SANCHR

Privacy Without Compromise

Technical White Paper

Suraj Pandey
Zynclave Tech
suraj@zynclave.com

April 8, 2026
Version 0.0.1

This is a preprint. We welcome review and critique from the security community.

Abstract

Sanchr is an end-to-end encrypted messaging platform that extends forward secrecy beyond message content to all auxiliary cryptographic state. Built on the Signal Protocol's Double Ratchet, Sanchr adds three defenses unified by a single enforcement layer: (1) an OPRF-PSI contact discovery protocol on Ristretto255 that bounds discovery exposure to 24 hours, (2) ratchet-derived media keys with per-step erasure that bind media forward secrecy to the Double Ratchet chain, and (3) an Ephemeral Key Framework (EKF) that enforces temporal guarantees across all auxiliary state classes. The system enforces a single invariant: **no cryptographic material is both cross-domain and long-lived.** This white paper describes Sanchr's architecture, cryptographic design, and threat model for a technical audience.

Contents

Part I: Overview

Introduction

Sanchr is an end-to-end encrypted messaging platform designed for users who want strong security, transparent design, and practical privacy controls. It is built by Zynclave Tech.

Core Thesis

End-to-end encrypted messaging has achieved mainstream adoption through the Signal Protocol. The protocol's Double Ratchet provides strong forward secrecy for message content: each message is encrypted under a unique key, and chain keys are erased after advancement.

However, forward secrecy stops at message content. Production deployments accumulate *auxiliary cryptographic state* outside the ratchet's lifecycle: contact graphs, media encryption keys, presence metadata, and pre-key bundles. This auxiliary state persists with no formal lifecycle management. A compromise at time t exposes not only the current ratchet step but the entire history of auxiliary state since deployment.

Sanchr's position: Forward secrecy must extend beyond message content to all auxiliary cryptographic state. We operationalize time as a security boundary through explicit expiration semantics enforced at the system level.

Design Principles

1. **Time as a security boundary.** Every piece of cryptographic material gets a class, a TTL, and an enforced lifecycle.
2. **No cross-domain persistent state.** No cryptographic material is both accessible from multiple trust domains and long-lived.
3. **Honest threat model.** Every security claim is accompanied by its limitations and residual risks.
4. **Privacy by default.** Strong privacy settings are active out of the box, not hidden in settings menus.
5. **Graceful degradation.** No single-component failure cascades below baseline Signal Protocol guarantees.

What This Paper Covers

- Part I: Overview and design principles
- Part II: Architecture and cryptographic design
- Part III: Threat model
- Appendices: Cryptographic specifications, infrastructure, and assumptions

What This Paper Does Not Cover

Message content confidentiality is handled by the Signal Protocol's Double Ratchet, key distribution by X3DH, and group key management by Sender Keys. These are not modified and are not described here. For formal security proofs and performance benchmarks, see the companion research paper: *"Threat-Driven Extensions to the Signal Protocol"* (Pandey, 2026).

Part II: Architecture & Cryptographic Design

Signal Protocol Integration

Sanchr uses the Signal Protocol for end-to-end encryption of all message content. The integration uses the official Signal Foundation Rust crate (`libsignal`) without modification to the core cryptographic primitives.

X3DH Key Agreement

Before two users can communicate, they must establish a shared secret. Sanchr uses the Extended Triple Diffie-Hellman (X3DH) protocol for asynchronous key agreement. Each device registers three types of public keys: a long-term identity key, a medium-term signed pre-key (rotated weekly), and a set of one-time pre-keys (consumed on first use). A new session combines three or four Diffie-Hellman operations across these key types to derive a shared secret.

Double Ratchet

After session establishment, all messages use the Double Ratchet algorithm. The ratchet combines a Diffie-Hellman ratchet (providing per-message forward secrecy and post-compromise recovery) with a symmetric-key ratchet (deriving unique keys for each message). Chain keys are erased after advancement — compromising a chain key reveals only future messages in that chain, not past messages.

Session Management

Sanchr maintains per-device sessions. Multi-device support uses independent ratchet sessions per device pair. Pre-key bundles are replenished in batches of 100 when the server count drops below 20.

Private Contact Discovery

Contact discovery — the process of determining which of your phone contacts also use Sanchr — presents a fundamental privacy challenge.

The Problem with Hash-Based Discovery

The standard approach of uploading SHA-256 hashes of phone numbers is catastrophically weak. The E.164 phone number space contains approximately 10^{10} valid numbers. At SHA-256's throughput of $\sim$50M hashes/second on a consumer GPU, the entire space is enumerable in $\sim$200 seconds. A precomputed rainbow table costs $\sim$320 GB of storage. Server compromise yields the complete social graph of every user.

Signal mitigates this via Intel SGX enclaves, which have been broken by multiple side-channel attacks (Foreshadow, Plundervolt, ÆPIC Leak), rendering hardware-enclave-based approaches unreliable as a sole defense.

Sanchr's Approach: Hybrid Cached Hash + OPRF-PSI

Sanchr employs a two-layer design that combines a negative filter with a full oblivious pseudorandom function (OPRF) private set intersection (PSI) protocol.

Layer 1: Cached Hash (Negative Filter)

The first layer is a Bloom filter of SHA-256$(e164 \,\|\, daily_salt)$, used *exclusively* for negative filtering: contacts confirmed as non-users are excluded from the OPRF path. The filter leaks only negative membership information within a 24-hour window.

The daily salt is rotated via EKF's discovery class (24-hour TTL, §5). The privacy model is explicit: an observer learns that certain numbers are *not* on the platform within a 24-hour window. This is a strictly smaller information leak than the positive-membership leak of hash-based discovery.

Layer 2: Full OPRF-PSI on Ristretto255

For all contacts not eliminated by the negative filter, Sanchr executes a full OPRF-PSI protocol on Ristretto255:

1. Client generates a random blinding scalar r_i per contact.
2. Client computes blinded point: $B_i = r_i \cdot H(e164_i)$ via hash-to-curve and scalar multiplication.
3. Client sends $\{B_1, \ldots, B_n\}$ to the server.
4. Server computes $R_i = k \cdot B_i$ using server secret k, returns $\{R_1, \ldots, R_n\}$.
5. Client unblinds: $T_i = r_i^{-1} \cdot R_i = k \cdot H(e164_i)$.
6. Client compares T_i against the server's registered user set.

What this provides. The server sees only random Ristretto255 points (blinded). Under the Decisional Diffie-Hellman (DDH) assumption, blinded points are computationally indistinguishable from random group elements. The server cannot determine which phone numbers the client queried, even with unlimited computation.

Compared to v1.1 (SHA-256 hash upload). This is a significant upgrade from the hash-based discovery in Version 1.1 of this whitepaper. The previous approach prevented the server from *trivially* reading

phone numbers but did not protect against dictionary attack by a malicious server operator. The OPRF-PSI approach provides formal obliviousness under standard cryptographic assumptions.

Production Optimizations

- **Batch processing:** vectorized scalar multiplication for 500+ contacts.
- **Bloom filter pre-filtering:** skip OPRF for contacts already confirmed as non-users.
- **Background execution:** OPRF runs asynchronously; the UI shows cached results immediately.

Optional and Selective

Contact sync remains entirely optional. Users can skip it during onboarding or choose to sync only selected contacts.

Vault: Forward-Secure Media

The Problem with Standard Media Encryption

In standard E2EE messaging designs, each media attachment is encrypted with a random AES-256 key stored alongside message metadata. The Double Ratchet erases chain keys after advancement, but media keys are independent artifacts that persist on the device indefinitely. Device compromise at time t exposes *all historical media* — the ratchet's forward secrecy does not extend to attachments.

Sanchr's Approach: Ratchet-Derived Media Keys

Sanchr binds media encryption keys to a parallel HKDF chain that advances in lockstep with the conversation:

$$\text{MediaK}_n = \text{HKDF}(\text{CK}_n,\ \text{file\_hash},\ \texttt{"media-v1"})$$

where CK_n is the parallel media chain key at step n (erased after advancement), $\text{file\_hash} = \text{SHA-256}(\text{plaintext})$ provides per-file uniqueness, and `"media-v1"` is the domain separation label.

Why a Parallel Chain

Binding media derivation to the message ratchet directly would couple reliability failures (upload retries, network timeouts) to cryptographic state progression. Media upload is asynchronous and may span multiple ratchet steps; a failed upload should not advance or block the message chain. The parallel chain maintains independence while inheriting equivalent forward secrecy.

Encryption

AES-256-GCM with MediaK_n as the key, chunked at 1 MB blocks. Encrypted ciphertext is uploaded to object storage via a presigned URL. MediaK_n is wrapped under the recipient's current ratchet key and transmitted as message metadata. The server never possesses the decryption key.

The Re-Access Problem

After ratchet advancement, CK_n is erased. The recipient cannot re-derive MediaK_n to re-open previously received media.

Solution: Device-Local Access Key Cache

$$\text{AccessK}_i = \text{HKDF}(\text{MediaK}_n,\ \textit{dls},\ \texttt{"access-v1-<id>"})$$

where *dls* is a 256-bit device-local secret generated at application installation, stored in the device's secure enclave, and never transmitted. AccessK_i is stored exclusively in device secure storage and enables re-decryption without retaining MediaK_n. EKF enforces a 30-day TTL on AccessK_i entries (§5).

Trust Domain Separation

MediaK_n transits the server (inside encrypted messages); AccessK_i never does. This eliminates server-side recoverability of historical media keys.

- **Server compromise alone:** yields encrypted ciphertext but no device-local secret. The server gains nothing beyond what the ratchet already protects.

- **Device compromise alone:** yields only the unexpired AccessK_i entries (bounded to $\leq$30 days by EKF).
- **Compound compromise:** without our design, server breach recovers MediaK_n for all media ever sent. With our design, server breach yields no device-local key material; device breach yields only the time-bounded subset.

Core Invariant

No cryptographic material is both cross-domain and persistent.

MediaK_n is cross-domain (transits the server) but not persistent (erased at ratchet step). AccessK_i is persistent (up to 30 days) but not cross-domain (never leaves the device). No key violates both constraints. This invariant is what makes the design resilient to staggered cross-domain compromise.

Dual-Mechanism Server-Side Deletion

Each Vault item has an expiration timestamp set at creation time. Expiration is enforced through two independent mechanisms:

1. **Database TTL** automatically deletes the metadata row.
2. **Object storage lifecycle rules** automatically delete the encrypted blob.

If one mechanism fails, the other catches it. The server never holds the AES decryption key — it stores and eventually deletes an opaque encrypted blob.

Browsable Organization

Unlike disappearing messages that vanish from the chat timeline, Vault items are organized in a dedicated interface with filtering by media type, sender attribution, and remaining time until expiration. Users can explicitly save items to their device before expiration if desired.

Share Controls

The sending user can configure whether the recipient is permitted to save or forward Vault items. These controls are enforced by the client application and provide meaningful friction against casual redistribution. They do not prevent a determined user from capturing content through external means.

Ephemeral Key Framework

The Problem: Auxiliary State Accumulation

Production E2EE deployments accumulate auxiliary cryptographic state outside the ratchet: contact discovery sessions, media access keys, presence metadata, pre-key bundles. Without explicit lifecycle management, this state persists indefinitely. At $\sim$200 bytes/user/day, one million users produce $\sim$73 GB of unmanaged cryptographic material per year.

EKF: Unified Lifecycle Management

The Ephemeral Key Framework (EKF) provides a uniform abstraction for temporal scoping of cryptographic state across heterogeneous subsystems. It is not an independent defense parallel to contact discovery and media encryption — it is the *enforcement mechanism* that makes their temporal security guarantees hold.

Without EKF:

- The Bloom filter salt in contact discovery never rotates (the 24-hour bound becomes unbounded).
- AccessK entries in media encryption never expire (the 30-day bound becomes indefinite).

Every piece of auxiliary cryptographic state is assigned a structured lifecycle:

```
EphemeralEntry {
    material:    bytes
    class:       KeyClass  // {discovery, media,
                           //  presence, prekey}
    created_at:  timestamp
    ttl:         duration
    policy:      ExpPolicy // {Rotate, Delete,
                           //  Overwrite}
}
```

Key Classes and TTLs

Class	Material	TTL	Policy	Purpose
discovery	OPRF state, Bloom hashes	24 h	Rotate	Bounds contact discovery exposure
media	AccessK entries	30 d	Delete	Bounds post-compromise media exposure
presence	Typing/read state	5 min	Overwrite	Prevents presence metadata accumulation
prekey	One-time pre-keys	7 d	Replenish	Maintains forward secrecy for new sessions

Table 1: EKF key classes and their temporal bounds.

Expiration Policies

- **Rotate:** replace with freshly derived equivalent; zeroize old value. For continuous-availability material.

- **Delete:** destroy with no replacement. For cacheable material that can be rebuilt on demand.
- **Overwrite:** replace with null sentinel (not delete). Prevents absence-of-record from leaking information.

Server-Side Enforcement

EKF runs as a server-side garbage collection layer with client callbacks. The server never decrypts auxiliary material to enforce TTLs — lifecycle management operates purely on metadata (timestamps, class tags, policy flags).

Vync Mode: Atomic Privacy State

Vync Mode addresses a fundamental UX problem in privacy-focused applications: the gap between available privacy features and *actually-configured* privacy features.

When a user activates Vync Mode, the server atomically updates six privacy-related settings in a single transaction:

1. Screenshot protection is enabled
2. Notification previews are hidden
3. Online status is hidden
4. Read receipts are disabled
5. Typing indicators are disabled
6. Stealth theme is activated

Screenshot protection limitations. Screenshot protection relies on OS-level APIs. These APIs are effective against casual screenshots within the app but do not prevent external cameras, screen recording on rooted/jailbroken devices, accessibility service abuse, or OS-level screen mirroring. Vync Mode's screenshot protection is a friction mechanism, not a cryptographic guarantee.

Research in usable security consistently demonstrates that single-action privacy controls achieve higher adoption rates than multi-step configurations.

Calling Architecture

Sanchr provides end-to-end encrypted voice and video calls using WebRTC with a self-hosted signaling layer.

Signaling

The call signaling service handles SDP offer/answer exchange and ICE candidate trickle between call participants. The signaling server manages call session state and enforces timeouts.

Media Transport

For 1:1 calls, media flows directly peer-to-peer when network conditions permit. When direct connectivity is not possible, media is relayed through TURN servers at edge locations. TURN credentials are time-limited and generated using HMAC-based authentication.

Call E2EE

Sanchr's call encryption pre-exchanges SRTP key material through the existing Signal Protocol session rather than relying on standard DTLS-SRTP negotiation. This ensures that relay servers cannot decrypt media, since they never participate in key negotiation.
Implementation caveats under active development:

- WebRTC stack integration — injecting pre-shared SRTP keys requires platform-specific handling.
- Rekeying — the current design uses a single key pair per call; periodic rekey is planned.
- Identity binding — SRTP keys are bound to Signal session identity, with no additional media authentication.
- Renegotiation — call hold/resume must survive SDP renegotiation without falling back to DTLS-SRTP.

Data Storage Architecture

Data Store	Type	Purpose	Encryption
User accounts, keys, contacts, settings	Relational DB	ACID guarantees	Disk-level encryption
Messages, pending queue, receipts, vault metadata	Distributed store	High-throughput with TTL	E2EE ciphertext
Sessions, presence, typing, rate limits	In-memory cache	Ephemeral state	In-memory; encrypted persistence
Media files (photos, videos, documents)	Object storage	Encrypted blobs	Client-side AES-256-GCM

Table 2: Data storage architecture.

Messages are partitioned by conversation and clustered by timestamp in descending order, so fetching the most recent messages is a single-partition sequential read — the optimal access pattern for chat history pagination.

Performance

All three extensions (OPRF-PSI contact discovery, ratchet-derived media keys, and EKF) are implemented in a production messaging application (Rust backend, native iOS client) and evaluated against real workloads.

Metric	Result	Notes
Text message latency overhead	+2.2%	vs. baseline Signal Protocol
Media encryption overhead	+8.5%	HKDF derivation + AccessK cache
Auxiliary state reduction	92.3%	Material pruned by EKF lifecycle enforcement
Implementation size	∼7,400 LOC	Across Rust and Swift
Single OPRF evaluation	0.12 ms (p50)	Server-side, x86-64
Batch 500 contacts	48 ms (p50)	Server-side
Full discovery (cold)	52 ms (p50)	Server-side
Full discovery (warm)	6 ms (p50)	With Bloom filter cache
MediaK derivation	0.003 ms	Server-side
AccessK derivation	0.008 ms	Client-side

Table 3: Performance benchmarks.

For detailed benchmarks and methodology, see the companion research paper.

Part III: Threat Model

Temporal Compromise Analysis

The following table shows what is exposed under device compromise at time t, with and without Sanchr's extensions:

Data Class	Baseline (compromise at t)	With Sanchr Extensions
Messages after t	Protected (ratchet)	Protected (ratchet)
Messages before t	Exposed (if stored)	Exposed (if stored)
Media keys	All historical MediaK	Erased at ratchet step; $\leq$30 d AccessK
Contact graph	Full graph via hash inversion	OPRF state: 24 h window; Bloom: 24 h salt
Presence metadata	All historical presence	$\leq$5 min window (EKF overwrite)
Pre-key bundles	All uploaded pre-keys	$\leq$7 d window (EKF replenish)

Table 4: Temporal compromise analysis: what an attacker learns.

Threat 1: Compromised Server

Scenario. An attacker gains full control of Sanchr's server infrastructure.
What the attacker learns:

- Encrypted ciphertext blobs — cannot be decrypted without device-held private keys
- Server timestamps of message send/receive events
- Conversation participant lists
- Content type hints ("text", "image", "video") but not actual content
- Blinded Ristretto255 points from OPRF queries (indistinguishable from random under DDH)
- Bloom filter state — reveals only negative membership within a 24-hour window
- Public key bundles (public by design)
- Presence information (bounded to $\leq$5 min by EKF)

What the attacker cannot learn:

- Message plaintext, media content, or file contents
- Private keys or ratchet state (device-only)
- SRTP keys for call media (transmitted inside E2EE messages)
- AccessK entries for media (device-only, never transmitted)
- Which phone numbers a client queried (OPRF obliviousness)

Key improvement over v1.1: The OPRF-PSI protocol means server compromise no longer yields the reversible hash table of all users' contact queries. The contact graph exposure is reduced from "complete social graph since deployment" to "negative-only Bloom filter within a 24-hour window."

Threat 2: Man-in-the-Middle Attack

Scenario. An attacker intercepts the connection and attempts to substitute public keys.
Protection. Safety number verification allows users to compare shared safety numbers out-of-band.
Limitation. Safety number verification requires user action. Users who do not verify are vulnerable to a persistent MITM attack. This limitation is shared by all Signal Protocol implementations.

Threat 3: Device Seizure

Scenario. An attacker gains physical access to a user's unlocked device.
What is exposed: decrypted message history, private keys and ratchet state, unexpired AccessK entries ($\leq$30 days), downloaded Vault items, and contact/conversation metadata.
Key improvement over v1.1: Media exposure is now bounded to $\leq$30 days (AccessK TTL) rather than "all historical media." Past the 30-day window, media keys are irrecoverable even with full device access.
Other mitigations: Vync Mode, screen lock with biometric authentication, Vault expiration, and disappearing messages.
Limitation. An unlocked device with full disk access exposes all locally stored data within the TTL window. This is a fundamental limitation of any E2EE system.

Threat 4: Metadata Analysis

Scenario. An attacker with access to server logs performs traffic analysis.
Available metadata: timing of sends/receives, message sizes, conversation participation, online/offline patterns, call metadata, push notification routing tokens, client IP addresses.
Current mitigations: presence suppression via Vync Mode ($\leq$5 min EKF overwrite), minimal content type hints, no message content in push notifications.
Planned mitigations: sealed sender (V2), ciphertext padding, IP log rotation, push notification metadata minimization.

Threat 5: Legal Compulsion

Scenario. Sanchr receives a legal order to produce user data.
What can be provided: encrypted ciphertext (useless without device keys), account timestamps, Bloom filter state (24-hour window only), IP addresses (subject to retention), timing/call metadata.
What cannot be provided: message content, private keys, call audio/video, AccessK entries, or the phone numbers behind OPRF queries.

Threat 6: Pre-Key Exhaustion

Scenario. An attacker rapidly consumes all one-time pre-keys.
Impact. New sessions fall back to the signed pre-key only, slightly reducing forward secrecy.
Mitigations: rate limiting, pre-key count monitoring with automatic client alerts, batch replenishment of 100 pre-keys, and EKF 7-day TTL with replenish policy.

Threat 7: Endpoint Compromise

Scenario. Malware, a rooted/jailbroken OS, or a compromised accessibility service runs on the user's device.

A compromised endpoint can read all decrypted content, extract private keys, bypass screenshot protection, capture vault items, log keystrokes, and exfiltrate metadata. This is a fundamental limitation of all E2EE systems.

Planned improvements: platform secure enclave integration for key storage, root/jailbreak detection, multi-device key sync with independent ratchets, client app attestation.

Threat 8: Staggered Cross-Domain Compromise

Scenario. Server compromised at time t, device compromised at time $t + \delta$.

This is the critical scenario where traditional designs fail completely.

Without Sanchr's extensions: server breach yields MediaK_n for all media ever sent (inside encrypted messages stored server-side). Subsequent device breach is irrelevant — all media is already recoverable.

With Sanchr's extensions: server breach yields *no device-local key material*. Device breach at $t + \delta$ yields only AccessK entries that have not TTL-expired. Blast radius reduces from "all media ever" to "device-local + time-bounded subset."

This protection comes directly from the core invariant: no cryptographic material is both cross-domain and persistent.

Cryptographic Specifications

Key Types and Algorithms

Key	Algorithm	Size	Lifetime
Identity Key	Curve25519	256-bit	Permanent (per device)
Signed Pre-Key	Curve25519	256-bit	Rotated weekly
One-Time Pre-Key	Curve25519	256-bit	Single use (consumed)
Message Key	AES-256-GCM	256-bit	Single message
HMAC Key	HMAC-SHA256	256-bit	Per ratchet step
MediaK	HKDF-SHA256	256-bit	Per ratchet step (erased)
AccessK	HKDF-SHA256	256-bit	30-day TTL (EKF)
OPRF Blinding Factor	Ristretto255 scalar	256-bit	Per session
OPRF Server Secret	Ristretto255 scalar	256-bit	Weekly rotation
Vault Media Key	AES-256-GCM	256-bit	Per media item
SRTP Master Key	AES-128-CM	128-bit	Per call session
Password Hash	Argon2id	Variable	Stored as hash
Bloom Filter Salt	SHA-256	256-bit	24-hour TTL (EKF)

Table 5: Key types and algorithms.

X3DH Key Agreement Parameters

- Curve: Curve25519
- Hash: SHA-256
- Info string: “Sanchr-X3DH”
- One-time pre-keys per batch: 100
- Replenishment threshold: 20

Double Ratchet Parameters

- Root key derivation: HKDF-SHA-256
- Chain key derivation: HMAC-SHA-256
- Message key derivation: HMAC-SHA-256
- Symmetric encryption: AES-256-GCM
- Maximum skip: 1000 message keys

HKDF Domain Separation Labels

All HKDF derivations use prefix-free domain separation:

- MediaK: `"media-v1"`
- AccessK: `"access-v1-<id>"`
- Cached hash: `"disc-v1"`
- Rotation key: `"rotate-v1"`

Argon2id Password Hashing

- Memory: 64 MB
- Iterations: 3
- Parallelism: 4
- Salt: 16 bytes (random)
- Output: 32 bytes

Media Encryption

- Algorithm: AES-256-GCM
- Key: 256-bit (derived via HKDF from parallel media chain)
- IV/Nonce: 96-bit random
- Authentication tag: 128-bit
- Chunk size: 1 MB
- Key distribution: inside Signal Protocol encrypted message

SRTP Call Encryption

- Cipher: AES-128-CM (Counter Mode)
- Authentication: HMAC-SHA1-80
- Key derivation: from master key exchanged via encrypted message
- Rekeying: not currently implemented; planned for future release

Infrastructure Specifications

Server Components

Component	Technology	Purpose
Core service	Rust (async)	Auth, messaging, contacts, vault, media, settings
Call service	Rust (async)	WebRTC signaling, TURN credential management
EKF lifecycle manager	Rust	Temporal enforcement, TTL garbage collection
OPRF-PSI service	Rust	Private contact discovery
Relational DB	PostgreSQL	Users, keys, contacts, conversations, settings
Message DB	ScyllaDB	Messages, pending queue, receipts, vault metadata
Cache	Redis-compatible	Sessions, presence, typing, rate limits
Event bus	NATS JetStream	Message routing, call signaling, push dispatch
Object storage	S3-compatible	Encrypted media blobs
TURN/STUN	Self-hosted	WebRTC media relay

Table 6: Server components.

Implementation Breakdown

Component	Language	LOC
OPRF-PSI service	Rust	∼2,200
Media key derivation	Rust + Swift	∼1,400
EKF lifecycle manager	Rust	∼1,800
Client key management	Swift	∼1,100
Protocol integration	Rust	∼900
Total		**∼7,400**

Table 7: Implementation breakdown by component.

Deployment

- Orchestration: Kubernetes with horizontal autoscaling (3 nodes)
- TLS: TLS 1.3 external, mutual TLS internal
- CI/CD: automated formatting, linting, testing, build, and container delivery

Capacity Design Targets

These are architectural design targets, not benchmarked results.

Metric	Design Target
Concurrent users	1M+
Message throughput	100K+ messages/second
Message delivery latency (p99)	<500 ms
Call setup latency	<2 seconds
Vault expiration accuracy	Within 60 seconds of configured TTL
Pre-key replenishment latency	<5 seconds from alert to upload
OPRF batch latency (500 contacts)	<100 ms
EKF lifecycle tick accuracy	<60 seconds

Table 8: Capacity design targets.

Assumptions and Non-Goals

Assumptions

1. **Client devices are trusted at the application layer.** Sanchr assumes the mobile OS provides basic process isolation and that the client application has not been tampered with.
2. **The Signal Protocol implementation is correct.** Sanchr uses the official Signal Foundation Rust crate. A bug in the protocol library would affect encryption guarantees.
3. **Standard cryptographic primitives are secure.** Security depends on the computational hardness of Curve25519 ECDH, AES-256-GCM, SHA-256, HKDF-SHA-256, Argon2id, and the DDH assumption on Ristretto255.
4. **Deployment infrastructure provides disk encryption and network isolation.** Server-side security assumes the hosting provider's protections function as documented.

Non-Goals

Sanchr explicitly does not claim to:

- **Protect against a compromised endpoint.** If the recipient's device is compromised, all content displayed on that device within the TTL window is exposed.
- **Prevent screenshots by external means.** A camera pointed at a screen can capture content regardless of software protection.
- **Provide anonymous communication.** Sanchr requires phone number registration and is a privacy tool, not an anonymity tool.
- **Fully hide communication metadata.** The server observes conversation membership, message timing, and call metadata. Sealed sender (V2) will reduce but not eliminate metadata exposure.
- **Guarantee media deletion on recipient devices.** Server-side expiration removes media from Sanchr's infrastructure only. EKF enforces AccessK expiration on the device, but cannot guarantee erasure if the device is compromised before TTL fires.
- **Replace legal or operational security practices.** Sanchr provides technical privacy controls, not protection against social engineering, coercion, or legal compulsion.

Sanchr is built by the Zynclave team.

For security disclosures, contact `security@sanchr.com`

For the companion research paper, visit `https://sanchr.com/research.pdf`