Kryptark

Post-Quantum Encrypted Chat on X1 Blockchain

ML-KEM-768 Ed25519 AES-256-GCM HKDF-SHA256 Stealth Payments IPFS Archive Sovereign Identity

Federation X1 Sovereign Stack X1NS Aliasing Group Conversations Self-Custodial AI Trading Agents Mobile Onboarding

TECHNICAL WHITE PAPER
Version 5.0 July 2026

Component	Technology	Standard
Key Exchange	ML-KEM-768 (Kyber)	NIST FIPS 203
Signing	Ed25519	RFC 8032
Symmetric Enc.	AES-256-GCM	NIST SP 800-38D
Key Derivation	HKDF-SHA256	RFC 5869
Stealth Payments	X25519 ECDH + HKDF	RFC 7748
File Transfer	ML-KEM-768 (v3, current) + X25519 (legacy)	FIPS 203 / RFC 7748
Message Archive	HKDF-SHA256 self-derived (v2)	PQ-safe, no DH/KEM
Sovereign Identity	contactId via HKDF	Wallet-unlinkable
Mobile Onboarding	Soft wallet + WebAuthn passkey	FIDO2 PRF extension
Federation	xCFP relay protocol	RFC-style Ed25519
X1NS Aliasing	SPL Name Service	X1 mainnet
Group Conversations	Pairwise fan-out, no shared key	No re-keying on removal
AI Trading Agents	Self-custodial, own PQ identity + wallet	Owner policy, runtime-enforced
Voice/Video	WebRTC DTLS-SRTP	RFC 5764
Blockchain	Solana-compatible / X1 Mainnet	Anchor Framework

Live at https://kryptark.com X1 Mainnet Federated Relay Network

Table of Contents

1 Introduction and Mission

Kryptark is a post-quantum encrypted, blockchain-anchored instant messaging platform built on the X1 blockchain. It combines six technology domains that have historically never appeared together in a single consumer product: **post-quantum cryptography**, **sovereign decentralised identity**, **zero-knowledge server architecture**, **privacy-preserving payments**, **federated relay network**, and a **four-layer decentralised internet stack**.

NAMING

Kryptark is the product. "xchat" is the protocol/engine name baked into the code, file names, and message tags below.

A note on naming, since both names appear throughout this document: the underlying protocol and engine were built under the internal name **xchat** — this is why source files, npm-style module names, on-chain program comments, and message-type tags (`pq_message`, `xchat-file-v3`, `xchat-pq-v1`) all carry that name. The public product built on top of it is branded **Kryptark** — identity anchor transactions are signed with the memo "Kryptark · Post-Quantum Identity · ML-KEM-768 on X1", and the wallet signature that derives every key is requested against the string "Kryptark Post-Quantum Identity". Where this document quotes code or on-chain seeds, the xchat name is used verbatim for accuracy; everywhere else, Kryptark is the name of the product. The xchat protocol and engine were originally built by **Jack Levin**, founder of the X1 blockchain.

The project was built in response to a convergence of threats that existing messaging platforms have failed to address. Centralised servers accumulate message histories, identity graphs, and encryption keys. Quantum computers are advancing toward breaking the elliptic-curve and RSA cryptography that underpins virtually every secure messaging system in use today. And the practice of using a wallet address as a user identity creates an immutable, permanent on-chain privacy leak for every Web3 user.

Core Mission:

* Deliver provably private, post-quantum-safe messaging with fresh per-message key material, and be honest in this document about exactly what that does and does not guarantee (Section 4).
* Eliminate the server as a point of trust: the relay stores nothing, knows nothing.
* Anchor identity on-chain without linking it to a wallet address or any real-world identity.
* Integrate payments natively: every conversation is also a payment channel, with full privacy available.
* Ensure permanent message history through encrypted IPFS archives, not held hostage by any server.
* Build a federated relay network so no single operator controls the infrastructure.
* Onboard anyone in a browser tab, with or without an existing wallet or extension.
* Provide a programmable AI agent interface so autonomous agents can participate as first-class citizens.

LIVE

In production at https://kryptark.com on X1 mainnet since April 2026, federated across two independently operated relays.

2 The Problem with Current Messaging Platforms

2.1 Centralised Trust is a Single Point of Failure

Signal, Telegram, WhatsApp, and iMessage all rely on centralised infrastructure. Even when end-to-end encryption is used, the servers retain metadata: who talked to whom, when, how frequently, and for how long. A court order, a rogue employee, or a data breach can expose the full social graph of every user.

2.2 Quantum Computers Will Break Today's Crypto

The elliptic-curve Diffie-Hellman and RSA key exchanges that protect most messaging platforms today are vulnerable to Shor's algorithm. NIST finalised the first post-quantum standards in August 2024 (FIPS 203, 204, 205) because this threat is considered credible within the 10-15 year horizon. "Harvest now, decrypt later" attacks are already underway: adversaries record encrypted traffic today to decrypt once quantum hardware matures.

2.3 Wallet Address as Identity is a Privacy Catastrophe

In Web3, most dApps use the wallet address directly as the user identity. This creates a permanent, irreversible on-chain fingerprint. Every transaction, every contract interaction, every message is linked to that address forever. Kryptark breaks this link with a derived contactId that is cryptographically unlinkable to the wallet address.

2.4 Payments Leak Identity On-Chain

Standard on-chain payments link sender and recipient wallet addresses permanently and publicly. Kryptark offers an optional stealth payment system that routes a payment through a one-time stealth address, breaking the on-chain link between the two parties.

2.5 Message History Held Hostage by Servers

When a server goes down, your message history disappears. Kryptark's IPFS archive ensures encrypted conversation history is stored in a content-addressed network, anchored on-chain so it is always discoverable by the parties involved.

2.6 Single-Operator Infrastructure is Censorship-Prone

Any messaging system with a single relay operator can be shut down, censored, or compelled by legal process to block users. Kryptark's federated relay network (xCFP) allows multiple independent operators to run relays that interoperate. No single operator can silence the network.

2.7 "Connect a Wallet" is a Barrier, Not a Front Door

Requiring a browser extension wallet before a user can send a single message excludes most of the mobile web. Kryptark generates a wallet in the browser itself and protects it with a device passkey, so a new user can start messaging with nothing installed beforehand — see Section 6.

Feature	Signal	Telegram	WhatsApp	Matrix	Kryptark

E2E encryption	YES	Opt-in	YES	YES	**YES**
Post-quantum crypto	NO	NO	NO	NO	**YES ML-KEM-768**
No phone/email reg.	NO	NO	NO	NO	**YES wallet or soft wallet**
Wallet-unlinkable ID	NO	NO	NO	NO	**YES contactId**
No server key/history	NO	NO	NO	NO	**YES relay only**
Stealth payments	NO	NO	NO	NO	**YES optional, 2-tx unlinked**
Encrypted IPFS archive	NO	NO	NO	NO	**YES**
Forward secrecy (ratchet)	YES	Opt-in	YES	YES	**Roadmap (Section 25)**
Federated relays	NO	NO	NO	YES	**YES xCFP**
No-extension onboarding	NO	NO	NO	NO	**YES soft wallet + passkey**
Self-custodial AI agents	NO	Bots (3rd-party, custodial)	NO	NO	**YES, own wallet + policy**

3 Architecture Overview

Kryptark follows a strict separation of concerns. The browser is the only place where plaintext and private keys ever exist. The relay server is a thin pipe that routes opaque ciphertext blobs. The blockchain provides tamper-evident identity commitment. IPFS stores encrypted file bytes and conversation archives. The xCFP federation layer enables relay-to-relay message delivery. The X1 Sovereign Stack layers name resolution, web hosting, and browser-native DNS on top. Each layer independently upholds the privacy model.

Layer	Component	Role	Knows Plaintext?
Client (Browser)	xchat-pq-crypto.js xchat.js	Derive keys, encrypt/decrypt all messages and files	YES only layer
Relay (Server)	server.js WebSocket /ws	Route ciphertext by contactId Queue offline messages	NO opaque blobs only
Federation (xCFP)	POST /api/relay/inbound Ed25519 signed envelope	Relay-to-relay message delivery Gossip peer discovery	NO ciphertext only
Blockchain (X1)	xchat_pq_v2 (F3ydfN...GduB) + xchat-sovereign PDA per contactId	Commit identity key_hash Anchor IPFS CIDs on-chain Relay + identity v3 trust chain	NO hashes only
X1NS (mainnet)	SPL Name Service + X1NS Registrar	Human-readable name -> contactId aliasing	NO hashes only
IPFS (vault.x1.xyz)	Kubo RPC API pin=true — single gateway today	Store encrypted message batches E2E file ciphertext	NO ciphertext only
X1 Sovereign Stack	xchat-publish CLI Chrome MV3 Extension x1-bootstrap	Publish websites to IPFS Resolve *.x1 names in browser Bootstrap resolver for non-extension	NO
WebRTC (P2P)	DTLS-SRTP STUN	Direct peer media stream No server in media path	Peers only E2E

Key Design Principles

* **Server-Zero:** The relay routes messages without reading them. No message history is persisted.
* **Client-Sovereign Keys:** All private keys are derived in the browser. The server never receives a key.
* **Sovereign Identity:** Your contactId is HKDF-derived from your signing key and never itself reveals your wallet — see Section 5 for the one known exception (on-chain archive anchoring).
* **Per-Message Key Freshness:** Every message uses a fresh ML-KEM encapsulation against the recipient's static public key, giving ciphertext unlinkability -- not forward secrecy (Section 4.2).
* **Permanent Archive:** Conversation history encrypted on IPFS, CID anchored on-chain forever.
* **Federation:** Multiple independent relay operators, none can censor the network.
* **No hard extension requirement:** a browser-generated soft wallet gets a new user to the same identity, on the same chain, with the same guarantees.
* **Defence in Depth:** Encryption, authentication, routing privacy, and payment unlinkability are independent layers.

4 Cryptographic Protocol in Depth

4.1 Identity Derivation

When a user connects a wallet (or creates a soft wallet — Section 6), Kryptark requests a single deterministic signature of the string "Kryptark Post-Quantum Identity". This 64-byte Ed25519 signature becomes the root entropy for all key derivation. The wallet is never touched again for messaging.

```
// ML-KEM-768 keypair mlKemSeed = HKDF-SHA256(walletSig, info="xchat-pq-v1-mlkem768", len=64) {mlKemPub, mlKemSec} = ML_KEM_768.keygen(mlKemSeed) // Ed25519 signing keypair signSeed = HKDF-SHA256(walletSig, info="xchat-pq-v1-signing", len=32) signPrivKey = signSeed signPubKey = Ed25519.getPublicKey(signSeed) // stealthId: message-signing recipient binding only (not used for routing) stealthId = SHA-256(mlKemPubKey) // 32 bytes // contactId: public messaging identity (wallet-unlinkable) contactId = HKDF-SHA256(signPrivKey, info="xchat-contact-id-v1", len=32) contactIdB58 = base58(contactId) // pdaSeed: private on-chain anchor seed, never shared pdaSeed = HKDF-SHA256(signPrivKey, info="xchat-pda-seed-v1", len=32) // keyHash: MITM protection in contact bundles keyHash = SHA-256(mlKemPubKey || signPubKey)
```

4.2 ML-KEM-768 Key Encapsulation

ML-KEM-768 (NIST FIPS 203) replaces Diffie-Hellman for key establishment. A fresh encapsulation is performed for every message, against the recipient's **static** ML-KEM-768 public key -- the same key re-derived from their one seed, permanently. This gives real per-message semantic security and ciphertext unlinkability (no shared secret is ever reused across two messages) -- but it is **not** forward secrecy. Decapsulation only ever needs that one long-term secret key, and there is no ratchet, so identity (seed) compromise retroactively decrypts every message ever sent to that identity that an attacker also captured (relay logs, an intercepted IPFS archive, or a malicious relay operator). Strict forward secrecy would need per-message or per-session ephemeral keys deleted after use on both sides -- a real protocol addition (a post-quantum ratchet), not yet built; see Section 25. Ciphertext is 1088 bytes vs 64 bytes for ECDH-P256.

Parameter	ML-KEM-768	ECDH-P256 (classic)
Public key size	1 184 bytes	64 bytes
Ciphertext size	1 088 bytes	64 bytes
Shared secret	32 bytes	32 bytes
Security level	NIST Level 3 (AES-192+)	NIST Level 1 (~AES-128)
Quantum-safe	YES (lattice, Module-LWE)	NO (Shor breaks ECDH)
Key freshness per use	YES (fresh encap/message)	Depends on the protocol built on top
Forward secrecy	NO (needs a ratchet -- not built; see 4.2)	NO (needs a ratchet -- not built here either)

4.3 AES-256-GCM Payload Encryption

```
msgKey = HKDF-SHA256(sharedSecret, info="xchat-pq-v1-msg", len=32) iv = randomBytes(12) ciphertext = AES-256-GCM.encrypt(msgKey, iv, JSON.stringify(payload)) blob = iv || ciphertext // opaque to relay
```

4.4 Ed25519 Message Authentication

Each payload carries an Ed25519 signature bound to content, timestamp, and recipient stealthId. Messages without a valid signature are dropped with a fatal error. The signature lives inside the AES-GCM ciphertext, invisible to the relay. Ed25519 is also used for relay WebSocket authentication, identity registration authorization, X1NS name-claim proofs, and federation envelope signing.

```
digest = SHA-256(content_bytes || timestamp_u64_be || recipientStealthId) sig = Ed25519.sign(digest, signPrivKey) // Recipient drops message if sig missing or invalid (fatal)
```

4.5 Key Derivation Layers

HKDF-SHA256 (RFC 5869) is used at every derivation boundary to ensure domain separation. Different info strings produce independent, non-correlatable keys from the same root material.

HKDF Context	Info String	Output Use
walletSig root	"xchat-pq-v1-mlkem768"	ML-KEM-768 seed (64 bytes)
walletSig root	"xchat-pq-v1-signing"	Ed25519 seed (32 bytes)
signPrivKey	"xchat-contact-id-v1"	contactId (32 bytes)
signPrivKey	"xchat-pda-seed-v1"	private on-chain PDA seed
signPrivKey	"xchat-storage-v1"	localStorage-at-rest AES key
signPrivKey + salt=theirPub	"xchat-archive-v2"	self-owned IPFS archive key (v2, no DH)
X25519 ECDH (stealth)	"xchat-stealth-v1"	stealthSeed for payment
signPrivKey + txSig salt	"xchat-payment-receipt-v1"	receipt AES key
ML-KEM sharedSecret	"xchat-pq-v1-msg"	per-message AES key
ML-KEM sharedSecret	"...x1-file-v3"	file key (v3, PQ-safe)

5 Sovereign Identity: Zero Wallet Linkage

Kryptark's identity model solves a fundamental problem: how to publish a cryptographic identity that is verifiable and tamper-evident, without linking it to the wallet address that paid for it, and without linking it across different communication partners.

The Three-Layer Identity Stack

Layer	Identifier	Derived From	On-Chain	Wallet Linkable?
Payment	Wallet addr	Key generation	YES always	IS the wallet
Anchoring	pdaSeed	HKDF(signPrivKey)	YES (PDA)	NO: private seed
Messaging	contactId	HKDF(signPrivKey)	NO	NO: separate derivation

The pdaSeed: On-Chain Anchor

The identity anchor PDA is keyed by the contactId itself, but the anchor is discovered and re-derived through a second, private HKDF output — pdaSeed — that is never shared with anyone, including contacts. This means an observer who has your contactId (which you hand out to message people) still cannot derive your message-archive PDAs without also holding your signing key. The identity PDA stores: contactId and key_hash = SHA-256(mlKemPubKey || signPubKey).

```
walletSig --> HKDF --> signSeed --> signPrivKey | contactId = HKDF(signPrivKey,
"xchat-contact-id-v1") pdaSeed = HKDF(signPrivKey, "xchat-pda-seed-v1") | Identity PDA =
findProgramAddress(["xchat-id-v2", contactId (32 bytes)],
F3ydfNgdM89BK5hDh7amVmt8AAGQSYCX1afb3EWZKGh8) Archive PDA = findProgramAddress(["xchat-msg",
pdaSeed, contactId], F3ydfNgdM89BK5hDh7amVmt8AAGQSYCX1afb3EWZKGh8)
```

The contactId: Messaging Identity

The contactId is derived from the Ed25519 signing key via HKDF with a distinct label. What users share with each other is a contact bundle: { contactId, mlKemPubKey, signPubKey, relayDomain, keyHash }. The wallet address is never in this bundle. keyHash = SHA-256(mlKemPubKey || signPubKey) provides MITM protection when adding a new contact.

```
// Contact bundle shared between users: { contactId: base58(contactId), mlKemPubKey: base64,
signPubKey: base64, relayDomain: "kryptark.com", keyHash: base58(sha256(pk1||pk2)) } // No
wallet address. No pdaSeed. No on-chain link exposed.
```

On-Chain Registration

Registration is required for new users on X1 mainnet, program `F3ydfNgdM89BK5hDh7amVmt8AAGQSYCX1afb3EWZKGh8` ("xchat_pq_v2"). A prior testnet-era program, `EWNGaAyEVDWFGTa71qR54eJXRp6CoufuE4hfCkmKz2Jc`, is kept only as a legacy reference and is no longer written to. The wallet is the fee payer but is NOT stored in the PDA data. An observer doing getSignaturesForAddress on the wallet could find the anchor transaction, but the PDA data itself contains only contactId + key_hash. A separate, newer identity record ("identity v3", in the xchat-sovereign program) additionally binds a home relay domain and that relay's Ed25519 pubkey — see Section 16 for the trust chain this enables.

The relay's GET /api/identity/contact/:contactId endpoint returns keys without the wallet address. Even a compromised relay database reveals only contactId to keys, not wallet to keys.

6 Mobile and No-Extension Onboarding

A browser-extension wallet is not available on most phones, and it is friction even on desktop. Kryptark's soft wallet path generates a real Solana-format keypair inside the browser, funds it with just enough XNT to register on-chain, and lets the user protect it with a device passkey (Face ID, Touch ID, Windows Hello) instead of writing down a seed phrase for daily use. The cryptographic identity produced this way is identical to the extension-wallet path — same HKDF derivation, same on-chain anchor, same PQ guarantees.

6.1 Creating the Soft Wallet

1	**Generate a fresh keypair** seed = crypto.getRandomValues(32 bytes); keypair = Solana Keypair.fromSeed(seed). The seed, base58-encoded, is the entire recovery secret.
2	**Show and confirm the seed** UI displays the seed and wallet address; a confirmation checkbox must be ticked before "Create Account" is enabled.
3	**Fund the wallet (zero-balance bootstrap)** A brand-new keypair holds 0 XNT and cannot pay its own transaction fee. The client builds an unsigned claim_user_airdrop instruction (program stskNxdsPZ9ZgPYrfYaxZGT8325naiVKGfyM53oKTr2), partially signs it with the new keypair, and POSTs it to the relay.
4	**Relay co-signs as fee payer** POST /api/faucet/claim: the relay validates exactly 1 instruction targeting the faucet program, checks itself as feePayer, fetches a fresh blockhash, co-signs, submits, and confirms. Rate-limited to 3 claims per IP per day. After this the user has ~0.04 XNT.
5	**Derive identity and register** Same HKDF path as an extension wallet: sign "Kryptark Post-Quantum Identity" with the new keypair, derive ML-KEM-768 + Ed25519 keys, announce to the relay, then anchor on-chain — now self-funded.
6	**Offer passkey protection** Non-blocking: the user can protect the seed with a passkey immediately, or skip and rely on the recovery seed. Skipping does not block messaging.

6.2 Passkey Protection (WebAuthn PRF)

Where supported, the 32-byte seed is encrypted at rest using the output of the WebAuthn PRF (pseudo-random function) extension — a hardware-bound secret derived from the platform authenticator (Face ID, Touch ID, Windows Hello, or a security key). The plaintext seed is deleted from localStorage once passkey protection succeeds.

```
// One-time setup credential = navigator.credentials.create({ publicKey: { rp: { name:
"Kryptark", id: location.hostname }, authenticatorSelection: { residentKey: "required",
userVerification: "required" }, extensions: { prf: { eval: { first: fixedSalt } } }, ... } })
prfKey = credential.getClientExtensionResults().prf.results.first seedCiphertext =
AES-256-GCM.encrypt(prfKey, iv, seed) // stored: { credentialId, iv, ciphertext } — plaintext
seed removed // Every subsequent login assertion = navigator.credentials.get({ publicKey: {
allowCredentials: [{ id: credentialId }], extensions: { prf: { eval: { first: fixedSalt } } },
userVerification: "required" } }) prfKey =
assertion.getClientExtensionResults().prf.results.first seed = AES-256-GCM.decrypt(prfKey,
iv, seedCiphertext)
```

Because passkeys are a platform credential, they typically sync across a user's devices through the same ecosystem (iCloud Keychain, Google Password Manager) without the seed ciphertext itself ever leaving the device unencrypted. If the authenticator does not support the PRF extension, passkey protection is skipped gracefully and the user continues on the plaintext recovery seed.

6.3 Recovery and Portability

* **Recovery key:** the base58 seed can be copied at any time and re-imported on a new device or browser to fully restore the account — there is no other recovery path, exactly like a wallet mnemonic.
* **Export to a real wallet:** the full 64-byte Solana secret key can be exported and imported into Phantom or any Solana-compatible wallet, letting a user "graduate" a soft wallet into an extension wallet without changing identity.
* **Auto-reconnect:** a passkey-protected account shows a "Sign in with Face ID / Fingerprint" prompt on return visits; a plaintext-seed account resumes silently.
* **Mobile-reliable confirmation:** the soft-wallet anchor path (anchorIdentityWithKeypair) polls getSignatureStatus over HTTP instead of subscribing over WebSocket, because WebSocket subscriptions are unreliable on mobile browsers that suspend background connections.
* **Logout vs. full removal:** disconnecting keeps the encrypted/plaintext seed in place so the same soft wallet resumes on next visit; only an explicit account-removal path wipes the seed, passkey credential, and cached wallet address.

KEY PROPERTY

Soft-wallet identities are cryptographically identical to extension-wallet identities — same HKDF derivation, same on-chain PDA layout.

7 WebSocket Authentication: Ed25519 Challenge-Response

Earlier versions allowed any client knowing a victim's public contactId to connect to the relay as that contactId and receive their queued messages. The Ed25519 challenge-response protocol eliminates this attack: the server requires proof of possession of the Ed25519 signing private key before activating any WebSocket connection.

Authentication Flow

1	**Client connects** wss://.../ws?contactId=<contactIdB58> -- connection not yet active
2	**Server lookup** identityMap[contactId] --> signPubKey. Unknown contactId --> close 1008 "Unknown contactId"
3	**Challenge issued** { type: "auth_challenge", nonce: <32-byte random, base64url> } sent to client
4	**Client signs** sig = Ed25519.sign(b64decode(nonce), state.pqIdentity.signPrivKey) --> { type: "auth_response", sig }
5	**Server verifies** DER SPKI prefix (302a300506032b6570032100) + signPubKey bytes --> crypto.verify(null, pubKey, nonce, sig)
6	**Connection activated** auth_success --> state.relayWs set, updateConnectionStatus(true), offline queue flushed. auth_failure or 10s timeout --> 1008

Security Properties

* Relay never sees the wallet address during WebSocket authentication.
* Existing connections are only displaced after new connection fully authenticates -- prevents displacement DoS.
* WS onopen does NOT activate the connection: activation waits for auth_success message.
* 10-second timeout prevents hanging connections from consuming relay resources.
* Relay stores signPubKey in identityMap keyed by contactId: the two are never linked to a wallet.

KEY PROPERTY

Relay authenticates users by contactId + Ed25519 sig only. Wallet address is never seen.

8 Message Flow and Group Conversations

Sending a Message

1	**Key Exchange** ML_KEM_768.encapsulate(recipientMlKemPubKey) --> (kemCt: 1088 bytes, sharedSecret: 32 bytes). Fresh per message.
2	**Match Tag** matchTag = HMAC-SHA256(sharedSecret, "xchat-pq-v1-tag")[0:8] --> 8-byte routing hint.
3	**Sign Content** digest = SHA-256(content || timestamp_u64_be || recipientStealthId) sig = Ed25519.sign(digest, signPrivKey)
4	**Encrypt** msgKey = HKDF-SHA256(sharedSecret, "xchat-pq-v1-msg", 32) iv = randomBytes(12) blob = iv || AES-256-GCM.encrypt(msgKey, iv, payload_json)
5	**Relay** POST /api/relay { to: contactId, type: "pq_message", kemCt, matchTag, blob }
6	**Delivery / Federation** Relay looks up contactId --> relayDomain in identityMap. If relayDomain == RELAY_DOMAIN: deliver to local WebSocket or queue. If relayDomain is a remote relay: forward via xCFP (see Section 16). Relay never reads kemCt, matchTag, or blob.

Offline Queue

If the recipient is offline, the relay queues pq_message type blobs with a 7-day TTL, maximum 200 per contactId. The queue is persisted to data/queue.json so it survives relay restarts. The queue is flushed immediately when the recipient authenticates via WebSocket. Other message types (typing indicators, read receipts, call signals) are dropped when the recipient is offline.

Delivery Confirmation

A relay reporting "delivered" only means the recipient's socket looked open at send time, not that the recipient's client actually processed the message. After signature verification succeeds on the receiving side, the recipient sends a small plaintext acknowledgement back through the relay; the sender waits briefly for it and retries once if it does not arrive, then leaves the durable offline queue above to guarantee eventual delivery. A WebSocket heartbeat independently detects a silently-dropped connection (a network change with no clean disconnect) within seconds rather than leaving a dead connection marked as live indefinitely.

Group Conversations ("Arks")

A group conversation is not built on a shared encryption key. Instead, sending a group message performs the exact same ML-KEM encapsulation, signing, and AES-GCM encryption described above independently for each member -- N individually-encrypted envelopes reusing the identical 1:1 message

path, with the signature bound to the shared group identifier instead of a single recipient. The relay cannot tell a group fan-out apart from N independent 1:1 sends; there is no group-identifying metadata in what it sees on the wire.

This has a direct consequence for membership: removing someone from a group needs no re-keying, because there is no shared key to rotate -- they simply stop being included in future fan-out. The tradeoff is that this does not scale to very large channels the way a shared-key scheme would (message cost is O(N) per send), and a relay can observe timing correlation across a group's fan-out even though it cannot read any of it -- a disclosed, accepted limitation of this design choice rather than an oversight.

Joining requires an explicit invitation and an explicit accept, not automatic membership on first contact. Group voice and video calls use a mesh topology, reusing the same call-signaling and privacy-mode infrastructure as 1:1 calls (Section 14). Group files use a single symmetric key for the file itself (one IPFS upload instead of N), with that key embedded inside each member's already individually-encrypted message rather than transmitted separately.

9 IPFS Message Archive: Permanent Encrypted History

Kryptark stores conversation history in a cryptographically linked list of AES-256-GCM encrypted batches on IPFS. The latest CID is anchored on the X1 blockchain. The current archive encryption (v2) derives each party's key purely from their own identity key — no Diffie-Hellman or KEM exchange is involved, which removes an entire class of cross-party key-agreement bugs and is simpler to reason about under a quantum adversary model.

Archive Encryption (v2 — current)

```
// Each party derives their OWN key — asymmetric by design. // A never needs anything from B to encrypt/decrypt A's own batches. convKeyV2 = HKDF-SHA256(ikm=mySignPrivKey, salt=theirSignPubKey, info="xchat-archive-v2", len=32) // Each batch (JSON with prev_cid pointer) is AES-256-GCM encrypted batch = { v:1, from, contact, seq, prev_cid, archived_at, msgs:[...] } encrypted = AES-256-GCM.encrypt(convKeyV2, iv, JSON.stringify(batch))
```

A deprecated v1 scheme derived this key via X25519 ECDH between the two parties' converted Ed25519 keys. It is no longer written; any v1-encrypted archive segment still on IPFS simply fails AES-GCM authentication on restore and is silently skipped — it cannot be mistaken for valid history.

Archive Lifecycle

Trigger	IPFS Upload	On-Chain Anchor	Notes
Every 10 minutes	YES	NO	No wallet popup: silent background
After 50 new messages	YES	NO	BATCH_MSG_THRESHOLD guard
On clean logout	YES	YES	archiveAllPending(true): one wallet popup, only if there are new messages
On login (catch-up)	NO	YES if needed	anchorPendingCids(): handles force-quit; compares local seq vs on-chain seq

KEY PROPERTY

IPFS sees only ciphertext. No keys, no plaintext, no identity of uploader.

10 File Transfer: E2E Encrypted IPFS

File transfer has three format generations in the codebase. v3 is post-quantum and is the current default; v1/v2 remain readable only for backward compatibility with files sent before v3 shipped.

10.1 v3 (current) — ML-KEM-768

Every new file transfer encapsulates a fresh ML-KEM-768 shared secret to the recipient's post-quantum public key — the same primitive used for messages. Files at or below 5 MB upload as a single encrypted blob; larger files are split into 5 MB chunks so only one chunk is ever held in memory, with per-chunk keys derived from the same file key so no chunk reuses another chunk's keystream.

```
{ kemCt, sharedSecret } = ML_KEM_768.encapsulate(recipientMlKemPubKey) // 1088-byte ct fileKey = HKDF-SHA256(sharedSecret, info="x1-msg-v1-x1-file-v3", len=32) ct = AES-256-GCM.encrypt(fileKey, iv, fileBytes) cid = POST vault.x1.xyz/ipfs/api/v0/add?pin=true // Attachment in encrypted chat message (<=5 MB): { type:"xchat-file-v3", cid, name, mime, size, iv, kemCt } // >5 MB: { type:"xchat-file-v3", chunks:[{cid,iv},...], kemCt } // one KEM ciphertext covers every chunk; per-chunk keys derive from fileKey
```

10.2 v1 / v2 (legacy) — X25519 ECDH

Files sent before the v3 migration used the X25519 ECDH shared secret between the two parties' converted Ed25519 keys — classical, not quantum-resistant. v1 is a single-blob format (<=5 MB); v2 added chunking for larger files with the same key derivation. Both remain readable for backward compatibility only; nothing new is encrypted this way.

10.3 Personal Vault (Bookmarks) — Self-Encrypted

A separate "vault" feature lets a user store a file that only they can ever decrypt — no recipient, no shared secret. The key is derived purely from the owner's own private key, so even someone holding the CID cannot decrypt it without the owner's key. This uses the same X25519-derived key material as the legacy file format, but because it never leaves the owner's device, key exposure would require compromising the owner directly, not intercepting an exchange.

```
selfKey = HKDF-SHA256(ownPrivateKey, info="x1-bookmark-v1", len=32) ct = AES-256-GCM.encrypt(selfKey, iv, fileBytes) // { type:"xchat-bookmark-v1", cid, name, mime, size, iv } — owner-only, ever
```

Privacy Proxy for Link Previews

URLs in messages are fetched server-side via /api/og, preventing client IP leakage to third-party websites. OG images are proxied via /api/proxy. YouTube uses the oEmbed API (clean JSON) rather than HTML scraping. The SSRF guard (isPrivateUrl) re-validates on every redirect hop to prevent redirect-chain attacks. The proxy is limited to a maximum redirect depth of 5.

11 Decentralised Storage: Today and the Roadmap

HONEST STATUS

Every IPFS operation today is hardcoded to a single gateway, vault.x1.xyz. This is a real single point of failure.

Message archives, file transfers, the personal vault, and website publishing (Section 18) all currently upload to and fetch from the same IPFS gateway. If vault.x1.xyz becomes permanently unavailable, in-flight uploads fail and previously-uploaded content becomes unfetchable until the gateway returns or a client-side fallback is shipped. Content is still end-to-end encrypted regardless of which gateway serves it — decentralising the storage layer is about availability and censorship-resistance, not confidentiality, which is already guaranteed by the encryption in Sections 4, 9, and 10.

What Already Exists

* Relay CDN (server.js /api/content/:cid/*) already supports an optional local Kubo daemon via the IPFS_API_URL environment variable — not currently set on either production relay.
* The relay already gossips full website content between peers (site-announce) so a published site survives its home relay going offline — see Section 18.
* Every archive and file fetch enforces a hard byte cap (10 MB for archive batches, 50 MB for files) to prevent a malicious or corrupt CID from exhausting client memory.

Planned Fix, In Order

1	**Install Kubo on both relays** Each relay becomes a real IPFS node; set IPFS_API_URL=http://localhost:5001 in relay.env. This alone makes the relay a pinning node for website content and identity snapshots.
2	**Add POST /api/pin { cid }** After every client upload to vault.x1.xyz, fire-and-forget a re-pin request to the home relay so a second, independent copy exists immediately.
3	**Fallback gateway list on the client** Replace the single hardcoded vault.x1.xyz constant in xchat-archive.js and ipfs-upload.js with an ordered list (vault.x1.xyz, ipfs.io, dweb.link); race all on fetch, take first success.
4	**Surface the relay's own gateway to clients** Add ipfsGateway to GET /api/relay/info so relay operators can steer their users to their own IPFS node as the primary gateway.

Message Recovery Across Devices — a Related, Smaller Gap

Cross-device recovery already works cleanly after a normal logout: disconnecting anchors the latest archive CID on-chain, and a new device reads that on-chain pointer and follows the CID chain backward. The gap is a browser crash, force-quit, or cookie clear on a device that never logged out cleanly — localStorage held the pointer to the most recent unanchored upload, and clearing it loses that pointer even though the encrypted content is still sitting on IPFS.

The planned fix is a relay-held CID hint: after every IPFS upload the client POSTs contactId, cid, and seq to its home relay, signed with the same Ed25519 key already used for everything else (POST

/api/archive-hint). The relay cannot read message content from this — it only ever sees a routing identity it already knows and a content hash it cannot decrypt — but it can hand the pointer back at next login if local storage came back empty, closing the gap with no new wallet popups and no fee change.

NOT YET LIVE

Neither /api/pin nor /api/archive-hint exist in the deployed server today — both are specified and ready to implement.

12 Payments: Direct Transfers and Stealth Unlinkability

Kryptark supports two payment paths that trade off simplicity against unlinkability. Both are triggered from inside a conversation and both produce an encrypted, signed record embedded in the chat itself — a payment is always also a message.

12.1 Direct Payments

The everyday path: a /pay <amount> [memo] command parses into a plain SystemProgram.transfer from the sender's wallet directly to the recipient's wallet. This is fast and has no extra on-chain footprint, but — like any normal on-chain transfer — sender and recipient wallet addresses are linked publicly and permanently on X1. Agents can send the same transfer headlessly with a raw keypair, no popup, via the same code path used by browser wallets.

```
tx = SystemProgram.transfer({ fromPubkey, toPubkey, lamports }) // Browser wallet:
provider.signAndSendTransaction(tx) -- user-visible popup // Agent / headless:
tx.sign(keypair); sendRawTransaction(tx.serialize())
```

12.2 Stealth Payments (Optional, Unlinkable)

For payments where the parties want the on-chain link broken, Kryptark routes through a one-time stealth address. On-chain: Sender --> StealthAddress (Tx1), then StealthAddress --> RecipientWallet (Tx2). A blockchain observer sees two unrelated transactions. The recipient's real wallet never appears in Tx1.

Sender Flow

```
ephPriv = randomBytes(32) // ephemeral X25519 ephX25519Pub = X25519.getPublicKey(ephPriv)
recipX25519 = edwardsToMontgomeryPub(recipSignPubKey) ecdhSecret =
X25519.getSharedSecret(ephPriv, recipX25519) stealthSeed = HKDF-SHA256(ecdhSecret,
salt=ephX25519Pub, info="xchat-stealth-v1") stealthPubKey = Ed25519.getPublicKey(stealthSeed)
// one-time Solana address // Tx1: Sender --> stealthPubKey (recipient real wallet never in
Tx1) // conf.stealthEphPub = base64(ephX25519Pub) in encrypted message
```

Recipient Claim (No Wallet Popup)

```
ephX25519Pub = b64decode(conf.stealthEphPub) // 32 bytes, length validated ownX25519 =
edwardsToMontgomeryPriv(signPrivKey) ecdhSecret = X25519.getSharedSecret(ownX25519,
ephX25519Pub) stealthSeed = HKDF-SHA256(ecdhSecret, salt=ephX25519Pub,
info="xchat-stealth-v1") stealthKey = Keypair.fromSeed(stealthSeed) // function scope only //
Tx2: StealthAddress --> recipientRealWallet, signed by stealthKey // amount = balance -
actualFee: stealth account closed at exactly 0 // state.claimedStealthTxs (Set, persisted)
prevents double-claim
```

13 Per-Payment Encrypted Receipts

Each payment generates a cryptographic receipt with selective disclosure. The receipt key is deterministic and re-derivable at any time from the wallet signature. The wallet address never crosses the party boundary: receipts use conf.fromContactId, not conf.fromWallet.

```
receiptKey = HKDF-SHA256(signPrivKey, salt=utf8(txSig), info="xchat-payment-receipt-v1", len=32) receiptCt = AES-256-GCM.encrypt(receiptKey, iv, JSON.stringify(receiptObj)) receiptSig = Ed25519.sign(SHA-256(receiptObj), signPrivKey)
```

Mode	Generated by	Shows	Hides
Payment	Sender	Payer wallet + Tx1	Recipient identity
Receipt	Recipient	Recipient wallet + Tx2	Sender identity
Full	Either party	All known fields	Recipient wallet (sender view)

isSenderView uses conf.fromContactId (not conf.fromWallet): the wallet address never crosses the party boundary in a payment confirmation. Recipient generates a fresh signed attestation keyed to Tx2 -- they never decrypt the sender's AES-GCM blob.

14 Voice and Video Calls: WebRTC DTLS-SRTP

Real-time voice and video calls use WebRTC. The media path is peer-to-peer: the relay is involved only in SDP/ICE signaling, never in the media stream. Call signaling is Ed25519-signed by the initiator and sent through the same authenticated channel as messages, so only authenticated users can initiate or answer calls, and the relay cannot forge a call invitation on someone's behalf.

* **DTLS for key exchange:** Peers negotiate a session key directly. No server involvement in media.
* **SRTP for media:** All audio/video encrypted by WebRTC natively.
* **STUN/ICE:** A relay operator can run a self-hosted STUN/TURN server (coturn); the client's default "balanced" mode prefers it over any third-party STUN, falling back to Google's public STUN only when a relay has none configured. TURN also covers symmetric-NAT connections that STUN alone cannot traverse.
* **Signaling privacy:** SDP/ICE forwarded by relay (plaintext) but contains no message content. A "relay" call-privacy mode routes media itself through the self-hosted TURN server so neither peer learns the other's IP; the default "balanced" mode exchanges direct peer candidates for lower latency, which does reveal each peer's IP to the other party (not to the relay or STUN operator) in the way WebRTC calls ordinarily do.
* **Call controls:** Mute, video toggle, mute speaker, 60-second unanswered timeout, minimize-to-corner during an active call.
* **Group calls:** Mesh topology -- every participant connects directly to every other participant, suited to small groups; a media-relay (SFU) architecture would scale further but is not built.
* **Agent limitation:** Voice/video requires a browser media stack. AI agents cannot participate in WebRTC calls.

15 Server Architecture: The Thin Relay

server.js is intentionally minimal: a single Node.js file using Express and ws. Its only job is to route opaque byte sequences. It holds no message history, no private keys, and no plaintext of any kind. The production relay is managed by pm2, on HTTPS_PORT=9443 and PORT=4000.

Data File / Type	Stored?	Duration	Notes
data/identities.json	YES	Persistent	contactId -> keys + relayDomain + x1nsName. No wallet.
data/names.json	YES	Persistent	X1NS name -> contactId aliases (Section 17).
data/queue.json	YES	7 days, 200/key	pq_message blobs only. Persisted to disk.
data/relay-identity.json	YES	Permanent	Relay Ed25519 keypair. Gitignored.
data/relay-peers.json	YES	Persistent	Gossip peer list from /api/relay/peers.
data/agents.json	YES	Persistent	Agent declarations. Cap: 10000.
data/cache/	YES	7 days	IPFS CDN cache. Cap: 10 GB.
Active WebSockets	In-memory	Until disconnect	contactId to WS handle. Cleared on disconnect.
Message history	NO	N/A	No database. Never persisted.
Private keys	NO	N/A	Never sent to server by design.

Security Hardening

* **CORS:** Restricted to explicit ALLOWED_ORIGINS set. No wildcard.
* **CSP:** Content-Security-Policy with default-src none and explicit allowlists for each source type.
* **SSRF:** isPrivateUrl() covers RFC-1918, 169.254.x.x (cloud metadata), IPv4-mapped IPv6, hex/decimal loopback. Re-validated on every redirect hop.
* **IP:** Rate limiting uses req.socket.remoteAddress only. X-Forwarded-For never trusted.
* **Relay frame injection:** conversation_cleared and message_deleted frames dropped client-side.
* **Key validation:** /api/identity validates mlKemPubKey length (1400-1700 chars) and signPubKey length (40-50 chars).
* **Key-update guard:** If existing signPubKey differs from submitted, registration is rejected.
* **Faucet guard:** /api/faucet/claim rejects any claim tx with more than one instruction or a fee payer other than the relay itself, so a malicious client cannot smuggle a fund-draining instruction into the relay's co-signature.
* **PUBLIC_URL:** Set to canonical HTTPS URL when relay is behind a reverse proxy. X-Forwarded-Host is NOT trusted anywhere.

16 Federation: The xCFP Relay Network

The xCFP (xchat Federation Protocol) is analogous to SMTP for email: each relay operator runs an independent relay, and messages route between them transparently. No single relay operator can censor or shut down the network. A user registered on one relay can message a user registered on any other relay running the same protocol, with no coordination beyond both relays knowing how to reach each other.

How xCFP Message Routing Works

1	**Identity registration stamps relayDomain** POST /api/identity stores { contactId, mlKemPubKey, signPubKey, relayDomain } for new users. relayDomain = the relay where the user registered. Never overwritten for existing entries.
2	**Sender relay looks up recipient relayDomain** POST /api/relay { to: contactId, ... } --> relay checks identityMap[contactId].relayDomain. If different from own RELAY_DOMAIN: initiate federation delivery.
3	**Challenge-response per inbound request** Sender relay: GET https://recipRelay/api/relay/challenge --> { nonce: base64 } Sender relay signs: Ed25519.sign(nonce || blobHash || senderPubKey || recipientContactId || hop || timestamp, relaySignPrivKey)
4	**Forwarded delivery** POST https://recipRelay/api/relay/inbound { envelope: sig, blob } Recipient relay verifies Ed25519 sig, then delivers to recipient WebSocket or queues locally.
5	**Routing fallback** If homeRelay == RELAY_DOMAIN but recipient not connected: fresh federation lookup before queuing. If forwarding fails: message queued locally; recipient recovers via IPFS archive.

Gossip Protocol

Relays discover peers via gossip. POST /api/relay/announce lets relays introduce themselves to peers. GET /api/relay/peers returns the known peer list. After first contact the peer list grows dynamically. The KNOWN_RELAYS / BOOTSTRAP_RELAYS environment variable seeds the initial peer list as bare domain:port (no https:// prefix). In production, INBOUND_RATE_LIMIT is raised from its 100/min default to 6000/min — the default is too low for real cross-relay traffic once federation is actually carrying messages.

Trust Tiers

Trust Level	How It Works	Config
Explicit trust	TRUSTED_RELAY_DOMAINS env: allowlist. Unknown relays rejected before any network call.	TRUSTED_RELAY_DOMAINS=relay1.x1,relay2.x1
On-chain trust	Identity v3 PDA (xchat-sovereign program) can bind a home relay's Ed25519 pubkey to a contactId. Clients compare it against the live relay's /api/relay/info pubkey to detect DNS hijacking or domain spoofing.	Automatic if the identity was registered with a relay pubkey binding
TOFU	First-contact: accept any relay and cache its pubkey. Subsequent contacts must match cached pubkey.	No config: default when TRUSTED_RELAY_DOMAINS unset

Reverse Proxy Requirement

Relays behind a reverse proxy must proxy BOTH /api/ AND /.well-known/ to the relay backend. Proxying only /.well-known/ leaves federation key lookups and inbound message delivery broken. Set PUBLIC_URL=https://canonical.domain in relay.env to control the URL shown in /.well-known/xchat-agent.json. Inter-relay TLS uses rejectUnauthorized: false to accept self-signed certificates. The from_relay field accepts host:port format (e.g. 203.0.113.10:9443).

On-Chain Relay Registration

Relay operators register on-chain via tools/xchat-relay-register.js. PDA seeds: ["xchat-relay-v1-pk", relay_pubkey[32]]. The fee_payer is NOT stored in the PDA -- the operator wallet is never linked to the relay identity on-chain. relay_signer co-signs to prove key ownership.

```
node tools/xchat-relay-register.js \ --relay-identity data/relay-identity.json \ --fee-payer wallet.json \ --domain relay.x1 // PDA: findProgramAddress(["xchat-relay-v1-pk", relayPubkey[32]], // DdyU28aFeSqUgiD3bLdck7Gzv7AvqpdgT9Yw1doiwWH8)
```

17 X1NS: Human-Readable Names for contactIds

DISAMBIGUATION

This section covers the OFFICIAL X1NS (SPL Name Service, mainnet). Section 18 covers a separate, xchat-specific name+content registry used only for website hosting.

A contactId is a 32-byte value, base58-encoded — accurate, but not something a person reads over a phone call. Kryptark lets any user link a name they already own on the official on-chain X1NS (SPL Name Service, deployed on X1 mainnet — program `nameQyUhZQQgnirGbbJRR9ECWSdq1W7mMaNUZZTBvtq`, registrar `X1NS1M4Lh9zwpYe7Mi2RKyy7QWq7dRtqyr7KxsLyUn5`, with .x1, .xnt, and .xen top-level domains) to their contactId, so contacts can be added by name — `alice.x1` — instead of a base58 string.

Claiming a Name

POST /api/identity/claim-name links a domain to a contactId only after three independent checks all pass:

1	**Proof of xchat identity** xchatSig = Ed25519.sign(claimMsg, signPrivKey) — proves the caller controls the signing key for the contactId being linked.
2	**Proof of wallet control** walletSig = Ed25519.sign(claimMsg, walletPrivKey) — proves the caller controls the wallet that owns (or claims to own) the X1NS domain.
3	**On-chain ownership check** The relay independently queries the X1NS API for the domain's current registered owner and rejects the claim if the submitted wallet does not match.

```
claimMsg = "kryptark-name-claim:v1:{name}:{contactId}:{timestamp}" // +/-5 min window // All three checks pass --> namesMap[name] = { contactId, claimedAt } // The wallet address itself is never stored -- only used transiently to verify.
```

Resolving a Name

GET /api/identity/resolve-name/:name returns the same shape as a direct contactId lookup. If the name is not claimed locally, the relay fans out to its actively-gossiping federation peers before giving up — names are relay-local state, not gossiped automatically, so a name claimed on one relay is discoverable from any relay in the network via this fallback.

Name-Sync Gossip

After a successful local claim, the owning relay proactively gossips the new name to its peers (POST /api/relay/name-sync) so future lookups resolve in one hop instead of a fan-out. Locally-claimed entries always take precedence and are never overwritten by gossiped data. Every peer-sourced entry is marked source: "remote" in the API response, which signals the client to independently verify the returned keys against the on-chain identity PDA before trusting them for anything sensitive — the gossip channel accelerates discovery, it is not itself a trust root.

Also available: GET /api/x1ns/resolve/:name and /api/x1ns/available/:name proxy the official X1NS API through the relay so a client's IP is never exposed directly to a third-party X1NS resolver.

INTEGRITY

from_relay on /api/relay/name-sync is Ed25519-signed and verified against the sending relay's on-chain-registered pubkey, the same envelope scheme as inbound message federation. See Section 21.

18 X1 Sovereign Stack: Four-Layer Decentralised Internet

The X1 Sovereign Stack is the most ambitious component built on top of Kryptark's messaging core. It replicates the four foundational layers of the traditional internet -- SMTP, DNS, web hosting, and browser DNS -- as a fully decentralised stack anchored on the X1 blockchain. Each layer independently removes a centralised chokepoint. Layer 1 here is an xchat-specific name-and-content registry deployed on X1 **mainnet** — distinct from the official X1NS covered in Section 17, which resolves names for contactId aliasing only and has no concept of hosted content.

Layer	Analogue	Replaces	Key Files	Status
0 -- xCFP Federation	SMTP (email relay)	Email servers / centralised relay operators	server.js /api/relay/inbound /api/relay/challenge	Live, mainnet
1 -- xchat-sovereign name+content registry	DNS (name system)	Domain registrars + DNS infrastructure, for hosted xchat sites	xchat-sovereign program DdyU28...doiwWH8	Live, mainnet
2 -- xchat-publish CLI	Web host	Web hosting companies + CDN providers	tools/xchat-publish.js site/ directory	Complete
3 -- Chrome MV3 Extension	Browser DNS resolver	OS DNS resolver / registrar lock-in	x1-extension/ + x1-bootstrap/ resolver.js	Complete

Layer 0: xCFP Federation (SMTP Analogue)

xCFP is the relay-to-relay message delivery protocol described in Section 16. It replaces centralised email servers with a federated network of independently operated xchat relays. Each relay announces itself via gossip, relays look up recipient relayDomain from identityMap, and forward messages using Ed25519-signed envelopes with a per-request challenge-response. No single operator controls message routing.

Layer 1: The xchat-sovereign Name + Content Registry (DNS Analogue)

Separately from the official X1NS in Section 17, the xchat-sovereign Anchor program (DdyU28aFeSqUgiD3bLdck7Gzv7AvqpdgT9Yw1doiwWH8, X1 **mainnet**) maintains its own name registry purpose-built for hosting: each name resolves to a { CID, relayDomain } record for website content, not just a contactId.

Address format: contactId@relay.domain.x1 encodes both the messaging identity and the home relay in a single resolvable address.

Layer 2: xchat-publish CLI (Web Host Analogue)

tools/xchat-publish.js allows any user to publish a website to the X1 sovereign network in a single command. The CLI: (1) scans the directory recursively and hashes all files; (2) builds a JSON manifest with filename, content-type, and SHA-256 hash for each file; (3) signs the manifest with the user's Ed25519 key; (4) uploads all files to vault.x1.xyz IPFS (pin=true); (5) anchors the root CID on-chain via the xchat-sovereign program; (6) prints the DNS TXT record for traditional DNS bridge to the .x1 name.

```
# Publish a directory to the xchat-sovereign name registry node tools/xchat-publish.js
./mysite --name alice.x1 --wallet wallet.json # Output: IPFS CID, on-chain tx signature, DNS
```

```
TXT record # Serve: https://relay.x1.xyz/api/content/{cid}/index.html # TXT record:
_xchat.alice.x1 "cid=Qm... relay=kryptark.com"
```

Layer 3: Chrome MV3 Extension (Browser DNS Analogue)

The x1-extension/ Chrome MV3 extension intercepts all HTTP/S requests to *.x1 domains using declarativeNetRequest. The resolver pipeline is:

* **Step 1:** Query the xchat-sovereign PDA for the requested name.
* **Step 2:** Extract CID and relayDomain from the PDA data.
* **Step 3:** Fetch from relayDomain's /api/content/{cid}/ (relay CDN, local relay, highest trust).
* **Step 4:** Fallback to KNOWN_RELAYS list if home relay is unreachable.
* **Step 5:** Final fallback to IPFS gateway.
* **Verification:** SHA-256 of every file verified client-side against the manifest, regardless of source.
* **Rendering:** Content rendered in a sandboxed iframe to prevent XSS from untrusted site content.

Bootstrap Resolver: x1-bootstrap/

x1-bootstrap/ is an HTTPS resolver for users who do not have the Chrome extension. It resolves names to CIDs, verifies the manifest, and renders the site in a sandboxed iframe. A service worker caches resolved content for offline access. This provides the same resolution guarantee without requiring browser extension installation.

Website Pinning Across Relays

When a user publishes a website via xchat-publish or the Web Launcher UI, the home relay broadcasts a site-announce message to all peers via gossip. Each peer relay fetches and caches the full site content to local disk (10 GB cap, 7-day TTL). Sites are served at /api/content/{cid}/index.html with correct MIME types. This means a published website survives the home relay going offline -- any peer relay can serve it from its cache. The SHA-256 manifest ensures content integrity regardless of which relay serves the files.

SIGNIFICANCE

Layer 3 completes a fully sovereign internet for hosted content: no ICANN, no AWS, no registrar. Owned by your key.

19 AI Agent Protocol

Kryptark treats AI agents as first-class participants rather than a bolt-on bot API. There are two layers. The first, covered in 19.1, is a lightweight bootstrap protocol any agent can use to onboard with nothing but an Ed25519 keypair -- send and receive messages, make payments, publish content. The second, covered from 19.2 onward, is a self-custodial in-group agent: a deployable participant with its own post-quantum identity and its own on-chain wallet, that can join conversations and groups, hold funds an owner explicitly authorizes it to spend, and execute trades under an owner-signed, runtime-enforced policy -- with no custody of anyone's wallet key ever passed to it, and the underlying LLM never given raw key material at any point.

19.1 Machine-Readable Bootstrap: /.well-known/xchat-agent.json

Every relay exposes a machine-readable manifest at /.well-known/xchat-agent.json (5-minute cache). The manifest provides everything an agent needs to onboard: relay endpoints, WebSocket URL, npm package list, airdrop program address, and step-by-step onboarding instructions. The URL is constructed from PUBLIC_URL env var or req.headers.host -- X-Forwarded-Host is never used (attacker-controlled).

```
GET /.well-known/xchat-agent.json { "relay": "https://kryptark.com", "websocket":
"wss://kryptark.com/ws", "api": { "identity": "/api/identity", "relay": "/api/relay",
"agents": "/api/agents" }, "airdrop_program": "9zkypzFPQ2s3D5UqbYuixt3iXo5ig3ZNWLK1TrbNf5eR",
"npm_packages": ["@noble/post-quantum", "@noble/curves", "@noble/hashes", "@noble/ciphers",
"@solana/web3.js"], "onboarding_steps": [ /* step-by-step agent onboarding */ ] }
```

Agent Onboarding Flow

1	**Fetch manifest** GET /.well-known/xchat-agent.json -- obtain relay base URL, WebSocket URL, airdrop program address
2	**Generate Ed25519 keypair** EVM/secp256k1 agents: generate fresh Ed25519 keypair (no chain connection required). Solana agents: use existing wallet or derive from HKDF like human users.
3	**Request airdrop** requestAgentAirdrop(keypair, rpcUrl) in xchat-pq-chain.js -- calls program 9zkypzF... on X1 mainnet. Anchor discriminator: sha256("global:airdrop")[0:8]. Requires a tiny seed amount of XNT already held.
4	**Derive identity** Same HKDF derivation as human users: walletSig -> ML-KEM keypair + Ed25519 keypair -> contactId.
5	**Register identity** POST /api/identity { contactId, mlKemPubKey, signPubKey, sig } -- Ed25519 sig over "contactId:mlKemPubKey:signPubKey"
6	**Declare as agent** declareAgent(apiBase, contactId, signPrivKey, { type, description, capabilities }) in xchat-pq-chain.js POST /api/agents/register with Ed25519 sig over "agent-declare:<contactId>:<type>"
7	**Connect WebSocket** wss://.../ws?contactId=<cid> -- complete Ed25519 challenge-response (Section 7)
8	**Send/receive messages** Full PQ E2E encryption: ML-KEM-768 + AES-256-GCM + Ed25519. Same as human users.

Agent Registry

Endpoint	Method	Auth	Notes
POST /api/agents/register	POST	Ed25519 sig	Sig over "agent-declare:<cId>:<type>". Rate: 5/min.
DELETE /api/agents/register	DELETE	Ed25519 sig	Sig over "agent-undeclare:<cId>". Removes declaration.
GET /api/agents	GET	None	List agents. ?type=&limit=&offset=. Rate: 30/min.
GET /api/agents/:contactId	GET	None	Look up specific agent by contactId.

Agent Types and Capabilities

Capability	Supported	Notes
PQ E2E messaging	YES	Full ML-KEM-768 + AES-256-GCM + Ed25519
Send XNT payments	YES	state.walletKeypair alternative to walletProvider. Direct or stealth, no popup.
Stealth payment claim	YES	confirmPayment() supports walletKeypair. No browser wallet needed.
IPFS file upload	YES	ipfs-upload.js: E2E encrypt (v3, ML-KEM-768) -> vault.x1.xyz -> embed CID in message
IPFS file download	YES	fetchAndDecryptV3(): fetch CID -> decapsulate -> decrypt
Website publishing	YES	xchat-publish CLI: scan dir -> hash -> sign -> IPFS -> on-chain anchor
X1NS name claim/resolve	YES	Same claim-name / resolve-name flow as human users (Section 17)
Vault access	YES	Owner-only HKDF(privateKey, "x1-bookmark-v1") encryption
Voice/video calls	NO	WebRTC requires browser media stack. Hard technical wall for agents.
Agent types	--	assistant, translator, notifier, bridge, oracle, moderator, custom

AIRDROP

Program 9zkypzFPQ2s3D5UqbYuixt3iXo5ig3ZNWLK1TrbNf5eR on X1 mainnet funds new agent accounts.

19.2 Self-Custodial In-Group Agents

An in-group agent is deployed either as a Node.js process (always-on, the full toolbelt) or directly in a user's browser tab (zero-install, runs only while the tab is open). Both derive their entire identity -- ML-KEM keypair, Ed25519 signing keypair, contactId, and a separate on-chain wallet -- from one seed via the same HKDF scheme human identities use (Section 4), so the exact same agent, memory and all, can move from a browser tab to a always-on host by copying that one seed. The relay never holds this seed and cannot read the agent's conversations or memory: the agent decapsulates messages and encrypts its own memory client-side, in its own process, exactly like a human client does.

Encrypted memory (recent turns, a rolling summary, durable facts, and per-contact/per-group profiles) is layered so it stays useful across long conversations without growing unbounded, and is AES-256-GCM-encrypted under a seed-derived key before ever leaving the process -- an IPFS snapshot of that memory is ciphertext, not plaintext, to anyone but the agent.

19.3 Owner Authority and Per-Member Spend Permissions

Every agent has exactly one owner -- the contactId that created it -- who has absolute, uncapped control over the agent's wallet, and who alone may grant other group members a bounded spending allowance (a per-transaction cap and a per-day cap, tracked and reset per UTC day). Authority is never established from anything the model reads or infers from a conversation: every privileged action is dispatched with the Ed25519-verified sender of the specific message that requested it, and the tool that performs the action re-checks that identity itself rather than trusting that the action was merely offered to the right person. A message claiming to be the owner, from someone who is not, is refused at the signature layer before any tool runs -- text cannot forge cryptographic identity.

19.4 Trading Policy Engine

Separately from who is allowed to spend, the owner can set a single trading policy that binds every spend and every trade the agent makes -- including the owner's own -- as a fat-finger guard, not an authority mechanism. Structured fields (maximum slippage, maximum price impact, maximum position size as a percentage of the agent's balance, a per-transaction cap, a mint allow/deny list, and a confirm-above-this-amount threshold) are enforced in code before anything is signed; free-text rules are advisory context for the model, never sold as a hard guarantee. An action above the confirm threshold is held rather than executed, and is released only when the OWNER's own next signed message contains the word "confirm" and is not itself a negation ("don't confirm") -- the model cannot fabricate that confirmation on the owner's behalf.

19.5 On-Chain Trading and Read-Only Bridge Monitoring

With signing explicitly enabled, an agent can execute swaps on XDEX (a constant-product AMM on X1) from its own wallet, including a two-hop route through native XNT when no direct pool exists, with slippage and price-impact guards applied before every signature. X1↔Solana bridge activity can be monitored and decoded for reporting, but the agent has no write/deposit capability there: the bridge's outbound-deposit account layout could not be independently verified from published transactions, and Kryptark does not sign transactions against an interface it cannot verify. The same standard applies to a separate, unverified swap-routing program noticed on-chain -- it is not wired for signing until its instruction format can be confirmed.

19.6 Scheduled Actions

The always-on Node runtime supports owner-only recurring scheduled actions (for example, a periodic DCA swap), limited to the same tool set a live chat message could trigger and dispatched through the identical permission and policy checks on every run -- a schedule is not a separate, weaker execution path. A scheduled run that would need above-threshold confirmation is refused rather than silently held forever, since there is no one watching a background tick to type "confirm."

19.7 Durable, Portable Memory

Memory persists locally by default and can optionally be snapshotted to IPFS as ciphertext for portability across hosts. Anchoring the latest snapshot pointer on-chain (so a new host with nothing local can restore memory purely from the chain) is available but off by default, for the same reason on-chain archive anchoring is off by default for human users -- see Section 21.

19.8 Browser vs. Node Runtime

	Browser agent	Node agent
Install	None -- runs in a tab	npm install; a guided setup wizard writes the config and can start it under pm2
Uptime	Only while the tab is open	Always-on
Best for	Trying an agent, lightweight reply duty	Trading, scheduled actions, anything that must survive a closed browser
Toolset	Chat, wallet reads, XDEX/bridge reads, signing tools when enabled	Everything the browser agent has, plus scheduling, sandboxed local code execution, and MCP tool servers
Identity	Same seed-derivation scheme -- portable to the Node runtime by copying the seed	Same

SCOPE

The relay is never a viable place to host an agent -- it would mean the operator holding the agent's seed and being able to read its conversations. Both supported runtimes keep the seed with the agent's own process.

20 Privacy Model: What Leaks and What Does Not

Observer	Can See	Cannot See
Relay Server	contactId (routing + WS auth) Timestamp, online status IP address of connection	Message content File content Payment details Wallet address (WS auth is contactId-based)
X1 Blockchain (identity anchor, one-time)	This wallet funded this contactId's identity anchor (fee payer is public on any transaction) key_hash (a hash, not the keys themselves)	Any message or file Which other contactIds this identity talks to Payment parties (stealth path)
X1 Blockchain (archive/memory anchor -- opt-in, off by default)	If enabled: this wallet anchors pdaSeed-based archive pointers, correlatable across every contact this identity has ever anchored	Message or file content (still just an encrypted CID pointer) Anything at all, if left off (Section 21)
IPFS (vault.x1.xyz)	Ciphertext (random bytes) CID (content hash)	File or archive content Decryption key Uploader identity
Payment Observer (blockchain, stealth path)	StealthAddress (Tx1 recipient) Stealth to Recipient wallet (Tx2)	Sender real identity Recipient in Tx1 Payment memo (in E2E msg)
Payment Observer (blockchain, direct path)	Sender wallet, recipient wallet, amount (like any transfer)	Message content, memo (transmitted only inside the E2E channel)
Federation Peer (relay-to-relay)	Sender relayDomain Recipient contactId Encrypted blob (ciphertext)	Message plaintext Sender or recipient wallet Content type or file content
STUN/TURN server (voice/video setup only)	Both parties' public IP addresses during call setup, in the default "balanced" mode	Call audio/video (DTLS-SRTP) Who the call is between Either party's IP, in "relay" call-privacy mode (media routed through TURN)

REMAINING LEAKS

Relay sees client IP on connection. Federation peers see sender relayDomain and recipient contactId. Default-mode calls exchange peer IPs directly (avoidable with call-privacy "relay" mode). On-chain archive/memory anchoring, if a user opts into it, links the anchoring wallet across that identity's conversations (Section 21) -- off by default.

21 Challenges in Building a Sovereign Privacy Stack

Unlinking the Fee Payer from Identity

The hardest problem in the design was making the wallet address -- which must pay on-chain fees -- invisible in the identity system. Standard Solana programs use the wallet public key in PDA seeds, directly linking the on-chain account to the wallet. Kryptark breaks this with independent HKDF derivations from the same signing key: one produces the public contactId that is shared with contacts, another produces a private pdaSeed used only to locate the user's own archive PDAs. For the identity anchor and every routing/message operation, nothing about the contactId, the messages, or the relay traffic requires revealing which wallet is behind an identity.

On-chain archive and agent-memory anchoring (Sections 9 and 19) is the one place a PDA-locating seed has to travel inside an on-chain transaction's own data for the program to find the right account -- which is why, unlike every other feature in this stack, it is an **explicit opt-in, off by default**. The default path -- periodic encrypted IPFS archiving plus the relay-backed archive-hint fallback (Section 11) -- gives the same cross-device message recovery without any on-chain transaction at all. Anchoring on top of that is available for anyone who wants the extra durability of a permanent on-chain pointer and is comfortable with that specific transaction being linked to the wallet that pays it; it is not required for normal use.

Per-Message Key Freshness with a Post-Quantum KEM

ML-KEM ciphertext is 1088 bytes per message versus 64 bytes for ECDH-P256 -- a 17x overhead, accepted for genuine per-message ciphertext unlinkability: no shared secret is ever reused across two messages, and every message is independently post-quantum resistant. Each message encapsulates against the recipient's public key with a fresh ephemeral operation rather than reusing a session-level secret. A full ratchet construction (as in Signal's Double Ratchet) -- rotating the underlying key material itself so that a future key compromise cannot reach past messages -- is a valuable, commonly-cited property that not every secure messenger implements, and is on Kryptark's roadmap as a post-quantum ratchet (Section 25) rather than something this design currently claims to provide.

Stealth Payments: The Two-Signature Problem

Making a blockchain payment without linking sender and recipient requires the recipient to control a one-time address derivable from public information shared by the sender, without that information being linkable to either party by a chain observer. The solution is X25519 ECDH with an ephemeral key: the sender generates a fresh ephemeral X25519 keypair, computes ECDH with the recipient's converted Ed25519 key, and embeds only the ephemeral public key in the encrypted message. The recipient derives the stealth seed from their own signing key and the ephemeral public. A chain observer sees two unrelated transactions with no connection between them.

IPFS Archive: Force-Quit CID Orphaning

Uploading to IPFS is cheap (no wallet popup). Anchoring the CID on-chain requires a wallet transaction, which was triggering disruptive popups every 10 minutes. The fix was an anchorOnChain flag: periodic archives are IPFS-only. Logout triggers one wallet popup for all pending anchors. The remaining problem: a browser crash or force-quit leaves IPFS batches uploaded but never anchored on-chain. Solution: anchorPendingCids() at login reads local archive state, checks on-chain seq numbers, and anchors any batches where local seq exceeds on-chain seq. The one case this cannot cover — localStorage itself being cleared before that next login — is addressed by the relay-hint plan in Section 11.

Bootstrapping a Wallet That Cannot Yet Pay

A soft wallet is generated client-side with a zero balance, but every on-chain action (the identity anchor itself) costs a fee. The relay cannot simply hand out XNT to any caller, or it becomes a free money faucet. The solution constrains the co-signed transaction tightly: exactly one instruction, targeting exactly the faucet program, with the relay itself as fee payer — so the worst a malicious client can do is waste the relay's own compute, not drain funds, capped further by a 3-claims-per-IP-per-day limit.

Rendering Untrusted Content Safely

Every message, contact name, file name, link preview, and payment memo a user sees originates from another party and must be rendered as HTML without ever being interpreted as markup or script. Kryptark's approach is a small number of shared, consistently-applied primitives rather than ad hoc escaping at each call site: a single character-map HTML-escaping function used for all text content and attribute values, a URL allowlist (safeUrl()) restricting links to http/https, and a strict rule that any value placed inside an inline event handler goes through a data-* attribute read at runtime rather than being interpolated into the handler's own source text. A Content-Security-Policy header adds a second layer on top. This is the kind of surface that benefits from periodic re-review as new rendering features (reply quotes, avatars, agent management panels) are added, since each new render site is a new place the same primitives need to be applied correctly.

Wallet Address in Payment Receipts

Payment receipts need to prove the payment happened, which means including on-chain transaction signatures. But the receipt also knows both parties' wallet addresses from those transactions. If shared, a receipt links both parties' wallets. Solution: three-mode selective disclosure -- Proof of Payment shows only the sender's wallet and Tx1 (stealth address); Proof of Receipt shows only the recipient's wallet and Tx2; Full Disclosure is generated only voluntarily. Additionally, conf.fromWallet was replaced with conf.fromContactId: the wallet address no longer crosses the party boundary in the encrypted payment confirmation message.

Federation Routing and relayDomain Integrity

Federation introduces a new routing layer with its own attack surface. A compromised relay could claim any relayDomain in its identity registrations, redirecting traffic. Layered mitigations: (1) relayDomain is only stamped on brand-new registrations -- existing entries are never overwritten, preventing re-registration hijacking; (2) TRUSTED_RELAY_DOMAINS env var gives operators an explicit allowlist; (3) the identity v3 on-chain relay-pubkey binding (Section 16) lets a client verify the relay it is talking to matches what the contactId owner registered. The xCFP per-request Ed25519 challenge-response additionally binds each forwarded message to a specific relay keypair, preventing replay attacks and message injection from unknown relays.

X1NS Name-Sync Gossip Integrity

The name-sync gossip endpoint (Section 17) that propagates newly-claimed names between relays uses the same nonce-challenge plus Ed25519-signed-envelope scheme as inbound message federation (Section 16), with the signing relay's pubkey checked against its on-chain registration -- a peer cannot forge a from_relay it doesn't control. As further defence in depth, locally-claimed names can never be overwritten by gossiped data regardless, and every peer-sourced record is tagged so the client independently re-verifies the keys against the on-chain identity PDA before trusting them for messaging or payments -- gossip accelerates discovery, it is never itself the trust root.

22 Comparison with Other Messaging Platforms

vs. Signal

Signal is the gold standard for consumer E2E messaging. However: requires a phone number; uses ECDH (not fully post-quantum, partial PQXDH announced); relies on centralised servers; has no blockchain identity, no federation, and no payments. Its per-session Double Ratchet is powerful but the KEM step is still breakable by a sufficiently large quantum computer.

vs. Telegram

Telegram's default conversations are server-encrypted (Telegram holds keys). Secret Chats are E2E but require a phone number, are not per-message forward-secret, and store metadata on Telegram servers. No PQC, no blockchain identity, no payments, no federation.

vs. Matrix / Element

Matrix is the closest philosophical cousin: federated, open, E2E via Megolm ratchet. Federation leaks metadata across homeservers, accounts require server registration, and there is no PQC, no blockchain identity, no stealth payments. Matrix federation uses DNS (centralised infrastructure); xCFP federation uses on-chain relay registry and Ed25519-signed envelopes with no centralised name resolution required to route.

Feature	Signal	Telegram	WhatsApp	Matrix	Kryptark
E2E encryption default	YES	NO	YES	YES	**YES**
Post-quantum crypto	Partial	NO	NO	NO	**YES Full**
Forward secrecy (ratchet)	YES	NO	YES	YES	**Roadmap**
No phone/email required	NO	NO	NO	NO	**YES**
Wallet-unlinkable identity	NO	NO	NO	NO	**YES contactId**
No-extension onboarding	NO	NO	NO	NO	**YES soft wallet + passkey**
No server key storage	NO	NO	NO	NO	**YES**
Stealth payments	NO	NO	NO	NO	**YES optional**
Encrypted history archive	NO	NO	NO	NO	**YES IPFS**
Federation	NO	NO	NO	YES	**YES xCFP**
On-chain identity anchor	NO	NO	NO	NO	**YES**
Human-readable identity alias	NO	NO	NO	NO	**YES X1NS**
Self-custodial AI agents	NO	Bots (3rd-party, custodial)	NO	NO	**YES, own wallet + policy**
Link preview IP privacy	NO	NO	NO	NO	**YES proxy**

23 Quantum Threat Landscape and Sovereign Identity Implications

In August 2024, NIST finalised the first three post-quantum cryptographic standards: FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), and FIPS 205 (SLH-DSA). These standards exist because NIST, NSA, and CISA collectively assess that quantum computers capable of breaking RSA-2048 and ECDH-P256 are achievable within this decade.

"Harvest Now, Decrypt Later"

Nation-state adversaries are recording encrypted traffic today with intent to decrypt once quantum hardware becomes available. Messages sent today could be decrypted in the early 2030s. For anyone with long-lived secrets, the time to switch is now.

Kryptark's Quantum Posture

* ML-KEM-768 as the sole key exchange for messages and current-generation (v3) file transfer: no classical fallback that could be targeted.
* AES-256-GCM: Grover halves effective key length to 128 bits -- still computationally infeasible.
* Message archive keys (v2) are pure HKDF-SHA256, not Diffie-Hellman -- no discrete-log assumption anywhere in that path.
* Ed25519 signatures live inside AES-GCM ciphertext: an attacker must break AES-GCM first to reach a signature.
* **Known gap:** Ed25519 itself (identity signing, WebRTC/websocket auth, X1NS claims, wallet-level Solana signing) is classical and quantum-vulnerable. Full migration to ML-DSA (FIPS 204) message and identity signing is the next major PQ milestone -- see Section 25. Wallet-level Solana signing is out of scope until the chain itself adopts PQ signatures.
* **Known gap:** legacy v1/v2 file transfers and the personal vault still use X25519-derived keys; only newly-created v3 file transfers are fully post-quantum.
* Module-LWE (basis of ML-KEM) has been studied since 2005 with no known quantum speedup.

Sovereign Identity and Implications for Blockchain

Kryptark's identity model represents a pattern for blockchain-based identity that has implications well beyond messaging. The core insight: a wallet address is a payment credential, not an identity. Conflating the two creates permanent, irreversible privacy loss. Separating them creates a new design space.

* **Self-sovereign:** Your identity is derived from your own key material, not assigned by a registrar.
* **Unlinkable:** Your communication identity (contactId) cannot be linked to your payment identity (wallet) without your private key.
* **Deterministic:** Your identity is reproducible from a single wallet signature. No backup phrase, no key export, no account recovery service.
* **Composable:** The same key material produces different identities for different contexts. Each is independent and non-correlatable.
* **Verifiable:** Anyone can verify your identity is anchored on-chain without learning your wallet address.
* **Portable to zero prior state:** a device with nothing installed can generate a fully-featured identity in-browser and reach the same guarantees as an extension wallet.

The patterns Kryptark demonstrates -- HKDF-derived sub-identities, stealth addresses, hash-only on-chain footprints -- are applicable to any Web3 application that values user privacy. For X1 specifically, Kryptark provides a reference implementation of privacy-preserving dApp design on Solana-compatible infrastructure.

The current state of Web3 -- where every app asks to "connect wallet" and immediately links your wallet address to your action -- is a privacy catastrophe. Kryptark shows the alternative is buildable and deployable today in a browser tab, with or without a pre-existing wallet.

24 Technical Significance and Innovation

First-of-Kind PQ + Blockchain + Stealth Payments + IPFS Archive

No other production messaging system combines NIST FIPS 203 ML-KEM-768, blockchain-anchored identity, stealth payment routing, and an encrypted IPFS message archive. Each component exists in research; Kryptark delivers them as a coherent, usable product running in a browser tab.

Browser-Native Post-Quantum Cryptography

The entire ML-KEM-768 stack runs in the browser with no extensions or binaries. @noble/post-quantum provides a pure-JavaScript FIPS 203 implementation. Any user with a modern browser can use post-quantum encryption with zero setup — with or without an existing wallet.

Full-Featured Identity With No Pre-Existing Wallet

The soft-wallet + passkey path (Section 6) generates the same PQ identity, on the same chain, with the same on-chain anchor, as an extension wallet — funded automatically by a tightly-scoped relay co-signature so a zero-balance browser session becomes a fully self-sovereign account with one tap.

Key Derivation from Wallet Signature Eliminates Key Management

Deriving all keys from a single wallet signature via HKDF eliminates key registration, backup, and recovery problems that plague other systems. The wallet IS the root of trust. Recover your wallet phrase (or soft-wallet seed) and you recover all Kryptark keys and conversation history.

Wallet-Unlinkable contactId Without ZK Proofs

Kryptark achieves wallet-unlinkable identity using only hash functions and HKDF: no zero-knowledge proofs, no ring signatures. The construction achieves full unlinkability with primitives available in every browser.

Stealth Payments Without Smart Contracts

Kryptark's stealth payment system requires no on-chain program, no ZK proof, and no special transaction type. It works with standard SystemProgram.transfer. Unlinkability is achieved entirely off-chain through ECDH key derivation.

Permanent Encrypted History Without a Server

The IPFS linked-list archive with on-chain CID anchoring solves the "server holds your history hostage" problem. Your conversation history is encrypted with a key only you hold, stored on a content-addressed network, and anchored on-chain for verifiable discoverability. It survives server outages, migrations, and company shutdowns — though today it still depends on a single IPFS gateway, honestly disclosed in Section 11.

xCFP Federation: Decentralised Relay Network

The xCFP protocol brings SMTP-style federation to post-quantum encrypted messaging. Multiple independent relay operators interoperate via Ed25519-signed envelopes and per-request challenge-response authentication, now strengthened by an on-chain relay-pubkey trust chain (identity v3) that lets clients detect DNS hijacking or relay impersonation.

Human-Readable Identity Without Sacrificing the Trust Model

X1NS name aliasing (Section 17) turns an unreadable contactId into a name a user already owns, verified through three independent signature and on-chain-ownership checks before the relay will link it — and every result reached via peer gossip is explicitly flagged for client-side re-verification rather than trusted blindly.

Four-Layer Sovereign Internet Stack

The X1 Sovereign Stack is a complete end-to-end replacement of SMTP + DNS + web hosting + browser DNS using only cryptographic primitives and a blockchain. Layer 0 (xCFP) replaces email relays. Layer 1 replaces DNS registrars for hosted xchat sites. Layer 2 (xchat-publish CLI) replaces web hosts. Layer 3 (Chrome MV3 extension) replaces OS DNS resolution for *.x1 names. Website pinning across relays ensures content survives any individual relay going offline.

AI Agent Protocol as First-Class Citizen

The .well-known/xchat-agent.json bootstrap, agent registry, and walletKeypair alternative make AI agents equal participants in the Kryptark network. Agents can send post-quantum encrypted messages, make payments, claim X1NS names, and publish websites with only an Ed25519 keypair -- no browser, no wallet extension, no special setup. The airdrop program on X1 mainnet funds new agent accounts automatically.

25 Roadmap and Next Steps + Conclusion

Live Today — X1 Mainnet

* ML-KEM-768 + Ed25519 + AES-256-GCM end-to-end encryption for messages
* contactId: wallet-unlinkable messaging identity derived via HKDF, independent pdaSeed for on-chain anchoring
* On-chain identity anchor via xchat_pq_v2 (F3ydfNgdM89BK5hDh7amVmt8AAGQSYCX1afb3EWZKGh8), X1 mainnet
* Mobile / no-extension onboarding: soft wallet + relay-funded bootstrap; optional WebAuthn passkey protection, or a passkey-direct account with no locally-stored seed at all
* Direct XNT payments plus optional stealth address payments (X25519 ECDH, two-tx unlinkability)
* Per-payment selective disclosure receipts (3 modes)
* IPFS encrypted message archive — self-derived HKDF keys (v2), no DH/KEM in the archive path; on-chain CID anchoring is an explicit opt-in on top (Section 21)
* Delivery-acknowledgement + one retry on every message, closing the "relay said delivered but it never arrived" gap; WebSocket heartbeat detects a silently-dropped connection within seconds
* Relay-backed archive-hint fallback (POST /api/archive-hint) for cross-device recovery without a prior clean logout
* File transfer v3 (ML-KEM-768, post-quantum); v1/v2 (X25519) kept for backward compatibility only
* Personal encrypted vault (self-only decryption)
* WebRTC voice and video calls, 1:1 and mesh group calls, with a self-hosted STUN/TURN privacy mode that avoids third-party STUN by default
* Group conversations ("Arks"): pairwise per-member fan-out with no shared group key, so member removal needs no re-keying
* Peer-exchanged profile pictures and display names — never centrally published, shared only with the parties in a conversation
* Link previews via server-side privacy proxy (no IP leak); optional relay-proxied IPFS uploads and RPC calls for the same reason
* WebSocket Ed25519 challenge-response auth
* Federation: xCFP relay-to-relay delivery with Ed25519 signed envelopes, deployable across any number of independently-operated relays
* X1NS human-readable name aliasing for contactIds, with signed cross-relay gossip and privacy-proxied resolution
* X1 Sovereign Stack: xCFP + xchat-sovereign name/content registry + xchat-publish + Chrome MV3 extension + bootstrap resolver, all on X1 mainnet
* Self-custodial in-group AI agents with their own PQ identity and wallet: owner/granted-member spend permissions, an owner-signed and runtime-enforced trading policy, XDEX swaps, scheduled recurring actions, and read-only bridge monitoring — see Section 19
* Bootstrap AI Agent Protocol: .well-known manifest, agent registry, on-chain airdrop funding
* On-chain relay registry v2 and identity v3 relay-pubkey trust chain

Near Term

* TURN server integration for calls behind symmetric NAT is live where an operator configures it; broadening default coverage as more relays adopt it
* Multi-relay redundancy for identity registration (register on more than one relay)
* A stable, unlinked funding path for on-chain archive/agent-memory anchoring, so opting in no longer means the anchoring wallet is the same one used elsewhere (Section 21)

Medium Term

* ML-DSA (FIPS 204) message and identity signing: closes the last classical-signature gap described in Section 23
* A post-quantum ratchet for true forward secrecy, layered on top of today's per-message key freshness (Section 4.2)
* Confidential Transfers (SPL Token-2022) for amount-hiding payments on a wrapped asset, alongside the existing stealth-address native-XNT path -- design complete, mint and client wiring not yet built
* Multi-device support with device key attestation
* Message search (client-side, encrypted index)
* contactId rotation: re-derive with new label, notify contacts
* Migrate remaining legacy (v1/v2) file-transfer and vault paths off X25519 where compatibility allows

Long Term

* Decentralised relay via libp2p (no central server)
* Onion routing for IP-level anonymity
* DAO governance of relay infrastructure
* Cross-chain identity (Ethereum, Cosmos, Bitcoin via Taproot)
* Confidential group payments (threshold encryption)
* Formal verification of the cryptographic protocol

Conclusion

Kryptark is one of very few messaging systems that simultaneously ship post-quantum encryption with fresh per-message key material, blockchain-anchored wallet-unlinkable identity, a zero-knowledge relay server, optional stealth payment routing, encrypted IPFS message archives, self-custodial AI trading agents, group conversations with no shared key, a federated relay network, human-readable identity aliasing, and a four-layer decentralised internet stack -- all in a browser-native application that requires no registration beyond a wallet signature, and no pre-existing wallet at all if the user starts from a soft wallet.

This document was written directly from the deployed source — every code sample, endpoint, and PDA layout above was read out of the files that actually ship, not carried forward from an earlier draft. Where the implementation has a known limitation — the single default IPFS upload gateway, the still-classical Ed25519 signing layer, no ratchet yet for true forward secrecy — it is disclosed here rather than smoothed over, with a concrete plan attached in the roadmap above.

The sovereign identity model Kryptark demonstrates is a template for how any blockchain application could handle user identity. The era of "wallet address as username" is a privacy problem that Web3 does

not have to accept. Kryptark shows the alternative is buildable and deployable today in a browser tab — reachable, too, by someone who has never held a crypto wallet before.

The X1 Sovereign Stack and X1NS aliasing take this further: not just private messaging, but a complete sovereign internet where names are owned by keys, websites are pinned across relay nodes, and the infrastructure is governed by the users who run it.

Live Application	`https://kryptark.com` `X1 Mainnet  Browser-native  No pre-existing wallet required`
Federation	`Deployable across any number of independently-operated relays; each installation federat`
Cryptographic Stack	`ML-KEM-768 (FIPS 203)  Ed25519 (RFC 8032)  AES-256-GCM  HKDF-SHA256  X25519 (legacy path`

On-Chain Programs	`xchat_pq_v2: F3ydfNgdM89BK5hDh7amVmt8AAGQSYCX1afb3EWZKGh8  (identity + archive, mainnet)`
	`xchat-sovereign: DdyU28aFeSqUgiD3bLdck7Gzv7AvqpdgT9Yw1doiwWH8  (relay registry + hosting`
	`X1NS: nameQyUhZQQgnirGbbJRR9ECWSdq1W7mMaNUZZTBvtq  (SPL Name Service, mainnet)`
	`Agent airdrop: 9zkypzFPQ2s3D5UqbYuixt3iXo5ig3ZNWLK1TrbNf5eR  (mainnet)`
	`User faucet: stskNxdsPZ9ZgPYrfYaxZGT8325naiVKGfyM53oKTr2  (mainnet)`

Libraries	`@noble/post-quantum  @noble/curves  @noble/hashes` `@noble/ciphers  @solana/web3.js`
Onboarding	`Wallet extension, or in-browser soft wallet protected by a WebAuthn passkey — no extensi`