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 [26], 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, 16, 3, 11, 13, 21, 7]. Unfortunately, leakage of confidential data from these application servers remains a significant problem [12].

Hence, such applications need systems that are both efficient at handling “big data” and able to preserve the data’s confidentiality. Regarding efficiency, many big data stores employ compression [20, 5], as it decreases costs (e.g., over $1 million for a Hadoop-based system [29]) and can significantly increase performance (e.g., by up to an order of magnitude [7, 29]). 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. Regarding data confidentiality, a natural solution is to encrypt the data stored on servers, and keep the decryption key on the client’s side. Even if an attacker (e.g., hacker, server administrator) gets access to all the data on these servers, the attacker cannot decrypt the data, because the server never gets the decryption key. Hence, an ideal system would incorporate both encryption and compression.

However, encryption and compression have been traditionally at odds. First, encryption is not compressible due to its pseudorandom properties and security guarantees. Second, while compressing and then encrypting works well in some systems, it is problematic for a key-value store. Compressing a single key-value pair typically provides limited benefits, while compressing multiple key-value pairs together removes the server’s ability to access an individual key and ensure consistency because the server cannot decrypt. As a result, existing systems choose either compression or encryption, but not both. For example, Microsoft tells its users that encryption and compression are mutually exclusive in their NTFS file system, and users must choose at most one of these [31].

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 the more data one compresses together, the better the compression ratio one gets. However, encrypting large compressed chunks is problematic for key-value stores because processing and sending these amounts over the network for every key request can hurt performance significantly. Our empirical observation is that by compressing together only relatively few key-value rows, one can achieve a high compression ratio, close to the one of compressing the entire dataset. As we discuss in §3, we observed this behav-
ior 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 [1], compressing 1 row (or key-value pair) yields a compression ratio of 1.6, compressing 25 rows yields a compression ratio of 4.1, 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 rows already provides 80% of the maximum compression ratio.

Based on this observation, MiniCrypt packs together few key-value pairs, compresses, and encrypts them. Clients now retrieve and update packs instead of individual key-value pairs. However, this simple design runs into a significant challenge: the encryption of packed rows removes the server’s ability to manage key-value pairs and maintain consistency because the server cannot decrypt. In a system without encryption, the server was able to decompress the data to read or update a key-value pair. Maintaining these properties turns out to be a challenging distributed system problem, which we address through a set of new techniques. To increase the generality and practicality of our solution we present these challenges and techniques in the context of existing key-value stores. In particular, in our solution we leverage only basic primitives likely to exist in many key-value stores, without requiring change to the internals of these systems.

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. 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, primitive which is provided by virtually any existing key-value store. Furthermore, since many big data workloads are mostly append (e.g., logs indexed by time), we provide a tailored protocol for append based on a fast pack management protocol, which achieves a write throughput close to the throughput of a regular key-value store.

The second challenge MiniCrypt addresses is that the server cannot manage key packs since the data is encrypted. As rows get inserted or removed, some packs could become too small or large, impacting either compression ratio or performance. Hence, these packs need to be split or merged. As a result, this task is left to clients, who can fall or act concurrently. Merge and split are operations over multiple rows so they would need multi-key transactions. To provide high performance, key-value stores provide poor transactional semantics (e.g., most stores do not support transactions over multiple keys), making it hard to resolve these issues using concurrency control. Instead, we provide two new algorithms to merge and split packs with the key property of being idempotent. These algorithms achieve guarantees for multiple rows while using a lightweight form of transactions for a single row.

We have implemented MiniCrypt on top of Cassandra [21] without any changes to Cassandra. Our performance evaluation of MiniCrypt shows that MiniCrypt delivers significant compression, while keeping the data encrypted: for example, on the Conviva dataset, MiniCrypt compressed 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. As a consequence, we show that MiniCrypt can increase the server’s throughput significantly as compared to regular Cassandra by fitting more data in main memory: for example, on the Conviva dataset, the throughput increases by up to 100 times for a disk-backed machine and by up to 9.2 times for an SSD-backed machine. Finally, it turns out that MiniCrypt’s packing delivers particularly good performance for range queries (which are common for time-series data), 5 to 40 times faster than regular Cassandra on SSD.

2 Overview
In this section, we present an overview of MiniCrypt and its architecture.

2.1 Threat model
There are two types of parties: a server (which can be distributed) and clients. The server hosts the key-value store encrypted. The clients share the decryption key.

The threat model is that an attacker has compromised the server and has access to all the data at the server (including any messages/queries from clients) and may leak the data. Nevertheless, we assume that the attacker is passive: it does not corrupt the data or answers queries incorrectly.

We assume that the clients are allowed to see the data, and they are trusted not to disclose it to the server. In this paper, we are concerned with protecting data confidentiality from the server.

MiniCrypt assumes there is a shared symmetric key on the clients’ side that is not available to the server. There are various options to setup such a key on the clients’ side, and MiniCrypt can work with them. For example, in a typical corporate deployment, a user machine registered within the corporation’s perimeter that is permitted access to this data store, gets a copy of this key. Alterna-
tively, the key can reside in client-side proxies while authorized user requests pass through these proxies. Some applications implement a finer access control to the store, permitting certain users access to only certain records. In MiniCrypt, this can be implemented at the level of the client-side proxies.

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. 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 MiniCrypt easier to adopt into various systems and enables it to inherit the performance, fault tolerance, and consistency guarantees of existing stores. Finally, MiniCrypt should preserve the consistency semantics of the underlying key-value store.

2.3 System API

MiniCrypt exposes the basic key-value store API together with support for range queries.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get (key)</td>
<td>returns the value associated to key value</td>
</tr>
<tr>
<td>put (key, value)</td>
<td>sets the value corresponding to key value</td>
</tr>
<tr>
<td>delete (key)</td>
<td>deletes the key key</td>
</tr>
<tr>
<td>get (low, high)</td>
<td>returns all the (key, value) pairs where low ≤ key ≤ high</td>
</tr>
</tbody>
</table>

Range queries get (low, high) are common for time-series big data [23, 10]. 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.

2.4 Design overview

Fig. 1 summarizes MiniCrypt’s architecture in comparison to a regular key-value store that provides confidentiality. 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 the sorted sequence of all keys in the dataset. Each pack has a packID, stored encrypted at the server. We choose the packID to be the smallest key in the pack, and assume there is an 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 the key. Each pack also has an associated hash over its ID, value, and any status message we use in later sections.

In §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. One can reduce the information leaked by the size of the pack by padding the encrypted packs to a tier of few possible sizes, which also retains some compression.

If the original system encrypts keys, MiniCrypt also encrypts packIDs to maintain the security in the original system – the reason is that the packID equals the lowest
key in the pack. Recall that a key-value store performs some simple computation on the keys (fetches keys by equality or by range) as in the API above. The state-of-the-art literature provides different options for key encryption based on the desired tradeoff in functionality versus security. We explain in Appendix F that whatever encryption the original system uses for keys, this encryption applies seamlessly to packIDs as well. MiniCrypt does not aim to innovate here, but merely use the state-of-the-art. Hence, in this paper, for brevity, we present MiniCrypt’s design using unencrypted packIDs. Nevertheless, we explain in Appendix F that encrypting packIDs requires either no change or a minimal change to MiniCrypt’s design.

2.4.1 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 support these (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 an 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 §4.1 and §4.2. All key-value stores support an index on the key and many of these enable order clustering or range queries on this key.

The second property, the update-if, 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 procedure. Most key-value stores support such transactions. For example, in Cassandra, these are called lightweight transactions [17].

We show that MiniCrypt maintains the semantics of the underlying store for eventual consistency and linearizability, two common consistency models. We expect MiniCrypt to maintain other kinds of consistency semantics too. Moreover, MiniCrypt inherits the fault tolerance and replication mechanisms of the underlying store.

2.4.2 Modes of operation

MiniCrypt provides two modes of operation: 

<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 (get, delete)</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 operation that maintains the packs.

if primitive. When a pack becomes too large, it is split. Since deletes are infrequent in many big data applications and 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 [23, 10], where the keys are timestamps, while values are typically actions or measurements. For example, Spotify uses Cassandra [25] in this mode: instead of storing a playlist as an individual record, Spotify stores pairs (timestamp, song added), and reconstructs the playlist from this history. 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.

Table 1 compares the two modes.

3 Key packing

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

The 6 datasets we examined are: anonymized Con- viva 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
Figure 2: Compression ratios for different datasets. Note that the x-axis is plotted on log scale, because the compression ratio is increasing very fast. The table summarizes the trend for each dataset on the zlib algorithm: it lists the compression ratio achieved when it is $\geq 75\%$ of the maximum one, as well as the average pack size (in number of rows/values) used to achieve that compression ratio and the total number of rows in the dataset for comparison.

Average value size: Conviva: 1.1 KB; Genomics: 1.3 KB; Twitter: 5.5 KB; Gas sensor: 135 B; GitHub: 11.6 KB; Wikipedia: 13.2 KB

4.1 Get

In MiniCrypt, since key-value pairs are packed and encrypted, 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 in presence of client failures.

Instead, MiniCrypt enables clients to fetch the correct pack even 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 $k$. Fig. 3 presents the overall procedure.

4 Read operations

In this section, we describe read operations in MiniCrypt, which works the same in both the generic and append mode.
records its hash to update a key in a pack. The client reads the pack and from each other as follows. Consider that client C1 wants hashes to ensure that clients do not overwrite changes to the same pack, thus violating consistency semantics. If designed naively, a concurrent client may overwrite updates from a different client to another key within the pack. The client can proceed with the original operation once a split successfully completes.

How to split a pack. Fig. 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.
1: procedure split(packID, pack, h)
2:     Assemble the right half of the pack:
3:         right_pack with 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.

5.4 Correctness
Recall that our notion of correctness was that MiniCrypt maintains the consistency semantics and the liveness of the underlying key-value store. We now present intuition for why this algorithm preserves these properties, and delegate an argument to Appendix D.1.

First, any put or delete to a pack that has a number of keys greater than max_keys will wait. The only operations allowed on this pack are splits. This allows concurrent modifications to co-exist with split without . The “insert if not exists” and “update-if” ensure that a client which started a split and was delayed does not overwrite changes made after the split finished (due to other clients) to either of the two packs. Regarding liveness, a client will not be stuck in an infinite loop because at least one 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 Appendix C, 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 [23, 10], where the keys are timestamps while values are some action or event. For example, Spotify uses Cassandra [25]: it does not store playlists but instead it stores pairs ⟨timestamp, song added⟩, and reconstructs the playlist from this history.

Let us define concretely 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: for any key $k$ with timestamp $t_k$ (assigned by a server replica when first inserted), if $t_s$ is the current timestamp at any of the replicas and $t_k < t_s - t_\Delta$, 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 an APPEND mode, MiniCrypt is able to execute 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. Each epoch is an epoch long, where $\text{epoch} > T_\Delta + T_{drift}$. We define $T_{drift}$ 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. The resulting packs are placed in a special epoch 0. The keys in the current epoch are eventually deleted. Even if two clients merge the same epoch, the results are the same. Fig. 7 shows the timeline of merge operations.

Epochs are useful not only for assigning keys to clients, but also to enable 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.

Second, MiniCrypt avoids having multiple clients merge the same epochs. 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. Even if only two clients merge the same epoch on average, the merge throughput is decreased by half. For this, MiniCrypt introduced a special service called the *epoch management* service or 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 instance updates `g_epoch` once every 8 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. Moreover, the EM instances can run on clients instead of the server. Due to higher churn on the server’s API. Moreover, the EM instances can run 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 essentially relying on the underlying store’s 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, 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,e}` but less than `k_{min,e+1}` are grouped together and merged. 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.

6.2 Correctness

We provide an intuition for why our append protocol preserves the semantics and liveness of the underlying key-value store, and delegate an analysis to Appendix §D.2. The writes are regular single-key inserts, thus inheriting the correctness of the underlying system. The merge process preserves correctness for two main reasons. First, there are no concurrent put and delete operations because clients only merge epochs that are two epochs older than $g_{epoch}$. Second, 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 [21]. 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 or the order-preserving encryption scheme from [28].

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 $\langle key, value \rangle$, 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 $\langle part_key, key \rangle$ where $part_key = SHA56(key) \mod N$. 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 §4.1. In addition to a generic interface, MiniCrypt also has the ability to support a compound primary key $\langle partition_key, clustering_key \rangle$. The primary key used in this situation is key pair $\langle part_key, key \rangle$ where $part_key = SHA56(partition_key) \mod N$.

Cassandra supports an update-if primitive that is used in generic MiniCrypt put operations [17]. 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 providing both good compression and good compression/decompression speed. Unless indicated otherwise, all MiniCrypt experiments (in both APPEND and GENERIC modes) set key 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.

8.1 Read performance

We ran a modified YCSB read workload on a small three-instance cluster. We pre-load data of different sizes into the database in separate tables, and run a 100% uniform read/range query workload. We compare MiniCrypt with a baseline encrypted client that compresses and encrypts each row separately. 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.

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.

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 baseline client.

8.1.2 Range queries

Range queries are very common in many workloads, such as with time series data. 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, for example, 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 $(k - 1000, k]$. The pack size is still set to be 50. Figure 8 shows that MiniCrypt experiences a significantly higher maximum range query throughput compared to the baseline client consistently, both when the data fits in memory and when it does not. MiniCrypt is able to achieve up to 5x performance gain over the baseline client. These experimental observations align with our analysis. Since our range is 1000 records, MiniCrypt is able to achieve much better performance even when both clients are able to fit data in memory. 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.

Fig. 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).

Fig. 10 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 Fig. 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.

Fig. 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 Fig. 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.

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 determine empirically 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. Fig. 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, \( \arg\min_n \{ \text{compratio}(n) \cdot \text{data size} < \text{memory size} \} \), where 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 (# 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 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 [21] 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 [7] supports compression for a key-value store, while enabling rich search capabilities. Nevertheless, Succinct does not support encryption.

**Databases.** A number of recent proposals in databases support queries on encrypted data [28], or on compressed data [30, 6, 14, 22, 7], but not both. Moreover, the former class often introduces a significant storage overhead compared to the unencrypted data (e.g., 5 times larger for [28]) and does not support compression while executing queries on encrypted data. MiniCrypt differs from the above systems in two main ways. First, it focuses on NoSQL stores and does not support the more generic SQL operations. Second, for this functionality, MiniCrypt achieves both data confidentiality through encryption and a significant compression ratio.

**File systems.** There has been a lot of work on designing encrypted file systems [8, 18, 19, 24] 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.
References


college courses.


A Pack size analysis
Fig. 14 shows the dependency of compression ratio on pack size for the other datasets we examined.

B Deleting packs
Deleting packs interferes with split because some 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 to a pack 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 number (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 - $T_{delay}$.

C Append-mode updates
Updates, such as put (insert, update) or delete can happen in append-mode too. These operations must happen only on keys that are already merged, namely, are in epoch $E$. The protocol for these operations is exactly like in 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 $E$ away from the current timestamp, the operation must wait until the corresponding epoch gets merged and the packs are moved to epoch $E$. 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.

D Correctness argument
To analyze MiniCrypt’s correctness, we must argue for the following two properties

- MiniCrypt preserves the consistency guarantees of the underlying key-value store for some common types of consistency semantics, and
- MiniCrypt preserves liveness.

We show the first property with respect to two common consistency levels: eventual consistency and linearizability. We’ll use the following semantic assumptions:

1. Under linearizability, a single record’s operations must be ordered so as to respect the real time order of operations. This means that if a get operation happens after a modification (a put or a delete), the get operation must reflect the latest state.

2. Under eventual consistency, we assume that if there is no update to a key, eventually all get operations will return the same value.

To ensure consistency, we rely on the update-if primitive (which can take the form of insert-if or delete-if). An update-if [17] ensures that any get operation happening after an update-if to the same row completed, returns the value of the latest update-if if no put or delete happened in the meantime.
Regarding liveness, we want to argue that MiniCrypt will allow clients to proceed in the presence of both server and client failures. In particular, we assume that a majority of machines on the server side are live, and that an arbitrary number of clients can fail. Of course, if all clients have failed, then the system is not expected to make any progress.

D.1 GENERIC mode

In this section, we provide an analysis for why GENERIC mode satisfies consistency and liveness properties.

Consistency. To show that MiniCrypt maintains consistency, we need to examine the values returned by get operations. We first consider the case when get operations are concurrent with modifications, but not with split s. MiniCrypt get operations use range queries to find the correct keys. Under eventual consistency, a range query should be able to find the correct pack eventually, after the latest update has propagated to all replicas. Under linearizable consistency, we assume that the underlying storage system provides range queries that follow the consistency requirements – that if a key exists in the database, any range query under linearizable consistency will return the latest value of a key. All modifications (put, delete) need to perform a get to retrieve the pack before applying a modification. These operations are serialized with update-if and a hash check. Hence, two put/delete operations to the same pack (even if for different keys) will not overwrite each other. Since the get operation is always able to return the correct pack, all concurrent modifications will also be able to apply their updates on the correct packs under different consistency levels.

We need to argue that the same remains true when considering split operations. It suffices to argue that split does not change the state of the database as perceived by get. In other words, after a split, all keys that are present before should still be present with the same values and accessible. Concurrent successful modifications should be preserved as well: a value inserted/updated in a successful put operation will not be lost, and a value that is deleted will not re-appear.

split has several operations. A client first retrieves the original pack $p$. It then splits the pack, inserts $p_{right}$ into the database and then modifies $p$ such that $p \rightarrow p_{left}$.

This operation does not lose keys. This means that all keys present before split happens will still be present after split is done. Once the right pack is inserted, any get to a key in the right pack will fetch values in that pack (even though the same keys will exist in the left pack for a while). This also means that updates to the right pack will update the right pack and be visible to get.

A split operation always writes back keys using
update-if. If a split insert fails (the right pack insert fails), the keys in the right pack will not be lost because the left pack will not be modified and will contain the original set of keys. Once the right pack has been inserted, the left pack can only be updated by a client who is splitting the pack. Any client who wants to modify the pack will have to participate in split first.

Next, we argue that a key inserted/modified during a successful concurrent put operation will appear in the correct pack. This is easy to see for two reasons:

- put operations are serialized with update-if and a hash check. Hence, two put operations to the same pack (even if for different keys) will not overwrite each other.
- No put is allowed to proceed on a pack that needs to be split. The client wishing to perform a put becomes a splitter and runs split. Then, the put will proceed as before.

The same argument holds for delete operations. The reason delete behaves the same as put is that we do not delete empty packs so deleting a key is essentially just modifying a pack. Not deleting empty packs is crucial for the safety of split because this operation relies on the hash of the packs to maintain safety. Hence, an empty pack cannot be deleted at the same time with split. As explained in §B, our delete protocol ensures this is the case.

Liveness. MiniCrypt provides liveness for the following reasons. First, the liveness of get follows immediately from the liveliness of the underlying system because a get in MiniCrypt consists of a SELECT query at the server.

Second, even though put executes a loop that completes only when an update-if succeeds, note that at least one client will succeed in making its update. Due to the liveness guarantees of the update-if in the underlying system, it follows that put in MiniCrypt maintains liveness. The delete operation is similar to put.

The split of a pack will finalize if there is one client who wants to update this pack that does not fail. Even if a client fails during the split procedure, other clients will finalize this split. The reason is that any client wishing to update this pack will become a splitter whenever it encounters the original pack (whose pack size exceeds a threshold).

D.2 APPEND mode

We now argue that our APPEND mode protocol preserves consistency and liveness of the underlying system.

Consistency. In APPEND mode, MiniCrypt’s reads are very similar to GENERIC mode reads. Each get operation first attempts to read from the merged packs, and if no key is found, the get operation will try to read from the list of unmerged keys. As analyzed in §D.1, a get on the merged packs will be well-behaved under both linearizable and eventual consistency. A get on unmerged keys will also adhere to the correct semantics because each put is a single-key insert. Thus, APPEND mode adheres to consistency requirements for put, get, and delete.

APPEND mode has another operation: merge. We now argue that merge does not change the state of the database as perceived by get.

There are two conditions that the merge process must satisfy:

- all inserted keys must be merged into some pack $p$ with a pack ID $p_{id}$
- every key $k$ must be merged into the correct pack such that it can be queried using the pack querying algorithm get in §4.1

Let us denote the minimum key from epochs $e−1, e, e+1$ to be $k_{min,e−1}, k_{min,e}, k_{min,e+1}$ respectively. As described in the APPEND design section, merging an epoch $e$ requires reading back epochs $e−1, e, e+1$ and merging all keys between $k_{min,e}$ and $k_{min,e+1}$. It suffices to provide the following guarantees:

- The set of intervals $[k_{min,1}, k_{min,2})...(k_{min,e}, k_{min,e+1})]$ covers all keys
- For all epochs $e_i, e_j$, if $e_i < e_j$, then $k_{min,e_i} < k_{min,e_j}$

The first condition can be satisfied if the second condition is satisfied. To argue for the second condition, we must first take into account how keys are arranged in epochs. Specifically, we want to show that

- Any two keys inserted within $T_\Delta$ (and hence could potentially be out of order) can be at most 1 epoch apart

This condition holds true because $EPOCH$ is greater than $T_\Delta + T_{drift}$, where $T_\Delta$ takes into account many delays, including the delay where keys can be inserted out of order. Let $k$’s epoch be $e$. If a key $k'$ is inserted before $k$, and within $T_\Delta$ time away from $k$’s insertion time, then a client that is in charge of $k'$ must have read an epoch value of at least $e−1$. This is true because the client can delay reading the newest epoch by at most $T_{drift}$.

With the above condition, we can ensure that $k_{min,e−1} < k_{min,e} < k_{min,e+1}$ for any epoch $e$. Since epoch $e$ lasts for $EPOCH$ seconds where $EPOCH = T_\Delta + T_{drift}$, there exists a key $k_{early,e}$ such that its insertion time is more than $T_\Delta$ away from $k_{min,e+1}$’s insertion time. The key $k_{min,e}$ is chosen such that it is the minimum key inserted in epoch $e$. By the transitive property, $k_{min,e} < k_{early,e} < k_{min,e+1}$. Therefore, $k_{min,e−1} < k_{min,e} < k_{min,e+1}$. 

16
Recall that, when MiniCrypt merges epoch $e$, we include all keys from epoch $e-1$ that are greater than $k_{min,e}$, as well as discarding all keys from epoch $e$ that are greater than $k_{min,e+1}$. The merge process would be correct if and only if the following occurs:

- a key that is between $k_{min,e}$ and $k_{min,e+1}$ is never inserted outside of epochs $e-1, e, e+1$

We know that all keys inserted in epoch less than $e-1$ must be strictly smaller than $k_{min,e}$, because they are all more 1 epoch away. All keys in epochs $e+2$ and beyond must be greater than $k_{early,e+1}$, and hence greater than $k_{min,e+1}$. Therefore, all keys between $k_{min,e}$ and $k_{min,e+1}$ will always be correctly merged. This can be extended to all keys because the condition is true for every epoch.

Given that merge does not change the logical state of the database, we need also need to make sure that merge in the presence of concurrent modifications will still keep the same semantics. As mentioned before, get operation behaves correctly when reading either from packs or unmerged keys. Under eventual consistency, a get operation will be guaranteed to return the value once a merged pack has been inserted and propagated to all replicas. Under linearizability, a merged pack is inserted with a linearizable operation (e.g., “INSERT IF NOT EXISTS”), which guarantees that all get-s executed after the insert will be able to read the correct pack and find the corresponding key.

**Liveness.** APPEND mode merge achieves liveness because we assume that the majority of server replicas will be live. Thus, epoch advancement is always achieved because there will always be a master EM instance. If a merge client fails, the master EM instance will be able to detect that the client went down and assign another client to merge that epoch.

**E  Compression intuition**

Providing a mathematical proof for our observation is hard because it depends on the distribution of the data and the details of the compression algorithm. Nevertheless we can still build some theoretical intuition as follows. Real data comes from skewed distributions so let’s consider a simplified model of such a distribution: a few strings are very popular, with a chance of being sampled of high, and most of the strings are less popular with a probability of low. Assume that the probability masses for high and low strings are equal. A row in our dataset consists of one string. Now consider an idealized compression algorithm that can replace each high string with 1 character and each low string with an $s$-byte character for $s > 1$. The expected compression ratio is

$$\frac{s}{1 + 2^{-s/2}} = \frac{2e}{1 + e^2}.$$ 

Now imagine sampling a large amount of data from this distribution: $N$ total rows. This models the data in the key-value store. Then, as we pack rows, and the pack size increases, by the law of large numbers, we approach the average compression ratio within each pack. But at the same time, note that the speed of convergence within each pack depends only on the size of the pack, and not on $N$. So, in some sense, the point $n$ where a pack size of $n$ yields a compression ratio within $10\%$ of the overall compression ratio is independent of $N$. As $N$ grows very large (for large datasets), this point remains the same. Hence, intuitively, this point $n$ is relatively small to $N$, although this does not explain why $n$ is small in absolute value.

**F  PackID encryption**

The goal of MiniCrypt is to take a key-value store that protects the confidentiality of values in key-value pairs by encrypting them, and enhance this original system with compression, while maintaining its level of security. The paper already discusses that we encrypted packs with a strong and standard encryption scheme (e.g., based on AES), which protects both the values and the keys in the original key-value store.

If the original system encrypts keys, MiniCrypt also encrypts packIDs while providing the same security. MiniCrypt encrypts packID-s because a packID is equal to the lowest key in the pack. However, since the key-value store system computes on keys (fetches keys by equality or by range), there are different options for key encryption based on the desired tradeoff in functionality versus security. If the original system uses a strong encryption scheme, the original system does not support range queries; in this case, MiniCrypt can use the same encryption for packIDs to provide the same level of security, and it also cannot support range queries. If the original system uses order-preserving encryption [9, 27] in order to support range queries (these are the fastest known encryption schemes allowing range queries), MiniCrypt can also use this encryption scheme for packIDs and support range queries.

We now discuss how MiniCrypt can incorporate these encryptions in the design. As we will see, the changes needed are minimal.

**F.1  Option 1: strong encryption**

Let us explain now the case when packIDs are encrypted with strong encryption schemes. In this case, the only property needed from the encrypted pack IDs is that the encryption is deterministic so that different clients encrypting the pack ID 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 setting, 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.

In this case, MiniCrypt creates the packs based on the order of the encrypted keys, as opposed to the order of plaintext keys. Concretely, each key is encrypted and the packs are formed in order of the encrypted keys. The packID is the smallest encryption value, which might 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 Fig. 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.

F.2 Option 2: order-preserving encryption

The case when the packIDs are encrypted with order-preserving encryption 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. Such an encryption scheme hides the value of pack ID, but enables the server to learn the order relations between encrypted pack IDs so it can run range queries efficiently.

F.3 Other encryption strategies

If the original system uses a different technique than the two options discussed above, it is likely that this technique applies to packIDs too while providing the same level of security: the reason is that a range query on keys translates to a range query on packIDs in MiniCrypt.

F.4 Performance

In Option 1, one can use Blowfish which produces 64-bit ciphertexts. In Option 2, using the scheme of [9], the ciphertexts are also 64 bit long. We ran 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.