Prysm Calls

A Comprehensive Analysis of the Voice Call Architecture of the Prysm P2P Encrypted Messenger

xmreur

Version 2.0 — July 2026

Abstract

Prysm implements a fully peer-to-peer voice calling system over Tor hidden services. Unlike conventional VoIP solutions that rely on WebRTC, STUN, TURN, and SIP, Prysm uses a custom protocol stack: call signaling is carried over the existing WebSocket JSON messaging channel, and encrypted Opus audio frames are transmitted as raw binary WebSocket messages. Per-call ChaCha20-Poly1305 session keys are exchanged via the X25519+HKDF key wrapping scheme used throughout the application. This document specifies the call state machine, signaling protocol, audio encryption, Opus codec integration, jitter buffering, and audio preprocessing pipeline.

Keywords: VoIP, Opus, ChaCha20-Poly1305, Tor, peer-to-peer, audio, WebSocket, signaling

Contents

1 Introduction

The Prysm calling system is a fully custom, peer-to-peer voice solution built entirely on top of the application's existing transport infrastructure. It deliberately avoids WebRTC and its dependency on STUN/TURN servers, which would introduce centralized infrastructure incompatible with Prysm's decentralized architecture.

The system comprises four subsystems:

1. **Signaling**: Call offer/answer/end/mute messages over the WebSocket JSON channel
2. **Encryption**: Per-call symmetric key exchanged via X25519+HKDF+AES-GCM wrapping, with per-frame ChaCha20-Poly1305 AEAD
3. **Audio Pipeline**: Opus codec capture/encode/transmit and receive/decode/playback
4. **Preprocessing**: High-pass filtering, adaptive noise reduction, voice activity detection, gain normalization

2 Call State Machine

The `CallManager` (`lib/services/call/call_manager.dart`) implements a state machine with six states:

`idle` → `connecting` → `ringing` → `active` → `ended`

An additional `incoming` state exists on the callee side before acceptance.

State	Description
`idle`	No call active; accepting new signaling events
`connecting`	Outbound: offer sent, awaiting answer
`ringing`	Outbound: peer is being notified
`incoming`	Inbound: offer received, awaiting user decision
`active`	Bidirectional audio streaming
`ended`	Call terminated; cleanup in progress

2.1 State Transition Rules

- `idle` → `connecting`: `startCall()` invoked
- `idle` → `incoming`: `call_offer` received
- `connecting` → `ringing`: Offer dispatched to transport
- `incoming` → `active`: `acceptIncoming()` invoked
- `incoming` → `ended`: `rejectIncoming()` or timeout
- Any → `ended`: `endCall()`, remote hangup, disconnect, error

3 Signaling Protocol

Call signaling is transported as JSON frames over the existing per-peer WebSocket connection. All signaling operations are defined in `wsCallOps`:

{`call_offer`, `call_answer`, `call_end`, `call_mute`}

3.1 Call Offer

The initiator sends a `call_offer` frame:

Listing 1: Call Offer Payload

```
{
  "callId": "<uuid>",
  "sessionId": <random-uint32>,
  "wrappedKey": "<base64-encrypted-key+salt>",
  "codec": {
    "sampleRate": 48000,
    "channels": 1,
    "frameDurationMs": 20
  }
}
```

The `wrappedKey` contains the 32-byte ChaCha20-Poly1305 session key concatenated with a 4-byte salt (36 bytes total), encrypted using `KeyManager.encryptBytesForPeer()`, which applies the standard X25519 ephemeral DH + HKDF + AES-GCM key wrapping (`dh-aead-1`).

3.2 Call Answer

The callee responds with a `call_answer` frame acknowledging the `callId` and `sessionId`. No additional key exchange occurs; the callee derives the same session key by decrypting the wrapped key.

3.3 Call End

Either party sends `call_end` with a reason:

$$\texttt{reason} \in \{\texttt{hangup},\ \texttt{declined},\ \texttt{timeout},\ \texttt{busy},\ \texttt{error}\}$$

3.4 Call Mute

Mute state is toggled via `call_mute` with a boolean `muted` field.

4 Audio Encryption

4.1 Session Key Establishment

Algorithm 1 Call Session Key Establishment

Require: Initiator and responder identities

1: INITIATOR:
2: $key \leftarrow (32)$ ▷ ChaCha20-Poly1305 key
3: $salt \leftarrow (4)$ ▷ Nonce salt
4: $wrapped \leftarrow$ DH-AEAD-ENCRYPT($key \parallel salt,\ initiator,\ pk_{peer}$)
5: Include *wrapped* in `call_offer`
6: RESPONDER:
7: $key \parallel salt \leftarrow$ DH-AEAD-DECRYPT($wrapped,\ responder$)
8: Both parties now share $(key, salt)$

4.2 Per-Frame Nonce Construction

Each audio frame uses a unique nonce derived from the per-call salt and a sequence number:

Algorithm 2 Nonce Derivation for Audio Frame n

Require: Per-call salt S (4 bytes), sequence number n

1: $nonce \leftarrow \texttt{0x00000000}^{12}$ (12 zero bytes)
2: $nonce[0:4] \leftarrow S$
3: $nonce[4:8] \leftarrow n$ (big-endian uint32)
4: **return** *nonce*

This construction ensures that frames within a call never reuse a nonce, as the sequence number is monotonically increasing. The 4-byte salt provides caller-specific separation across different calls.

4.3 Frame Encryption

Algorithm 3 Audio Frame Encrypt/Decrypt

Require: Opus payload P, session key K, salt S, sequence n, session ID I

1: $nonce \leftarrow \textsc{DeriveNonce}(S, n)$
2: $aad \leftarrow \text{UTF-8}(n)$ ▷ Associated data: sequence as string
3: $ct \leftarrow \texttt{.encrypt}(K, nonce, P, aad)$
4: $frame \leftarrow [\texttt{0xA1}, I_4, n_4, ct]$ ▷ 9-byte header + payload
5: **return** *frame*

Decryption reverses the process, verifying the authentication tag and checking that *frame.sessionId* matches the expected session ID. Frames with $seq \leq recvSeq$ are rejected as replays.

4.4 Security Properties

- **Confidentiality**: ChaCha20-Poly1305 provides authenticated encryption
- **Replay protection**: Monotonic sequence numbers in both nonce and associated data
- **Key secrecy**: Session key transported via X25519 + HKDF + AES-GCM
- **Per-call keys**: Fresh random key generated for each call (no key reuse)
- **No forward secrecy within call**: Key is static for the call duration

5 Audio Pipeline

5.1 Complete Chain

Step	Component	Description
1	Mic Capture	Raw PCM16 from microphone
2	Preprocessing	HPF, noise floor, voice gate
3	Gain Normalization	Adaptive RMS targeting
4	Opus Encode	`opus_encode()` frame
5	Encrypt	ChaCha20-Poly1305 per frame
6	Transmit	Binary WebSocket send
7	Receive	Binary WebSocket frame
8	Decrypt	ChaCha20-Poly1305 verify + decrypt
9	Opus Decode	`opus_decode()` to PCM16
10	Jitter Buffer	Smooth bursty delivery
11	Playback	Platform audio output

5.2 Microphone Capture

On Linux, audio capture uses the `pw-record` PipeWire utility. The raw PCM16 stream is read from a pipe and chunked into 20 ms frames at 48 kHz sample rate (960 samples per frame). The `PcmCaptureProcessor` applies preprocessing:

Algorithm 4 Audio Preprocessing Pipeline

Require: Raw PCM16 frame F (960 samples @ 48 kHz)

$F \leftarrow \mathsf{HPF}(F, f_c = 100 \text{ Hz})$ ▷ DC offset removal

$\sigma \leftarrow \text{RMS}(F)$ ▷ Noise floor estimation

if $\sigma < \theta_{noise}$ **then**

  return `silence` ▷ Voice activity gate

$F \leftarrow \textsc{Normalize}(F,\ \mathit{target} = -16 \text{ dBFS})$

return F

The high-pass filter removes sub-100 Hz noise (rumble, handling noise). The adaptive noise floor tracks the minimum RMS over a sliding window and gates transmission when the signal falls below threshold.

5.3 Opus Codec

The `OpusCodec` class wraps the native Opus library via FFI:

- Application: `OPUS_APPLICATION_VOIP` (optimized for voice)
- Sample rate: 48 kHz (Opus requirement)
- Frame duration: 20 ms (960 samples)
- Bitrate: Variable (default 32 kbps)
- Complexity: 5 (balanced)
- Packet loss concealment: Enabled (PLC)

Listing 2: Opus Encoder Configuration

```
opus_encoder_create(48000, 1, OPUS_APPLICATION_VOIP, &error);
opus_encoder_ctl(encoder, OPUS_SET_BITRATE(32000));
opus_encoder_ctl(encoder, OPUS_SET_COMPLEXITY(5));
opus_encoder_ctl(encoder, OPUS_SET_SIGNAL(OPUS_SIGNAL_VOICE));
opus_encoder_ctl(encoder, OPUS_SET_INBAND_FEC(1));
```

5.4 Jitter Buffer

The `PcmJitterBuffer` smooths network-induced delay variation:

- Minimum startup delay: 80 ms (4 frames)
- Maximum latency: 200 ms (10 frames)
- Adaptive: adjusts target buffer level based on observed jitter
- Frames are played at a fixed 20 ms interval via a timer-driven callback

5.5 Playback

Audio playback uses platform-specific backends:

- **Linux**: PipeWire pipe to `pw-play` or `paplay`
- **Android**: `AudioTrack` via platform channel
- **iOS/macOS**: AVAudioEngine via platform channel
- **Desktop fallback**: `JustAudio` package

6 Binary Wire Protocol

Call audio frames use a compact binary format optimized for low-latency transmission:

Listing 3: CallAudioFrame Structure

```
Offset  Size  Field       Description
------  ----  ----------  --------------------------
0       1     magic       Always 0xA1
1       4     sessionId   uint32 big-endian
5       4     seq         uint32 big-endian
9       n     payload     Encrypted Opus frame
                              (ChaCha20-Poly1305)
```

Total header overhead: 9 bytes per 20 ms audio frame.

7 Call Logging

Call history is persisted in the `call_logs` table (`chat_app.db`):

```
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
);
```

The `CallLogsService` provides CRUD operations and is used by the `CallHistoryScreen` UI.

8 Call Overlay UI

During an active call, a full-screen overlay is shown (`CallOverlay`). It displays:

- Peer name and onion address
- Call duration (live counter)
- Mute/unmute toggle button
- End call button
- Speaker/earpiece toggle
- Connection quality indicator (based on jitter buffer statistics)

9 Foreground Session

The `CallForegroundSession` ensures the app stays alive during calls:

- Keeps the WebSocket connection alive via periodic pings
- Prevents Tor from shutting down during calls
- On mobile, acquires a wake lock to prevent CPU sleep
- On desktop, registers a system tray indicator

10 Security Considerations

- **No WebRTC**: Avoids STUN/TURN dependency; all traffic stays within Tor
- **Key exchange reuses existing infrastructure**: No new key agreement protocol
- **Replay protection**: Sequence numbers in nonce and AAD
- **Caller authentication**: Implicit via established WebSocket + ratchet session
- **Busy signaling**: Automatic rejection with `busy` reason when already in call

11 Constants

Parameter	Value	Context
Audio sample rate	48 kHz	Opus requirement
Frame duration	20 ms	960 samples
Opus bitrate	32 kbps	Voice-optimized
Opus complexity	5	Balanced
ChaCha20 key	32 bytes	Per-call random
Nonce salt	4 bytes	Per-call random
Nonce total	12 bytes	Standard ChaCha20-Poly1305
Audio magic byte	0xA1	Binary frame type
Session key transport	dh-aead-1	X25519 + HKDF + AES-GCM
Ring timeout	45 s	Call establishment
Jitter buffer min	80 ms (4 frames)	Startup latency
Jitter buffer max	200 ms (10 frames)	Maximum latency