(**Disclaimer**: This post is intentionally kept terse and doesn't discuss the detailed design. The intention is to keep the community aware of upcoming feature additions to the FDB.)

At Snowflake we are currently working on the **FDB native Encryption data at-rest** feature. The below document highlights the feature details with the broader community incase it is useful for others and they want to start preparatory work to utilize the feature (or want to participate in early testing as soon as the feature is code-complete in the main branch)

Status Quo

FoundationDB being a multi-model, easily scalable and fault-tolerant, with an ability to provide great performance even with commodity hardware, plays a critical role enabling enterprises to deploy, manage and run mission critical applications. **Data-at-rest encryption** support is considered a table-stake feature for any modern enterprise service offering.

FDB currently **does not support** data-at-rest encryption semantics; a customer needs to implement application level encryption schemes to protect their data/metadata stored in FDB. Given the criticality of the feature, it seems highly desirable to implement FDB native data-at-rest encryption support adhering to the latest security standards.

Goals

- **Implement end-user transparent**, FDB native data-at-rest encryption support. Encrypting data in-transit and/or in-memory caches at various layers in the FDB query execution pipeline is out of scope.
- **Isolation guarantees**: the encryption domain matches the "**tenant**" partition semantics supported by a FDB cluster. For details on "**tenant**" concepts refer to documentation
- **KMS integration**: Ease of integration with external Key Management Service/Solutions. The proposed approach relies on external Key Management Solution (KMS) to create encryption keys and manage its lifecycle. In other words, FDB will **not** implement native KMS as part of this project.

Design Details

Encryption mode

- FDB will leverage **AES-256 CTR** block cipher mode to encrypt data and metadata stored in persistent storages.

- **Key Derivative Function (KDK)**: Encryption keys procured by external KMS are not used directly to encrypt the data, however, a KDF function leveraging **HMAC-SHA-256** algorithms is used to derive the encryption key used to perform the data encryption.
- **Authentication Tokens**: The scheme implements **crypto-secure** (HMAC-SHA-256) authentication token generation scheme to protect plaintext encryption header (stored along with encrypted payload to assist decryption on reads) against **tampering** and/or **bit rot/flip** attacks.

Write Request Workflow

- A FDB client initiates a write transaction, the respective FDB endpoint will receive `plaintext` {Key, Value} tuple to be stored in FDB.
- A FDB cluster host as part of write transaction processing will do the following:
  1. `CommitProxy` server will procure tenant specific encryption key from an in-memory encryption-key cache OR external KMS (if not already cached).
  2. Encrypt mutation i.e. {Key, Value} tuple before persisting mutation in TLog persistent disks. The client write transaction will be acknowledged after the TLog mutation is `deemed` durable.
  3. As a background task, `StorageServer` will pull encrypted mutations from TLog servers, decrypt mutation and populate its in-memory data structures, however, on data flush, StorageServer will encrypt the data before storing them on persistent disks. **Note**: The proposed scheme ONLY supports **Redwood Storage Server** encryption for now.
- The FDB process encrypting the buffer will generate an `encryption-header` preserving sufficient information to assist data decryption on reads; the header is persisted in `plaintext` along with the encrypted buffer.

Note: Given encryption is performed at the time of data flush, it doesn't impact the order of {Key, Value} tuples stored on the disk; in other words, the sequence of {Key, Value} stored on disk `with` or `without` the encryption would remain the same.

Read Request Workflow

- A FDB client initiates a read transaction.
- A FDB cluster host as part of processing reads transaction does the following:
  1. `StorageServer` will read desired data blocks i.e. {encryption-header, encrypted-buffer} tuple from the persistent storage.

2. Parse the `encryption-header`, validate `authorization-token` integrity, and regenerate the required encryption key to decrypt the data.
3. Decrypt the data and pass the `unencrypted` result to the caller.

FDB KMS integration

The design implements a generic framework allowing the community to implement a desired FDB native KMS connector. Further, this project implements a **RESTKmsConnector** module enabling FDB to connect with external KMS using REST protocol.

RESTKmsConnector <-> KMS integration

- **KMS Discovery**: The TLog mutations are encrypted, FDB on startup needs to discover and connect to KMS to enable ClusterRecovery. The scheme supports localhost file based KMS Discovery protocol (input obtained via command-line arguments).
- **FDB host authorization**: Given FDB <-> KMS communication deals with transferring highly sensitive `plaintext` encryption-keys on-wire, the scheme allows embedding **validation-tokens** in the REST `request_json_payload`. The tokens need to be stored as localhost files for now (input obtained via command-line arguments).
- **Encryption Key fetch**: KMS needs to implement the REST endpoint allowing FDB to fetch encryption keys on demand. The module defines acceptable JSON `request` and `response` schemas.

Timeline

Our target is to release this feature as part of **7.2 release**, which is currently being tracked to be released in **Fall 2022**. The code is actively developed and tested, we will write detailed documentation to assist the community to test this feature on a prerelease version.