100 times (disk-backed) and 9.2 times (SSD-backed) compared to the encrypted baseline, and even up to 6.2 times (SSD-backed) 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 multiple 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.

The threat model is that an attacker managed to gain access to the server, can see all the data there (including any messages/queries from clients), and may leak the data. We assume that the attacker is passive and does not corrupt the data or answers queries incorrectly (e.g., a curious system administrator). We assume that all the clients are allowed to see all server side data, and that the clients are trusted not to disclose it to the server. In this paper, we are concerned with protecting data confidentiality from the server.

Many applications implement finer client access control to ensure that certain clients have access to only a part of the data. Such access control can be implemented in various ways. For example, a proxy sitting between the clients and the server allows a client to obtain results for a subset of the queries. Another example is to have the clients maintain different keys for each group of data with the same access permissions. 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.

2.2 Goals

MiniCrypt has the following design goals. First, given a key-value store where the values are encrypted, MiniCrypt aims to add compression to this system, while maintaining the confidentiality of the values in key-value pairs. MiniCrypt aims to provide strong confidentiality and the added compression should not weaken notably the security of the system. Second, MiniCrypt aims to provide significant compression without compromising performance. For example, compression promises higher read throughput because more data can fit in memory than on disk, and MiniCrypt should indeed achieve such a higher throughput than a standard encrypted service without compression. 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 inherit their performance, fault tolerance, and consistency. MiniCrypt is a generic solution and not tied to a specific key-value store. Finally, MiniCrypt should preserve the semantics of the underlying key-value store (this includes correctness, consistency, etc) despite the encryption and compression.

2.3 System API

MiniCrypt exposes the basic key-value store API, as well as support for range queries. Range queries get (low, high) are common for time-series big data [25, 11]. Some key-value stores enable basic SQL-like queries. One can support such queries with MiniCrypt too, but for clarity, we focus on this simpler API in the paper.
In Section 3, we discuss our observation on the rapid convergence of the compression ratio as the number of rows, \( n \), in a pack increases, and explain how MiniCrypt selects \( n \).

To read a key, a client fetches the corresponding pack, decrypts and decompresses it. To write a key, a client needs to update 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.

MiniCrypt aims to add compression to a key-value store with encrypted values, while maintaining the confidentiality of the values in the original system. As such, MiniCrypt encrypts each pack with a strong and standard encryption scheme (AES-256 in CBC mode). Thus, this encryption protects the values in the original key-value store, and as a bonus, the keys contained within a pack. This randomized encryption scheme leaks nothing about the values, except for the byte size of the pack. At the same time, pack encryption hides the lengths of the original rows and the keys in the pack. MiniCrypt can reduce the information leaked by the size of the pack by padding the encrypted packs to a tier of a few possible sizes (e.g., small-medium-large or exponential scale), which retains significant compression.
or measurements. For example, Spotify uses Cassandra 
the keys are timestamps, while values are typically actions
2.5.2
MiniCrypt focuses on encryption and compression of val-
ues and not keys. MiniCrypt does not prescribe what en-
cryption scheme to use for packIDs. Instead, MiniCrypt
can likely use the same encryption (and hence provide the
same security) for packIDs as for keys in the original system
because the same operations run on packIDs as on keys.

Option 1: strong encryption. When packIDs are en-
crypted with strong encryption, the only property needed
from the encrypted packIDs is that the encryption is deter-
mistic so that different clients encrypting the packID can
refer to the same encryption. This is easy to implement by
using a block cipher (e.g., AES or Blowfish) without a fixed
initialization vector (IV) because these schemes are assumed
to be pseudo-random permutations; fortunately, in our set-
ting, such an encryption is virtually as secure as a strong
randomized encryption (so the lack of a random IV does
not weaken security) because the packIDs are unique.

MiniCrypt creates the packs based on the order of the
encrypted keys, as opposed to the order of plaintext keys.
The packID is the smallest encryption value, which might
do not correspond to the smallest key in the pack.

In generic mode, the design is entirely unchanged, except
that range queries are not supported. Note that get works
the same as before because the SELECT query in Figure 3 now
simply operates over the encrypted keys. In append mode,
MiniCrypt keeps the merged packs in the epochs (and no
longer moves them to epoch 0); moreover, the stats table
contains encrypted metadata to help the client find which
epoch contains the key of interest.

Option 2: order-preserving encryption. The case
when the packIDs are encrypted with order-preserving en-
cryption is trivial: nothing changes in our design. The
queries and algorithms MiniCrypt runs work the same on
encrypted packIDs as on unencrypted packIDs because the
encryption preserves the order, while hiding the value of
packID.

Performance. In Option 1, one can use Blowfish which
produces 64-bit ciphertexts. In Option 2, using the scheme
of [10], the ciphertexts are also 64 bit long. We ran

<table>
<thead>
<tr>
<th>Mode</th>
<th>Type of writes</th>
<th>Pack op.</th>
<th>Performance notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>all types</td>
<td>split</td>
<td>put uses update-if</td>
</tr>
<tr>
<td>Append</td>
<td>append, put, no delete</td>
<td>merge</td>
<td>very fast appends</td>
</tr>
</tbody>
</table>

Table 1: Comparison between MiniCrypt’s modes. Pack op. denotes the pack maintenance operation.
our throughput experiments with such encryption schemes. Since they produce the same ciphertext size, there is no noticeable difference in the server’s throughput when using either of these. The latency on the client-side is different: Blowfish runs in 0.0001ms per encryption, and OPE in 4ms per encryption in our experimental setup.

3 Key packing

In this section, we justify empirically 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 6 datasets we examined are: anonymized Conviva time-series data regarding user behavior, genomics data where each entry consists of an identifier and a sequenced human genome, time-series Twitter data consisting of tweets and metadata, time-series data from gas sensors, Wikipedia files, and Github files for the source code of Linux. 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 dependency of the compression ratio on the number of rows in the pack. For each set of data, we re-format it into key-value pairs. We then group the values into packs by adjusting a maximum threshold (in bytes) for each pack. Finally, we calculate the average number of values present in each pack. The x-axis shows that average number of values, while the y-axis shows the compression ratio.

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. We can see from Figure 2 that the compression ratio grows very fast in the number of rows, then plateaus quickly close to the compression ratio for 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 compressing/decompressing. 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 zlib for MiniCrypt: it achieves both a good compression ratio and good performance.

4 Read operations

In this section, we describe read operations in MiniCrypt, which work the same in both the generic and append mode.

4.1 Get

Since key-value pairs are packed and encrypted in MiniCrypt, clients can fetch only at the granularity of a pack. A question is: how does the client know the packID of interest? One option is to maintain a table mapping keys to packIDs at the server; this strategy is undesirable because it adds space, increases latency due to queries to this table, and is hard to keep consistent 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 there is an ordered index on packID: the server locates $k$ by retrieving the key immediately preceding it. Once the client receives the result of this query, it decrypts and then decompresses the result. Then, it scans the content and retrieves the value for the key $k$. Figure 3 presents the overall procedure.

4.2 Range queries

Range queries fit easily into MiniCrypt’s design. In fact, for large range queries that touch many keys, the total bandwidth is smaller in MiniCrypt than in 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, it means that the MiniCrypt client needs to fetch one more pack, the one 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 8.

5 Writes in the generic mode

In the generic mode, clients can perform any type of write from the API in Section 2.3. A pack may grow if there are inserts. Retrieving overly large packs may hurt performance, so MiniCrypt splits a pack once it reaches a certain size.

5.1 Put

Writes are more challenging than reads in MiniCrypt. Again, since the server cannot update an individual key in a pack due to encryption, each client has to first retrieve the pack before performing a write. The client updates the value of the specific key in the pack, compresses the pack, encrypts it and uploads it at the server. If designed naively, a concurrent client may overwrite updates from a different client to another key within the same pack, thus violating consistency semantics.

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

Figure 3: get pseudocode.

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

5.2 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. When a pack becomes empty, the client cannot remove it right away because it interferes with our split protocol: a delayed client performing a split might reinsert a right half of a pack that was deleted during the delay. To prevent this race condition, we need to make sure that no client performing a split is delayed until after the time of deleting the right half of the pack.

To achieve this goal, we make the assumption that a client will not be delayed by more than $T_{delay}$, where $T_{delay}$ can be set to a large value (e.g. 24 hours). Background processes similar to merge in APPEND mode can be used to delete empty packs if the packs’ timestamps are older than server’s current timestamp - $T_{delay}$.

5.3 Split

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 15-$n$, 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 from a previous read of 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. It compresses each pack and encrypts it as usual. It inserts the right pack and then the left pack, both using the update-if primitive.

5.4 Correctness

Recall that our notion of correctness was that MiniCrypt maintains the same semantics and the liveness of the underlying key-value store. We now present an intuition for why these properties are preserved.

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

Figure 6: split pseudocode

Figure 7: APPEND mode timeline.

other clients) to either of the two 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 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. (MiniCrypt can actually enable updates, inserts and deletes in append mode, as explained in Section 6.1, but for simplicity, we assume they are not allowed here.) This mode fits applications whose writes are append and enables these writes to be very fast. There are many APPEND mode use cases. A common pattern for big data is time-series data [25, 11], where the keys are timestamps while values are some action or event. For example, Spotify uses Cassandra [27]: it does not store playlists but instead it stores pairs ⟨timestamp, song added⟩ to reconstruct the playlist from this history.

Let us concretely define the assumption MiniCrypt makes in this setting. Since no append-based application inserts keys in a perfectly increasing manner and server replicas are not perfectly synchronized, MiniCrypt allows for a time lag in which keys do not appear (to get operations) in increasing order, but it requires that there is an upper bound on this lag denoted as $T_\Delta$. Concretely, $T_\Delta$ must be chosen to be an upper bound on the sum of the difference in timestamps at the replicas, an upper bound on the time ($t_\Delta$) during which keys are inserted out of order, and the time it takes for the underlying system to propagate a modification (put or delete) to all client get-s. $t_\Delta$ is a value that satisfies the following: for any key $k$ with timestamp $ts_k$ (assigned by a server replica when first inserted), if $ts$ is the current timestamp at any of the replicas and $ts_k < ts - t_\Delta$, then the application does not insert, update or delete any key $k'$ with $k' \leq k$.

6.1 Design

The put operation is slow in generic mode because MiniCrypt clients employ synchronization (using update-if) to avoid overwrites due to concurrent put operations, as well as requiring all put operations to perform an extra read. 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. Now each put is virtually as fast as in the underlying system.

The challenge, though, is that the merge process can be expensive and consume significant server throughput: clients read and delete a lot of keys, multiple clients might try to merge the same keys while leaving other keys unmerged, and some synchronization might still be required. We now explain our append design and how it overcomes these challenges.

First, MiniCrypt divides keys into epochs of time based on the timestamp of the keys. Epochs are useful in enabling batch processing. 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. Each epoch is EPOCH seconds long, where EPOCH > $T_\Delta$ + $T_{dafia}$. We define $T_{dafia}$ to be the maximum amount of time any client can be out of sync with the current global epoch. 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.

Second, MiniCrypt attempts to avoid having many multiple clients merge the same epoch. While the determinism ensures correctness, such a behavior can cause performance degradation: in a previous version of MiniCrypt where clients could choose randomly which epochs they merge, clients wasted a significant fraction of the server’s throughput performing repeated merges, while the number of unmerged epochs increased. Even if only two clients merge the same epoch on average, the merge throughput is decreased by half. To address this problem, MiniCrypt introduced a special service called the epoch management service (EM). The EM assigns epochs to clients, ensures that only one client is merging an epoch, and deals with client failures. We now explain how the EM service works.

The EM service. The EM maintains a stats table on the server with entries 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 periodically reads the clients table to assign clients to epochs, as well as making sure that currently active clients are still alive. If a client times out, the EM will scan through the stats table and assigns all unmerged epoch with the failed client’s ID to new clients.

The server database keeps $g_{epoch}$, the current global epoch number. The EM updates $g_{epoch}$ once every EPOCH seconds using update-if. This operation is lightweight because the update-if is only periodically executed.

For availability, the EM service runs on the server. Nevertheless, our design of the EM makes no changes to the
underlying key-value store because the EM is essentially a client application that runs by only invoking the server’s API. The EM can also run on clients instead of the server. Due to higher churn on the client side, it is more efficient to run them on the server.

For reliability, it is easy to distribute an EM service into multiple replicas. In our design, we assign each server replica an EM instance. One of these instances is the master EM, the only one that changes the stats table. 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 essentiality relying on the underlying store’s lightweight transactional mechanism to agree on who is the next replica to take charge of the EM service.

**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.

**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. The keys can get roughly out of order, so the actual key could be in either epoch e or e – 1. The client will execute get for at most two epochs. Note that due to concurrent merges, one could miss the key in this step. Therefore, if both reads miss, the client performs an extra read of epoch 0 to attempt to find the key.

**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. 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 ([k_min,e, k_min,e+1]), as markers for deciding which keys to merge. All keys from e – 1, e, e + 1 that are greater than or equal to k_min, but less than k_min, 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. It retrieves the stats table and scans the table for possible epochs that can be deleted. An epoch e can be deleted if its status is MERGED and epochs e – 1 and e + 1 are either DELETED or MERGED. After an appropriate epoch e is chosen, the entire epoch is deleted and the status of the epoch is set to DELETED.

**Updates.** Though APPEND mode is designed mainly for inserts of new keys, updates such as put (insert, update) or delete on existing keys (past the update lag period) can happen as well. These operations must happen only on keys that are already merged, namely, are in epoch 0. The protocol for these operations is the same as the generic mode: put, get and split can happen.

These operations will not be as fast as append; the assumption here is that these operations are rare in a system where append is the main form of writing. The reasons these operations will be slower than append are as follows. As in the generic mode, these operations will need synchronization. Moreover, if a client performs a put or delete to a key in the epochs that are undergoing merging or in the epochs that are less than TΔ away from the current timestamp, the operation must wait until the corresponding epoch gets merged and the packs are moved to epoch 0. Since the merge process is designed to keep up with the append rate, there should be a constant time interval during which epochs are undergoing merging – this time interval should not grow with data size. If packs become too small, they are not merged again. In a system where put and delete are rare operations and append is the main form of writing in this system, there should not be many small packs.

6.2 Correctness

We provide an intuition for why our append protocol preserves the semantics and liveness of the underlying key-value store. 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 choose 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 results 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 should have been set to DELETED. Finally, the merge protocol first inserts keys in epoch 0 before deleting them; get-s are not affected because a get looks in epoch 0 as well as unmerged epochs.

7 Implementation

We implemented MiniCrypt in approximately 5000 lines of C++ code on top of an existing key-value store, Cassandra [23]. 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 modification to Cassandra. MiniCrypt uses the Cassandra’s
C++ driver to interface with the storage.

We used the zlib algorithm to compress packs and OpenSSL AES-256 to encrypt packs. To encrypt pack-IDs and keys in append mode, we provide both the option of using AES and the order-preserving encryption scheme from [30].

Cassandra uses consistent hashing to distribute its data. Primary keys in Cassandra consist of a partition key and zero or more clustering keys. To find a particular piece of data, Cassandra takes the partition key and hashes it. Using this hash value, the system is able to determine which node contains the data. Cassandra does not support range queries directly on the partition key. Instead, users can order rows that have the same partition key based on the clustering keys. This allows for range queries within a Cassandra partition. Cassandra can order the rows in either ascending or descending order. MiniCrypt sorts data in descending order to optimize for get queries.

MiniCrypt’s implementation follows closely with that of Cassandra. 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 put the keys into 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) where part = SHA56(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. In addition to a generic interface, MiniCrypt also has the ability to support a compound primary key (part, key, key) where the primary key used in this situation is key pair (part, key) where part = SHA56(partition key) mod N.

Cassandra supports an update-if primitive that is used in generic MiniCrypt put operations [18]. This mechanism is a lightweight transaction that uses a 4-round Paxos to perform the conditional update. The Paxos operation is done on a single partition among all replicas.

8 Evaluation

Our experiments were conducted on Amazon EC2. All benchmarks are performed 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 is set to 3. All benchmarks use the Conviva dataset row values [1], consisting of approximately 1100 bytes of anonymized customer data. All experiments use a 8-byte integer as the key, and each value is the Conviva row value.

We encrypt using AES256 in CBC mode from OpenSSL, and we compress using zlib. The zlib algorithm is a well-rounded compression algorithm that provides both good compression ratio and fast compression/decompression speed. Unless indicated otherwise, all MiniCrypt experiments (in both APPEND and GENERIC modes) set pack size to 50 rows.

The pack size experiment is run only with MiniCrypt; all other benchmarks compare the performance of MiniCrypt with a baseline encrypted client. The baseline embodies how a system typically gives confidentiality guarantees by encrypting each row individually. This client has the same level of security as MiniCrypt, but it does not have the compression benefits of MiniCrypt. Nevertheless, we also use compression on the single 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 (SSD only).

8.1 Read performance

We ran a modified YCSB read workload on a small three-instance cluster. We pre-load data into Cassandra and run 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 is warmed up for 5 minutes for SSDs, and 10 minutes for disks. After warmup, each experiment is run for 60 seconds.

8.1.1 Point queries

Figure 8 plots the maximum server throughput (achieved by saturating the server with as many clients as possible) against varying overall dataset sizes. The same experiment is run on both SSD and magnetic disks.

We first compare MiniCrypt with the encrypted baseline client. When the dataset’s size is small, both MiniCrypt and baseline client fit 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). As the dataset size is slowly increased, baseline client cannot fit in memory anymore. Once it has to start accessing persistent storage for reads, the throughput drops because it needs to access data that is not resident in memory anymore. Because MiniCrypt compresses data, the same dataset that wouldn’t fit in memory for the baseline client can still fit in memory for a MiniCrypt client. Thus, MiniCrypt continues to maintain good 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 due to the fact that a large majority of the data it accesses is still in memory. The SSD graphs do not show a crossover point for the amount of data we had. 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 sharper drops. This behavior is expected because disks have a significantly lower read throughput for uniform access than SSDs. Once the data cannot fit in memory anymore, the disk immediately becomes the bottleneck and the maximum throughput drops drastically.

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 can achieve is not as good as MiniCrypt.

Hence, these graphs show that MiniCrypt provides a significant throughput increase for a significant range of data sizes — in which most of the data in MiniCrypt fits in memory, but not in the encrypted baseline/vanilla 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. Conviva analytical query workload, also retrieves customer data within a time range, where the range can be as low as one hour and as high 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 8 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 network bound. The baseline encrypted client is able to achieve a bit of compression over a single row, so it is slightly faster. As the size of the database increases, disk becomes more of a bottleneck, and vanilla and encrypted baseline reach the 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, write has two overheads: 1) MiniCrypt needs to do an extra read for every put. 2) MiniCrypt uses a lightweight transaction (update-if) for every put.

Figure 9 shows the result of running 100% uniform random writes on a pre-loaded 10 GB database, and each experiment runs for 120 seconds. Baseline client is extremely fast because it is able to perform blind writes. MiniCrypt GENERIC mode with update-if is slow and dominated by the extra read. Reads to disk are slower than writes to disk because writes simply append to a log. We also see that the usage of the lightweight transaction puts further stress on the GENERIC mode write. Our append mode writes increase the performance of put by several orders of magnitude, as we now discuss.

Append mode. We run two sets of experiments in MiniCrypt APPEND mode: 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 with 120 seconds (except for the long 100% write).

Write-only. We start with an empty database for both the encrypted baseline and MiniCrypt. Figure 10 compares baseline client with MiniCrypt in APPEND mode. Compared to Figure 9, MiniCrypt is able to keep up with 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 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 11 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 10. 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 the key inserts.

**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 are executed on a pre-loaded 70 GB database. We adjust an “interval” parameter that indicates the range of the read keys. 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 baseline client and MiniCrypt are warmed up for 5 minutes before each run. Each experiment runs for 120 seconds.

Figure 12 shows MiniCrypt’s performance under 50% reads and 50% writes. Reads are uniformly distributed over an interval, the size of which is indicated on the x-axis. When both baseline client and MiniCrypt fit in memory, baseline client performs better. When baseline client can no longer fit data into memory, the performance dips off. Since writes are faster than reads, the baseline’s performance settles to a point that is higher than the baseline performance in Figure 8 (a). The performance of MiniCrypt **APPEND** falls off as the size of the query interval increases. This happens because the merge process also needs to read recently inserted values. If more values are read into memory as part of the normal benchmark, the two processes will interfere with each other.

### 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: whether the data fits in memory, the throughput of the server, the size of each row, the workload (how many put/s it performs), how compressible the data is, disk or SSD bandwidth, network bandwidth, optimizations specific to the underlying data store, etc.

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 as depending on the pack size. MiniCrypt then chooses the pack size that provides the highest throughput. Figure 13 shows running YCSB 100% uniform read workload for 50 GB of Conviva data, plotting maximum throughput against different pack sizes. 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, \[ \text{argmin}_n \{\text{compratio}(n) \cdot \text{data size} < \text{memory size}\} \]

where \( \text{compratio}(n) \) is the compression ratio obtained when compressing a pack of size \( n \).

We recommend using MiniCrypt for cases when all or most of the data fits in memory when compressed by MiniCrypt, but would not fit in memory without MiniCrypt. We showed above 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 Other metrics

**Latency.** MiniCrypt adds extra latency overhead on the client side due to compression/decompression and encryption/decryption. This is a fixed overhead, independent of server load. Handling a pack in MiniCrypt as opposed to a single row in the baseline client adds 0.4ms.

**Network bandwidth.** MiniCrypt’s network bandwidth overhead can be determined by \( \# \text{ of rows in each pack} / \text{pack compression ratio} \). In our experiments, network bandwidth did not become a bottleneck. We expect MiniCrypt to be used in settings where network is not the bottleneck.

### 9 Related work

In this section, we discuss work related to MiniCrypt. To the best of our knowledge, MiniCrypt is the first key-value store that supports both data encryption (where the decryption key is not available to the server) and compression.

**Key-Value Stores.** Some key-value stores (e.g., Cassandra [23] 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 or the 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. Nevertheless, Succinct does not support encryption.

**Encrypted databases.** A number of recent proposals in databases support queries on encrypted data [30, 7]. However, encryption introduces a significant storage overhead.
Compressed databases. Compression is a common technique that exploits data compressibility trends and a new technique, key packing, that exploits this observation. 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 an order of magnitude by fitting more data in main memory.

File systems. There has been a lot of work on designing encrypted file systems [9, 19, 21, 26] 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

In this paper, we presented MiniCrypt, the first big data key-value store that reconciles encryption and compression. At the core of MiniCrypt is an empirical observation about data compressibility trends and a new technique, key packing, that exploits this observation. 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 an order of magnitude by fitting more data in main memory.

11 References