Prysm Database

A Comprehensive Analysis of the Persistence Architecture of the Prysm P2P Encrypted Messenger

xmreur

Version 2.0 — July 2026

Abstract

Prysm employs a multi-tier persistence architecture combining three SQLite databases for structured data, an on-disk blob store for oversized payloads, and OS-level secure storage for cryptographic key material. This document specifies all schema, data flows, migration history, and design rationales for the persistence layer. Topics include the relational model for users, groups, messages, and sessions; the outbox pattern for reliable delivery; the blob offloading mechanism for efficient storage; the encrypted key store; and the panic/wipe system for data destruction.

Keywords: SQLite, persistence, database, key-value store, secure storage, blob, outbox, migration

Contents

1 Introduction

Prysm's persistence layer is organized into four tiers, each serving a distinct purpose:

Tier	Technology	Purpose
Structured data	SQLite (`sqflite`)	Users, groups, messages, sessions
Blob store	Filesystem	Payloads > 512 KB
Key material	`flutter_secure_storage`	Identity keys, prekeys
User settings	`SharedPreferences`	Theme, notifications, preferences

This layered approach ensures that cryptographic secrets are stored with the maximum available protection, while application data benefits from the query capabilities of a relational database.

2 Database Architecture

Prysm uses three separate SQLite database files for isolation and performance:

File	Location	Contents
`chat_app.db`	App documents directory	Users, groups, sessions, prefs
`messages.db`	App documents directory	Messages, reactions, receipts
`pending_messages.db`	App documents directory	Outbound pending queue

All databases use WAL (Write-Ahead Logging) mode for concurrent read/write performance and are accessed through the `sqflite` / `sqflite_ffi` Dart packages.

3 Chat Database (`chat_app.db`)

The chat database stores identity, group, session, and preference data across eleven tables with a migration history spanning nine versions.

3.1 Schema Definition

3.1.1 Users Table

```
CREATE TABLE users (
  id TEXT PRIMARY KEY,
  name TEXT,
  avatarUrl TEXT,
  avatarBase64 TEXT,
  customName TEXT,
  publicKeyPem TEXT,
  identityJson TEXT
);
CREATE INDEX idx_users_name ON users(name);
```

The `id` field is the peer's `.onion` address. `identityJson` stores the v2 identity public keys as a JSON blob. `publicKeyPem` is a legacy field retained for backwards compatibility.

3.1.2 Groups Table

```
CREATE TABLE groups (
  id TEXT PRIMARY KEY,
  name TEXT NOT NULL,
  avatarBase64 TEXT,
  createdBy TEXT NOT NULL,
  createdAt INTEGER NOT NULL
);
```

3.1.3 Group Members Table

```
CREATE TABLE group_members (
  groupId TEXT NOT NULL,
  memberId TEXT NOT NULL,
  role TEXT NOT NULL,
  joinedAt INTEGER NOT NULL,
  PRIMARY KEY (groupId, memberId),
  FOREIGN KEY (groupId) REFERENCES groups(id) ON DELETE CASCADE
);
```

Roles are `admin` or `member`. The foreign key cascades deletion.

3.1.4 Group Keys Table

```
CREATE TABLE group_keys (
  groupId TEXT PRIMARY KEY,
  encryptedKey TEXT NOT NULL,
  keyVersion INTEGER NOT NULL DEFAULT 1,
  FOREIGN KEY (groupId) REFERENCES groups(id) ON DELETE CASCADE
);
```

Group keys are stored encrypted using the user's own X25519 key (self-wrapped via `dh-aead-1`). Each key has a version number for epoch transitions.

3.1.5 Session State Table (Ratchet)

```
CREATE TABLE session_state (
  peerId TEXT PRIMARY KEY,
  ratchetJson TEXT NOT NULL
);
```

Stores serialized Double Ratchet sessions as JSON blobs, keyed by peer onion address. Each session contains `sharedMaterial`, `isInitiator`, `sendCounter`, and `recvCounter`.

3.1.6 Group Sender Index Table

```
CREATE TABLE group_sender_index (
  groupId TEXT NOT NULL,
  senderId TEXT NOT NULL,
  nextIndex INTEGER NOT NULL DEFAULT 0,
  PRIMARY KEY (groupId, senderId)
);
```

Tracks the next message index per sender per group for sender-key encryption. This prevents replay attacks by ensuring monotonically increasing indices.

3.1.7 Conversation Preferences Table

```
CREATE TABLE conversation_preferences (
  conversationId TEXT PRIMARY KEY,
  isPinned INTEGER NOT NULL DEFAULT 0,
  pinnedAt INTEGER,
  isArchived INTEGER NOT NULL DEFAULT 0,
  archivedAt INTEGER
);
```

Stores pin/archive state for conversations (both direct and group).

3.1.8 Blocked Users Table

```
CREATE TABLE blocked_users (
  userId TEXT PRIMARY KEY,
  blockedAt INTEGER NOT NULL
);
```

3.1.9 Call Logs Table

```
CREATE TABLE call_logs (
  callId TEXT PRIMARY KEY,
  peerOnion TEXT NOT NULL,
  direction TEXT NOT NULL,
  status TEXT NOT NULL,
  startedAt INTEGER NOT NULL,
  endedAt INTEGER,
  durationMs INTEGER NOT NULL DEFAULT 0
);
CREATE INDEX idx_call_logs_peer ON call_logs(peerOnion);
CREATE INDEX idx_call_logs_started ON call_logs(startedAt DESC);
```

Directions: `inbound` / `outbound`. Statuses: `ringing`, `completed`, `missed`, `declined`, `failed`.

3.2 Migration History

Version	Change
v1	Initial: users, groups, group_members, group_keys
v2	Add avatarBase64 to users
v3	Add customName to users
v4	Re-create group tables if missing
v5	Create conversation_preferences
v6	Add identityJson to users
v7	Create session_state, group_sender_index
v8	Create blocked_users
v9	Create call_logs

4 Messages Database (`messages.db`)

The messages database stores all chat messages, media metadata, reactions, and read receipts across four tables with eleven migration versions.

4.1 Schema Definition

4.1.1 Messages Table

```
CREATE TABLE messages(
  id TEXT PRIMARY KEY,
  senderId TEXT NOT NULL,
  receiverId TEXT NOT NULL,
  message TEXT,
  type TEXT,
  fileName TEXT,
  fileSize INTEGER,
  timestamp INTEGER NOT NULL,
  status TEXT DEFAULT 'sent',
  replyTo TEXT,
  readAt INTEGER,
  viewOnce INTEGER DEFAULT 0,
  viewed INTEGER DEFAULT 0,
  groupId TEXT,
  deletedAt INTEGER,
  editedAt INTEGER
);
```

Indexes provide query performance:

- `idx_conversation(senderId, receiverId)` — direct chat lookups
- `idx_group_messages(groupId, timestamp)` — group chat pagination
- `idx_timestamp(timestamp)` — chronological queries
- `idx_status(status)` — filtering by delivery state
- `idx_read_status(readAt, status)` — unread count queries
- `idx_unread_inbound(senderId, status, readAt)` — per-sender unread
- `idx_direct_peer_ts(senderId, receiverId, timestamp DESC)` — cursor pagination

4.1.2 Storage ID Scheme

Group messages use a scoped identifier to prevent collisions:

$$\texttt{storageId} = \texttt{groupId::wireId}, \qquad \texttt{wireId} = \texttt{storageId.substringAfter(::)}$$

4.1.3 Self Messages Table

```
CREATE TABLE self_messages (
  id TEXT PRIMARY KEY,
  message TEXT,
  type TEXT DEFAULT 'text',
  fileName TEXT,
  fileSize INTEGER,
  timestamp INTEGER NOT NULL,
  replyTo TEXT,
  viewOnce INTEGER DEFAULT 0,
  viewed INTEGER DEFAULT 0,
  deletedAt INTEGER,
  editedAt INTEGER
);
```

Stores notes-to-self messages, using the same database connection as `messages.db`.

4.1.4 Message Reactions Table

```
CREATE TABLE message_reactions(
  targetMessageId TEXT NOT NULL,
  reactorId TEXT NOT NULL,
  emoji TEXT NOT NULL,
  groupId TEXT,
  timestamp INTEGER NOT NULL,
  PRIMARY KEY (targetMessageId, reactorId)
);
```

Indexes: `idx_reactions_target`, `idx_reactions_group`.

4.1.5 Message Read Receipts Table

```
CREATE TABLE message_read_receipts (
  messageId TEXT NOT NULL,
  groupId TEXT,
  readerId TEXT NOT NULL,
  readAt INTEGER NOT NULL,
  PRIMARY KEY (messageId, readerId)
);
```

Index: `idx_read_receipts_group`.

Read receipts track per-reader acknowledgements separately from the local `readAt` column on the messages table, enabling multi-device read state synchronization.

4.2 Migration History

Version	Change
v1	Initial messages table + indexes
v2	Add readAt column + idx_read_status
v3	Add viewOnce + viewed columns
v4	Add groupId + idx_group_messages
v5	Migrate group IDs to scoped format
v6	Add idx_unread_inbound + idx_direct_peer_ts
v7	Create message_reactions table
v8	Add deletedAt + editedAt columns
v9	Create message_read_receipts
v10	Create self_messages table
v11	Blob migration for oversized messages

5 Pending Messages Database (`pending_messages.db`)

The pending messages database implements an outbox pattern for reliable delivery.

```
CREATE TABLE pending_messages(
  id TEXT PRIMARY KEY,
  senderId TEXT,
  receiverId TEXT,
  message TEXT,
  type TEXT,
  fileName TEXT,
  fileSize INTEGER,
  timestamp INTEGER,
  status TEXT,
  replyTo TEXT,
```

```
  viewOnce INTEGER DEFAULT 0,
  groupId TEXT,
  targetMemberId TEXT
);
CREATE INDEX idx_pending_receiver ON pending_messages(receiverId);
CREATE INDEX idx_pending_timestamp ON pending_messages(timestamp);
```

5.1 Outbox Lifecycle

Algorithm 1 Pending Message Lifecycle

Require: Encrypted message payload

1: Insert into pending_messages with status `pending`
2: Deliver via TransportProvider
3: **if** delivery succeeds **then**
4: Update messages.db status to `sent`
5: Remove from pending_messages
6: **else**
7: Keep in pending_messages (in-memory retry queue with exponential backoff)
8: Retry on next maintenance cycle or when peer comes online

6 Message Blob Store

Messages with payloads exceeding 512 KB are offloaded from the SQLite `message` column to individual files.

6.1 Offloading Algorithm

Algorithm 2 Blob Offloading

Require: message content, storageId

1: **if** content.length > inlineThreshold (512 KB) **then**
2: Write content to `message_blobs/`*storageId*
3: Store marker string `"blob:"`*storageId* in SQLite
4: **else**
5: Store content directly in SQLite

On read, markers are resolved transparently by reading from the filesystem. A migration at database open time moves any existing oversized rows to the blob store.

7 Secure Storage

Cryptographic key material is stored using `flutter_secure_storage`, which delegates to platform-specific secure enclaves:

Key	Content	Protection
`ENCRYPTED_IDENTITY_V2`	Ed25519 + X25519 private keys (AES-GCM encrypted)	Passphrase + OS k
`PUBLIC_IDENTITY_V2`	Ed25519 + X25519 public keys + fingerprint	OS keychain
`PASSPHRASE_SALT_V2`	16-byte Argon2id salt	OS keychain
`CRYPTO_GENERATION`	Crypto protocol version marker	OS keychain
`SIGNED_PREKEY_PRIVATE_V2`	X25519 signed prekey seed	OS keychain
`ONETIME_PREKEY_POOL_V2`	JSON array of 16 X25519 key pairs	OS keychain

7.1 Identity Key Flow

1. User enters passphrase on app start
2. Passphrase + salt $\xrightarrow{\text{Argon2id}}$ 32-byte AES key
3. AES-GCM decrypt `ENCRYPTED_IDENTITY_V2` → `IdentityKeyPair`
4. Public keys loaded from `PUBLIC_IDENTITY_V2`
5. Prekey bundle loaded from prekey storage
6. All plaintext keys held in memory until app lock

7.2 Prekey Pool Management

The one-time prekey pool is maintained at 16 entries:

Algorithm 3 One-Time Prekey Pool Replenishment

Require: Pool size n

1: **if** $n <$ `replenishThreshold` (4) **then**
2: Generate $16 - n$ new X25519 key pairs
3: Append to pool
4: Persist updated pool to secure storage

8 Data Flow Architecture

8.1 Message Write Path

1. Message encrypted by crypto layer (Double Ratchet or group sender-key)
2. Encrypted payload passed to `MessagesDb.insertMessage()`
3. SQLite INSERT with status `pending`, blob offloading if > 512 KB
4. `PendingMessageDbHelper.insertPendingMessage()` for outbox
5. Delivery attempted; on success, status updated to `sent`, pending row removed
6. Stream notifier emits new message event to UI

8.2 Message Read Path

1. Query `messages` table by conversation or group with cursor pagination
2. For each result, resolve blob markers to full content
3. Decrypt using stored session state or group key
4. Return decrypted plaintext to UI

8.3 Conversation List

The conversation list is built from a combination of:

- Recent direct messages (aggregated from `messages.db`)
- Recent group messages (aggregated from `messages.db`)
- Self messages (from `self_messages` table)
- Conversation preferences (pin/archive status)

Unread counts are computed via indexed queries on `readAt` and `status`.

9 Panic / Wipe System

9.1 Decoy Mode

The `decoy` panic action replaces the app UI with an empty shell. All data remains encrypted on disk. The cryptographic keys stay in secure storage, protected by the passphrase.

9.2 Wipe Mode

The `wipe` panic action performs complete cryptographic destruction:

Algorithm 4 Panic Wipe Procedure

1: Delete all SQLite database files (chat_app.db, messages.db, pending_messages.db)
2: Delete message blobs directory
3: Clear all flutter_secure_storage entries
4: Clear SharedPreferences
5: Reset crypto generation marker
6: Return to initial setup screen

10 Design Rationale

10.1 Why Three Databases?

The separation serves multiple purposes:

- Isolation of concerns: peer relationships vs. message content vs. delivery queue
- Performance: message queries do not contend with metadata reads
- Wipe granularity: pending messages can be cleared independently

10.2 Why Blob Offloading?

SQLite performance degrades with large string values. The 512 KB threshold ensures that the overwhelming majority of messages remain inline while preventing any single message from causing database bloat. The filesystem is better suited for large binary payloads.

10.3 Why Secure Storage for Keys?

Operating system secure storage provides hardware-backed encryption on supported devices:

- Android: EncryptedSharedPreferences backed by Android Keystore (TEE)
- iOS: Keychain Services (Secure Enclave on supported devices)

This provides an additional layer of protection beyond the Argon2id-derived AES-256-GCM encryption.

11 Constants

Parameter	Value	Context
Inline blob threshold	512 KB	`message_blob_store`
One-time pool size	16	Prekey availability
Replenish threshold	4	Pool replenishment trigger
Database WAL mode	Enabled	Concurrent access
Max message body	Unlimited	Blob offloaded