Prysm Cryptography

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

xmreur

Version 2.0 — July 2026

Abstract

Prysm is a Tor-only peer-to-peer encrypted messenger with no central infrastructure. This document provides a complete specification of its cryptographic protocol stack: identity management via Ed25519/X25519 key pairs, key encapsulation at rest via Argon2id and AES-256-GCM, one-to-one messaging via ephemeral Diffie-Hellman with HKDF key derivation and a Double Ratchet protocol with X3DH-style prekey bundles for forward secrecy, file transfer via ephemeral key wrapping, and group messaging via sender-key encryption with per-sender per-message key derivation and Ed25519 signatures for authentication. Design rationales, security assumptions, and threat models are discussed throughout.

Keywords: end-to-end encryption, Curve25519, Double Ratchet, X3DH, Tor, peer-to-peer, group messaging, forward secrecy

Contents

1 Introduction

Prysm is a peer-to-peer encrypted messenger that routes all traffic exclusively through the Tor network. Every client exposes a Tor hidden service and communicates directly with other clients over `.onion` addresses. There is no central server; each node is simultaneously a client and a server. Messages are encrypted end-to-end before leaving the sender's device and are decrypted only by the intended recipient.

The cryptographic stack is organized in three layers. At the base lies the identity layer, consisting of a long-term Ed25519 signing key and a long-term X25519 Diffie-Hellman key, both generated locally and stored encrypted at rest. On top of this, the messaging layer provides three encryption schemes chosen by protocol: (i) an ephemeral DH-AEAD scheme for simple one-to-one messages, (ii) a Double Ratchet protocol with X3DH-style prekey bundles for forward-secret one-to-one conversations, and (iii) a sender-key scheme for group messaging. A separate file-transfer scheme wraps file-encryption keys using the same DH-AEAD mechanism.

This document specifies all cryptographic primitives, key-derivation procedures, wire formats, and security guarantees of the v2 protocol (current as of Prysm 0.3+). The legacy v1 protocol (RSA-based) is not covered. All primitives are instantiated using the Dart `cryptography` library (v2.7+) with platform-native backends.

2 Notation and Preliminaries

2.1 Sets and Strings

Let $\{0,1\}^*$ denote the set of finite bit strings and $\{0,1\}^n$ the set of n-bit strings. For byte strings we write `0x...` in hexadecimal or use base64 encoding for wire representation. The symbol $\|$ denotes concatenation. $|x|$ denotes the length in bytes. The empty string is denoted ε.

2.2 Cryptographic Primitives

Ed25519.

Edwards-curve Digital Signature Algorithm (Ed25519) is the EdDSA instantiation over the twisted Edwards curve

$$-x^2 + y^2 = 1 + \frac{121665}{121666}\, x^2 y^2$$

which is birationally equivalent to Curve25519. A signing key sk is a 32-byte seed; the public key is $pk = sk \cdot B$ where B is the base point. Signing produces a 64-byte (R, S) signature. Verification checks

$$8S \cdot B = 8R + 8H(R \,\|\, pk \,\|\, m) \cdot pk$$

where H is SHA-512.

X25519.

Curve25519 elliptic-curve Diffie-Hellman key agreement over the Montgomery curve

$$y^2 = x^3 + 486662\, x^2 + x \pmod{2^{255} - 19}.$$

A private key is a 32-byte scalar with clamping (bits 0–2 cleared, bit 254 set, bit 255 cleared). The public key is the x-coordinate of the scalar times the base point $u = 9$. The shared secret is

$$\mathsf{DH}(sk_A, pk_B) = sk_A \cdot pk_B = x\text{-coordinate of } sk_A \cdot sk_B \cdot 9.$$

AES-256-GCM.
AES in Galois/Counter Mode with a 256-bit key, 96-bit nonce, and 128-bit authentication tag. Provides authenticated encryption (AEAD). Encryption is $C = P \oplus E_K(\text{ctr}_1) \parallel E_K(\text{ctr}_2) \parallel \ldots$ with the tag computed via GHASH polynomial evaluation over $\text{GF}(2^{128})$.

ChaCha20-Poly1305.
ChaCha20 stream cipher with Poly1305 MAC, 256-bit key, 96-bit nonce. Also provides AEAD. The ChaCha20 quarter-round is:

$$\begin{aligned} a &\leftarrow a + b, \quad d \leftarrow d \oplus a, \quad d \leftarrow d \lll 16 \\ c &\leftarrow c + d, \quad b \leftarrow b \oplus c, \quad b \leftarrow b \lll 12 \\ a &\leftarrow a + b, \quad d \leftarrow d \oplus a, \quad d \leftarrow d \lll 8 \\ c &\leftarrow c + d, \quad b \leftarrow b \oplus c, \quad b \leftarrow b \lll 7 \end{aligned}$$

SHA-256 / SHA-512.
Standard Secure Hash Algorithm 2 family (NIST FIPS 180-4).

HKDF-SHA256.
HMAC-based Extract-and-Expand Key Derivation Function (RFC 5869):

$$\mathsf{PRK} = \mathsf{HMAC\text{-}SHA256}(\mathit{salt}, \mathsf{IKM}), \qquad \mathsf{OKM} = \mathsf{HMAC\text{-}SHA256}(\mathsf{PRK}, \mathit{info} \parallel 0x01) \parallel \cdots$$

Argon2id.
Memory-hard password-hashing function (RFC 9106). The "id" variant is a hybrid of Argon2i (data-independent memory access) and Argon2d (data-dependent). The compression function G operates on 1024-byte blocks using BLAKE2b:

$$(X, Y) \leftarrow G(X, Y)$$

The memory is a matrix of t rows (iterations) and p columns (lanes), each cell a 1024-byte block.

2.3 Wire Format

All cryptographically protected payloads are transmitted as JSON envelopes with a version field (`"crypto": "v2"`), a scheme identifier, and scheme-specific fields. The supported schemes are:

Scheme	Purpose
`dh-aead-1`	One-to-one DH-AEAD messages
`ratchet-1`	Double Ratchet messages
`group-aead-1`	Group symmetric encryption
`group-sender-1`	Sender-key group messages
`control-wrap-1`	Group key distribution
`file-aead-1`	File transfer encryption
`call-aead-1`	Voice/video call encryption

3 Identity Layer

3.1 Key Generation

Each Prysm identity consists of two Curve25519-based key pairs generated during initial setup: an $\mathsf{Ed25519}$ signing key pair (sk_{sign}, pk_{sign}) for authentication and an $\mathsf{X25519}$ key agreement

pair (sk_{agree}, pk_{agree}) for ECDH. Both keys are generated from fresh entropy via the operating system CSPRNG.

Mathematically, let

$$pk_{sign} = sk_{sign} \cdot B_{ed}, \qquad pk_{agree} = sk_{agree} \cdot 9 \pmod{2^{255} - 19}$$

where B_{ed} is the Ed25519 base point. Both sk_{sign} and sk_{agree} are uniformly random 32-byte values.

3.2 Fingerprint

The identity fingerprint binds the signing and agreement keys:

$$\mathit{fingerprint} = \mathsf{SHA-256}(pk_{sign} \parallel pk_{agree}) \tag{1}$$

This prevents substitution attacks where an adversary presents a valid signature from one identity but uses a different agreement key for decryption. The fingerprint is exposed as a hex string for out-of-band verification.

3.3 Signing and Verification

The Ed25519 signing key authenticates protocol messages including prekey bundles and group sender-key messages. Signing follows RFC 8032:

Algorithm 1 Ed25519 Sign

Require: Seed sk_{sign}, message m

1: $(r, R) \leftarrow \mathsf{SHA-512}(sk_{sign})[32:]$ (deterministic nonce)
2: $S \leftarrow (r + H(R \parallel pk_{sign} \parallel m) \cdot sk_{sign}) \bmod \ell$
3: **return** (R, S)

Verification checks $8S \cdot B = 8R + 8H(R \parallel pk_{sign} \parallel m) \cdot pk_{sign}$.

3.4 QR Payload

Identity public keys are exchanged via QR codes (in-person) or through the prekey bundle endpoint (online). The QR payload format is:

`prysm:v2:`⟨`onion_address`⟩`:`⟨`fingerprint`⟩

This is the root trust anchor: users verify the fingerprint out of band.

4 Key Derivation

4.1 Argon2id — Password-Based Key Derivation

Identity private keys are encrypted at rest with a key derived from the user's passphrase. Prysm uses Argon2id with parameters:

Parameter	Value	Rationale
Memory m	65,536 KiB (64 MiB)	Memory-hardness
Iterations t	3	Time-cost balance
Parallelism p	1	Single-threaded
Output length	32 bytes	AES-256 key
Salt	16 bytes	Fresh per encryption

Argon2id operates by filling a memory matrix of m blocks with BLAKE2b-based compression. The "id" variant mixes data-independent and data-dependent passes. The derived key is:

$$K = \mathsf{Argon2id}(P, S, m = 65536, t = 3, p = 1, d = 32)$$

where P is the passphrase and S is the salt.

4.2 HKDF-SHA256 — Protocol Key Derivation

All protocol-level key derivation uses HKDF (RFC 5869). The two-stage process is:

Extract: $\mathsf{PRK} = \mathsf{HMAC\text{-}SHA256}(\mathit{salt}, \mathsf{IKM})$

Expand: $\mathsf{OKM} = T(1) \parallel T(2) \parallel \ldots$ where

$$T(i) = \mathsf{HMAC\text{-}SHA256}(\mathsf{PRK}, T(i-1) \parallel \mathit{info} \parallel i)$$

Domain separation is achieved through unique *info* strings:

- `"prysm/dh-aead-1"` — DH-AEAD message keys
- `"prysm/ratchet"` — Double Ratchet root material
- `"prysm/group-key-wrap"` — Group key wrapping
- `"prysm/call-session"` — Call session keys

For DH-AEAD, the ephemeral public key is used as the HKDF salt, binding each derived key to a specific DH exchange.

4.3 Cryptographically Secure Random Bytes

All nonces, salts, ephemeral keys, and file-encryption keys use the operating system's CSPRNG (`/dev/urandom` on Linux, `SecureRandom` on Android, `SecRandomCopyBytes` on iOS).

5 AEAD Encryption

All bulk data encryption uses authenticated encryption with associated data (AEAD). Two algorithms are supported.

5.1 AES-256-GCM

Galois/Counter Mode converts AES into a stream cipher. The keystream is $E_K(\mathrm{ctr}_1) \parallel E_K(\mathrm{ctr}_2) \parallel \ldots$ where $\mathrm{ctr}_i = N \parallel i$ with N the 96-bit nonce. GHASH authentication uses polynomial evaluation over $\mathrm{GF}(2^{128})$ with $H = E_K(0^{128})$:

$$\mathrm{GHASH}(H, A, C) = \sum_{i=1}^{m} A_i H^{m+n+1-i} + \sum_{i=1}^{n} C_i H^{n+1-i} + L \cdot H$$

where L encodes the bit lengths of A and C.

5.2 ChaCha20-Poly1305

ChaCha20 generates a keystream by applying 20 rounds of the ChaCha quarter-round to a 4×4 state matrix $\sigma, K, N, \mathrm{ctr}$. Poly1305 is a one-time MAC:

$$\mathrm{Poly1305}_r(A, C) = \left(\sum a_i r^{n+m+1-i} + \sum c_i r^{m+1-i} + \ell \right) \bmod 2^{128}$$

with r derived from the ChaCha20 keystream at block 0.

6 One-to-One Messaging: DH-AEAD

6.1 Motivation

The `dh-aead-1` scheme provides end-to-end encryption for one-off messages and serves as a fallback when no ratchet session exists.

6.2 Encryption Protocol

Algorithm 2 DH-AEAD Encrypt (`dh-aead-1`)

Require: *plaintext* M, sender keypair (sk_a, pk_a), recipient X public key pk_{peer}

1: $(ek, epk) \leftarrow \mathsf{X25519.newKeyPair}()$ ▷ Ephemeral key
2: $shared \leftarrow \mathsf{DH}(ek, pk_{peer})$
3: $aead\_key \leftarrow \mathsf{HKDF}(\mathsf{IKM} = shared,\ salt = epk,\ info = \texttt{"prysm/dh-aead-1"})$
4: $nonce \leftarrow \mathsf{CSPRNG}(12)$
5: $(ct, tag) \leftarrow \mathsf{AES\text{-}256\text{-}GCM.encrypt}(aead\_key, nonce, M)$
6: **return** {`crypto` : "v2", `scheme` : "dh-aead-1", `ephemeralPub` : epk, `nonce` : $nonce$, `ciphertext` : $ct \parallel tag$}

6.3 Decryption Protocol

Algorithm 3 DH-AEAD Decrypt

Require: *envelope*, recipient keypair (sk_a, pk_a)

1: Parse envelope: extract epk, $nonce$, $ct \parallel tag$
2: $shared \leftarrow \mathsf{DH}(sk_a, epk)$
3: $aead\_key \leftarrow \mathsf{HKDF}(\mathsf{IKM} = shared,\ salt = epk,\ info = \texttt{"prysm/dh-aead-1"})$
4: $M \leftarrow \mathsf{AES\text{-}256\text{-}GCM.decrypt}(aead\_key, nonce, ct \parallel tag)$
5: **return** M

6.4 Security Properties

Confidentiality. Only the holder of sk_a or ek can compute $shared$.

Forward secrecy. Compromise of long-term keys does not reveal past messages because each message uses an ephemeral key.

KCI resistance. Compromise of sk_a does not permit decryption of messages where the user was the recipient (the ephemeral key is unknown).

No sender authentication. The scheme does not include a signature. Higher layers provide sender verification.

6.5 Self-Encryption

Encrypting a message to oneself uses the same protocol with pk_{self} as the recipient. The ephemeral key ensures self-encrypted ciphertexts are indistinguishable from peer-encrypted ones.

7 Double Ratchet Protocol

7.1 Overview

The Double Ratchet protocol provides forward secrecy and future secrecy (post-compromise security) for ongoing conversations. Prysm's implementation combines X3DH-style prekey bundles

for session initiation with a simplified symmetric-key ratchet.

7.2 Prekey Bundle (X3DH-Style)

A prekey bundle contains a signed prekey, its Ed25519 signature, and a one-time prekey.

Algorithm 4 Prekey Bundle Generation

Require: Identity keypair (sk_s, pk_s, sk_a, pk_a)

$(ssk, spk) \leftarrow \mathsf{X25519.newKeyPair}()$ ▷ Signed prekey

$sig \leftarrow \mathsf{Ed25519.sign}(sk_s,$ `"prysm-prekey:"` $\| spk)$

$otpk \leftarrow$ next key from one-time pool (size 16)

Store ssk persistently

Consume $otpk$ from pool; replenish to 16

return $\{spk, sig, otpk\}$

Bundle verification checks the Ed25519 signature against the identity's pk_{sign}.

7.3 Shared Secret Derivation (X3DH)

The initiator derives a 128-byte shared secret from four Diffie-Hellman operations:

Algorithm 5 X3DH Shared Secret — Initiator

Require: Local (sk_a, pk_a), peer $(pk_A, spk, otpk)$, ephemeral (ek, epk)

$\mathsf{DH}_1 \leftarrow \mathsf{DH}(sk_a, spk)$ ▷ Identity to signed prekey

$\mathsf{DH}_2 \leftarrow \mathsf{DH}(ek, pk_A)$ ▷ Ephemeral to identity

$\mathsf{DH}_3 \leftarrow \mathsf{DH}(ek, spk)$ ▷ Ephemeral to signed prekey

$\mathsf{DH}_4 \leftarrow \mathsf{DH}(ek, otpk)$ ▷ Ephemeral to one-time

return $\mathsf{DH}_1 \| \mathsf{DH}_2 \| \mathsf{DH}_3 \| \mathsf{DH}_4$

The responder computes the complementary operations using its stored ssk and the consumed one-time key:

Algorithm 6 X3DH Shared Secret — Responder

Require: Local (sk_a, pk_a, ssk), peer pk_A, initiator epk, consumed $otpk$

$\mathsf{DH}_1 \leftarrow \mathsf{DH}(ssk, pk_A)$

$\mathsf{DH}_2 \leftarrow \mathsf{DH}(sk_a, epk)$

$\mathsf{DH}_3 \leftarrow \mathsf{DH}(ssk, epk)$

$\mathsf{DH}_4 \leftarrow \mathsf{DH}(otpk_{priv}, epk)$

return $\mathsf{DH}_1 \| \mathsf{DH}_2 \| \mathsf{DH}_3 \| \mathsf{DH}_4$

The four DH computations ensure:

- DH_1: identity of the initiator is bound to the signed prekey
- DH_2: ephemeral key is bound to the responder's identity
- DH_3: ephemeral key is bound to the signed prekey
- DH_4: one-time prekey ensures the shared secret is unique per session

7.4 Message Key Derivation

Rather than a full DH ratchet at every step, Prysm derives message keys from the shared material using HKDF with monotonic counters:

Algorithm 7 Message Key Derivation

Require: *shared_material*, role $\in \{\mathsf{send}, \mathsf{recv}\}$, *counter*, *is_initiator*

salt ← `"prysm/ratchet/root-salt"`
info ← `"prysm/ratchet/"` ‖ *role* ‖ `"/msg/"` ‖ *counter*
msg_key ← HKDF(IKM = *shared_material*, *salt* = *salt*, *info* = *info*)
return *msg_key*

The initiator uses role send for outbound and recv for inbound; the responder uses the opposite. The *recvCounter* starts at -1, so the first expected message has *counter* $= 0$. Replay protection rejects messages with *counter* $\leq$ *recvCounter*.

7.5 Security Properties

Forward secrecy (past). Compromise of *shared_material* does not reveal past message keys because each key is derived with an irreversible HKDF step.

Future secrecy (limited). Full post-compromise security would require periodic DH ratchet exchanges, which is a planned enhancement.

Replay protection. The *recvCounter* monotonic invariant prevents replay.

8 File Transfer

8.1 Encryption

File transfer uses a hybrid encryption scheme. The file body is encrypted with a random symmetric key, and that key is wrapped for each recipient using DH-AEAD:

Algorithm 8 File Encryption (`file-aead-1`)

Require: *file_bytes*, sender identity, recipient pk_{peer}

file_key ← CSPRNG(32)
$(nonce, ct)$ ← AES-256-GCM.encrypt(*file_key*, *file_bytes*)
peer_wrapped ← DH-AEAD.encrypt(*file_key*, *sender*, pk_{peer})
self_wrapped ← DH-AEAD.encrypt(*file_key*, *sender*, pk_{self})
return {`peerPayload` : *peer_wrapped*, `selfPayload` : *self_wrapped*}

8.2 Decryption

Algorithm 9 File Decryption

Require: `file-aead-1` envelope, recipient identity

Extract *wrappedKey*, *nonce*, *ciphertext*
file_key ← DH-AEAD.decrypt(*wrappedKey*, *recipient*)
bytes ← AES-256-GCM.decrypt(*file_key*, *nonce*, *ciphertext*)
return *bytes*

9 Group Messaging

9.1 Group Encryption (`group-aead-1`)

Group messages are encrypted with a 32-byte shared group key. All members share the same key:

$$\mathsf{Encrypt}(K, M) \implies \mathsf{AES\text{-}256\text{-}GCM}(K, N, M)$$

9.2 Sender-Key Encryption (`group-sender-1`)

To provide per-message authentication, each sender derives per-message keys from a shared epoch key using their identity and a monotonic index:

Algorithm 10 Sender-Key Message Key Derivation

Require: *epoch_key*, *sender_id*, *message_index*

1: $info \leftarrow$ `"prysm/group-sender/"` $\|$ *sender_id* $\|$ `"/"` $\|$ *message_index*
2: $msg\_key \leftarrow \mathsf{HKDF}(\mathsf{IKM} = epoch\_key,\ salt =$ `"prysm/group-sender-salt"`$,\ info = info)$
3: **return** *msg_key*

Each message includes an Ed25519 signature over the metadata and ciphertext:

Algorithm 11 Sender-Key Encrypt

Require: *epoch_key*, *group_id*, *sender_id*, *index*, M, *sender_kp*

1: $msg\_key \leftarrow \mathsf{Derive}(epoch\_key,\ sender\_id,\ index)$
2: $(iv, ct) \leftarrow \mathsf{AES\text{-}256\text{-}GCM.encrypt}(msg\_key, M)$
3: $sig \leftarrow \mathsf{Ed25519.sign}(sk_s,\ group\_id \mid sender\_id \mid index \mid iv \mid ct)$
4: **return** {`scheme` : "`group-sender-1`", `senderId`, `index`, `iv`, `ct`, `sig`}

Recipients verify the signature before deriving the message key and decrypting.

9.3 Group Control Payloads (`control-wrap-1`)

Group keys are distributed via control payloads that wrap a fresh session key under DH-AEAD and encrypt the control message under that session key:

$$\mathsf{ControlEnc}(K_{peer}, M) = \mathsf{AES\text{-}256\text{-}GCM}(K_{session}, M) \quad \text{where} \quad K_{session} = \mathsf{DH\text{-}AEAD.encrypt}(epk, K_{peer})$$

9.4 Group File Encryption

Group files use a two-layer scheme: a random file key encrypts the file, and the file key is encrypted under the group key:

$$file\_key \xrightarrow{\mathsf{AES\text{-}256\text{-}GCM}} file\_ct, \qquad file\_key \xrightarrow{\mathsf{AES\text{-}256\text{-}GCM}(K_{group})} wrapped\_key$$

10 Key Storage at Rest

10.1 Identity Keystore

Identity private keys are encrypted with AES-256-GCM keyed by Argon2id-derived material, then stored in platform secure storage.

Algorithm 12 Identity Keystore Encryption

Require: Passphrase P, identity (sk_s, pk_s, sk_a, pk_a)

1: $S \leftarrow \mathsf{CSPRNG}(16)$
2: $K \leftarrow \mathsf{Argon2id}(P, S,\ m = 65536,\ t = 3,\ p = 1,\ d = 32)$
3: $(iv, ct) \leftarrow \mathsf{AES\text{-}256\text{-}GCM.encrypt}(K,\ private\_json)$
4: Store in secure storage: {`keystore` : "2", `iv`, `ct`, `salt`}

10.2 Prekey Storage

The signed prekey private key is stored under `SIGNED_PREKEY_PRIVATE_V2`. One-time prekeys are stored as a JSON array under `ONETIME_PREKEY_POOL_V2`, maintained at size 16 with a replenish threshold of 4.

10.3 Crypto Migration from v1

Prysm 0.2.x used RSA keys. Migration is not automatic: the app detects legacy keys and prompts the user to wipe all data, as the threat models differ fundamentally.

11 Threat Model and Security Analysis

11.1 Assumptions

- The Tor network provides transport-layer anonymity and integrity
- Curve25519 DLP is computationally intractable
- AES-256-GCM and ChaCha20-Poly1305 are secure under their nonce constraints
- SHA-256, SHA-512, and BLAKE2b behave as random oracles
- The device CSPRNG provides unbiased, unpredictable output
- The user's passphrase has sufficient entropy (≥ 12 characters)

11.2 Threats Mitigated

- **Passive eavesdropping:** All messages AEAD-encrypted end-to-end
- **Man-in-the-middle (offline):** Out-of-band fingerprint verification
- **Key-compromise impersonation:** DH-AEAD is KCI-resistant
- **Forward secrecy:** Ephemeral keys and ratchet protect past sessions
- **Replay attacks:** Monotonic counters and fresh ephemeral keys
- **Deniability:** X3DH allows both parties to compute the shared secret
- **Offline password guessing:** Argon2id with 64 MiB raises attack cost

11.3 Limitations

- **No perfect forward secrecy within sessions:** The simplified ratchet does not perform periodic DH exchanges
- **No post-quantum security:** All primitives are vulnerable to Shor's algorithm
- **Group metadata protection:** Group membership is not encrypted
- **One-time prekey exhaustion:** If the pool is not replenished, DH_4 is omitted
- **Side-channel resistance:** Application-layer constant-time is limited to comparison operations

12 Constants and Parameters

Parameter	Value	Rationale
aeadKeyLength	32 bytes	AES-256
gcmNonceLength	12 bytes	Standard AES-GCM IV
saltLength	16 bytes	128-bit salt
argon2MemoryKiB	65,536 (64 MiB)	Memory-hardness
argon2Iterations	3	Time-cost balance
argon2Lanes	1	Single-threaded
minPassphraseLength	12 chars	Entropy floor
oneTimePoolSize	16	Prekey availability
oneTimeReplenishThreshold	4	Replenish trigger
hkdfOutputLength	32 bytes	AES-256 key size

12.1 HKDF Domain Separation

```
"prysm/dh-aead-1"       -- DH-AEAD message key
"prysm/ratchet"         -- Ratchet root material
"prysm/group-key-wrap"  -- Group key wrapping
"prysm/call-session"    -- Call encryption
"prysm/group-sender/<sid>/<idx>" -- Sender key per-message
"prysm/ratchet/<role>/msg/<cnt>"  -- Ratchet message key
"prysm-prekey:"         -- Prekey signature payload prefix
```

13 Conclusion

Prysm's cryptographic architecture provides a defense-in-depth approach to secure peer-to-peer messaging. The identity layer anchors trust in Ed25519/X25519 key pairs with out-of-band fingerprint verification. The messaging layer offers three complementary schemes: DH-AEAD for simple ephemeral encryption, a Double Ratchet protocol with X3DH-style prekey bundles for forward-secret conversations, and sender-key encryption for authenticated group messaging. File transfers use a hybrid envelope scheme, and all keys are protected at rest by Argon2id-derived AES-256-GCM encryption within platform secure storage.

The protocol is designed for algorithm agility through versioned JSON envelopes, allowing cryptographic primitives to be upgraded without breaking compatibility. The choice of Curve25519-based primitives reflects the current consensus in applied cryptography for high-security, high-performance messaging.

Future work includes: (i) full DH ratchet within sessions for perfect post-compromise security, (ii) post-quantum hybrid key encapsulation (e.g., X25519Kyber), (iii) metadata hiding for group operations, and (iv) formal verification of the protocol implementation against this specification.

SIGNED: `xmreur`
DATE: July 2026