KeyPears: Federated Secret Exchange

Ryan X. Charles

ryan@ryanxcharles.com

April 2026

Abstract. KeyPears is a federated protocol for end-to-end encrypted communication and secret management. User identities are email-style addresses (`name@domain`) backed by NIST P-256 key pairs. Any domain can host a KeyPears server, and servers discover each other through DNS and a well-known configuration file. All cryptographic operations—key derivation, Diffie-Hellman key exchange, encryption, and proof of work—execute client-side using NIST-approved primitives (SHA-256, HMAC-SHA-256, PBKDF2-HMAC-SHA-256, AES-256-GCM, and P-256 ECDSA/ECDH). Servers store only ciphertext and never possess the keys needed to decrypt it. A proof-of-work mechanism provides Sybil resistance for account creation, authentication, and messaging without CAPTCHAs or third-party services. This paper describes the protocol design, cryptographic construction, federation model, and security analysis.

1. Introduction

Internet communication has relied on email for over four decades. Email got two things right: human-readable addresses (`name@domain`) and federation via DNS (any domain can run a mail server, and servers discover each other through MX records). These properties gave email universal reach without central control.

But email was designed for trusted networks. Two fundamental deficiencies, deeply embedded in the SMTP protocol, have resisted every attempt at correction.

No key exchange. Without cryptographic keys bound to addresses, end-to-end encryption requires users to manage keys manually. PGP attempted this: public-key cryptography layered onto email via key servers and a web of trust. Whitten and Tygar demonstrated the result in 1999 [1]: given 90 minutes with PGP 5.0, the majority of test participants could not successfully encrypt a message. Many sent plaintext believing it was encrypted. The problem was not the cryptography but the user interface—key generation, key exchange, trust decisions, and revocation are concepts that do not map onto any familiar workflow. Twenty-seven years later, encrypted email remains a niche practice.

No cost to send. Delivering an email costs the sender nothing. This makes spam economically rational: the cost of sending a million messages is negligible, and even a tiny conversion rate is profitable. Back proposed Hashcash in 1997 [2]—a proof-of-work scheme that imposes a computational cost on each message, making bulk sending expensive. The idea was sound, but SMTP has no mechanism to negotiate proof of work between sender and receiver. Hashcash could not be deployed on email because backwards compatibility prevents making it mandatory: a server that rejects mail without proof of work would lose legitimate messages from senders who have never heard of Hashcash.

Attempts to layer fixes onto email have produced extraordinary complexity. DMARC [3], published in 2015, requires three interlocking mechanisms—SPF for IP-based sender authorization, DKIM for cryptographic message signing, and DMARC itself for policy and reporting. Each has its own DNS records, failure modes, and deployment challenges. After years of effort, DMARC authenticates the sender's domain but provides zero confidentiality. The message content is still plaintext. This is the cost of backwards compatibility.

Centralized alternatives have taken a different approach. Signal solved end-to-end encryption for billions of users with a single software update, but at the cost of centralized identity: your address is a phone number controlled by a carrier, and a single organization runs every server. As Marlinspike argued [4], centralization

enables rapid iteration—but it also means one entity controls your identity, your keys, and your social graph. If that entity changes policy, you have no recourse except to leave and lose your address.

We propose a protocol that keeps what email got right—federated `name@domain` addressing and DNS-based server discovery—while adding what email could not: Diffie-Hellman key exchange for end-to-end encryption, and proof of work for spam mitigation. Both are mandatory from day one, not retrofitted onto a protocol that was never designed for them.

2. Design Principles

KeyPears is guided by five principles:

1. **Federated.** Any domain can run a KeyPears server. Servers discover each other via DNS. No registration authority controls participation.
2. **End-to-end encrypted.** Servers store only ciphertext. Plaintext never leaves the client. The server operator cannot read messages or vault entries.
3. **Client-side proof of work.** Every account creation, login, and message requires proof of work computed by the client. No CAPTCHAs, no third-party verification services.
4. **DNS-based identity.** Addresses are `name@domain`. Identity is bound to DNS domain ownership, not to a phone number or a central registry. If you own your domain, you own your identity.
5. **No trusted third party.** No certificate authority, no central key server, no phone-number registry. Domain verification relies on standard HTTPS/TLS.

3. Overview

A typical interaction proceeds as follows. Alice (`alice@a.com`) wants to send an encrypted message to Bob (`bob@b.com`). Neither Alice nor Bob has communicated before.

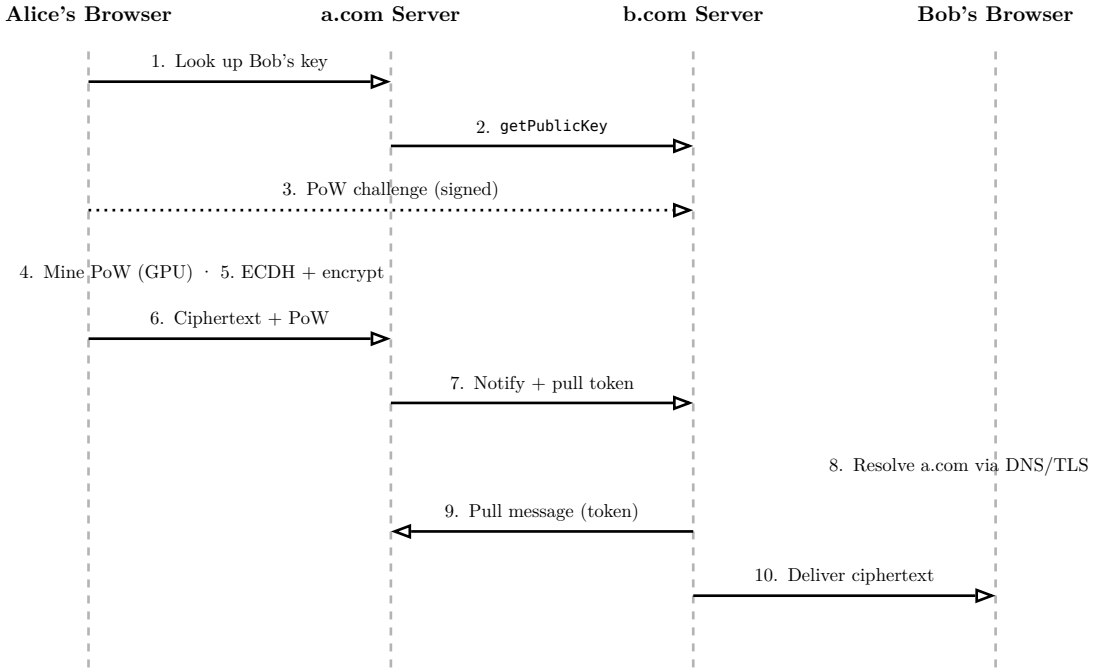

Figure 1: Message flow: Alice sends an encrypted message to Bob across two federated domains.

1. Alice's client asks her server for Bob's public key. Her server fetches it from Bob's server via the federation API.

2. Alice's client requests a proof-of-work challenge from Bob's server. The request is signed with Alice's P-256 private key to prove her identity.
3. Alice's client mines the challenge on the GPU.
4. Alice computes a shared secret via ECDH on P-256 (her private key, Bob's public key), derives an encryption key with SHA-256, and encrypts the message with AES-256-GCM.
5. Alice sends the ciphertext and PoW solution to her server.
6. Alice's server stores her copy, creates a pull token, and notifies Bob's server.
7. Bob's server independently resolves Alice's domain via DNS and TLS—it does not trust the notification. It pulls the ciphertext using the token.
8. Bob's client retrieves the ciphertext, re-derives the ECDH shared secret, and decrypts.

At no point does any server possess the plaintext or the keys needed to derive it.

4. Identity and Addressing

A KeyPears address has the form `name@domain`—intentionally identical to an email address. The protocol places no restrictions on the local part beyond what email itself allows. An organization with existing email addresses can use the same addresses for KeyPears without any changes. The domain is a standard DNS domain.

Each user holds one or more NIST P-256 key pairs. The most recent key is the active key, used for ECDH key agreement in new messages. Users may rotate keys freely, up to 100 per account. Old keys are retained so that messages encrypted under previous keys can still be decrypted.

Private keys are encrypted client-side with AES-256-GCM under the user's encryption key (Section 5) and stored on the server as ciphertext. The server cannot decrypt them. If a user changes their password, keys encrypted under the old password are re-encrypted; keys from a different password remain "locked" until the user provides the old password.

Identity is bound to domain ownership. An address like `alice@acme.com` survives changes in hosting provider: if `acme.com` migrates from one KeyPears server to another, Alice's address and identity remain valid. Only the `keypears.json` configuration file is updated.

5. Key Derivation

Password-based key derivation uses a three-tier PBKDF2-HMAC-SHA-256 scheme (RFC 8018). The server-side tier alone performs 600,000 rounds, matching the OWASP Password Storage Cheat Sheet recommendation for PBKDF2-HMAC-SHA-256; with the two client-side tiers of 300,000 rounds each, the full password-to-hash chain runs 1,200,000 rounds.

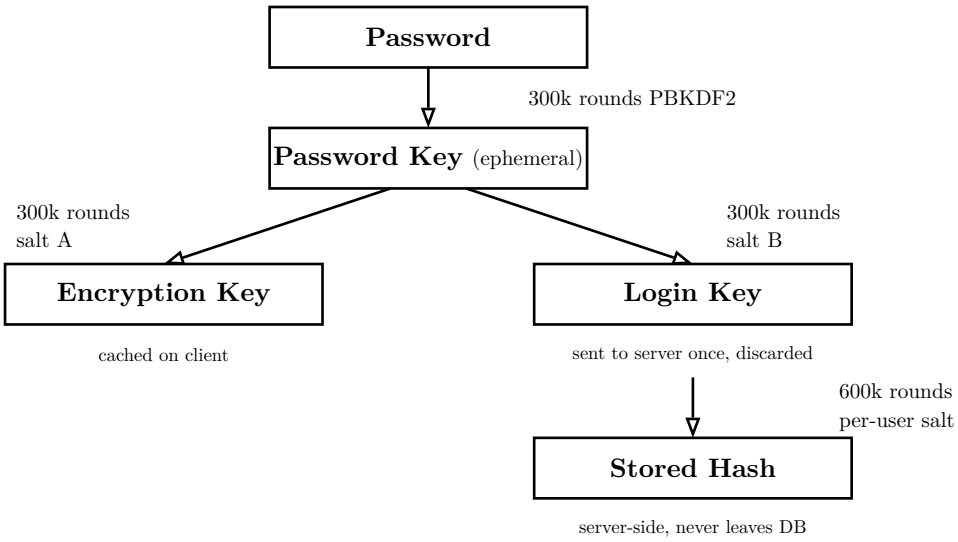

Figure 2: Three-tier key derivation. The password key is ephemeral. The encryption key and login key are siblings—compromising one does not reveal the other.

Tier 1: Password → Password Key. The password is stretched with 300,000 rounds of PBKDF2-HMAC-SHA-256 using a deterministic salt derived from the password itself via HMAC-SHA-256. The result is a 256-bit password key. This key is ephemeral—it is used to derive the encryption key and login key, then discarded.

Tier 2a: Password Key → Encryption Key. A second 300,000-round PBKDF2 derivation with a distinct salt produces the encryption key. This key is cached on the client and used to encrypt and decrypt P-256 private keys client-side. It is never sent to the server.

Tier 2b: Password Key → Login Key. A parallel 300,000-round PBKDF2 derivation with a different salt produces the login key. This key is sent to the server exactly once during account creation or login, then discarded on the client. The server hashes it with an additional 600,000 rounds of PBKDF2-HMAC-SHA-256 before storage, using a per-user salt derived deterministically from the user's ID. The server-side tier alone meets the OWASP recommendation of 600,000 rounds for PBKDF2-HMAC-SHA-256, independent of any work performed on the client. The per-user salt prevents parallel dictionary attacks across multiple users.

The encryption key and login key are derived from the same parent with different salts, making them cryptographically independent. An attacker who compromises client storage obtains the encryption key and can decrypt private keys on that device, but cannot derive the login key or impersonate the user on the server.

Vault key. A separate key for encrypting stored secrets is derived as $K_{\text{vault}} = \text{HMAC-SHA-256}(K_{\text{private}}, \texttt{"vault-key"})$, where the second argument is a fixed domain-separation string. Each vault entry is independently encrypted with AES-256-GCM under K_{vault}.

6. Encryption

KeyPears uses AES-256-GCM (NIST SP 800-38D) for all symmetric encryption. AES-GCM is an authenticated encryption mode (AEAD) that produces ciphertext and an authentication tag in a single pass—no separate MAC is required. Two encryption modes are used.

Message encryption. When Alice sends a message to Bob, she computes a shared secret via elliptic-curve Diffie-Hellman on the NIST P-256 curve:

$$S = \text{SHA-256}(\text{ECDH}(a, B))$$

where a is Alice's private key and B is Bob's public key. Here ECDH(a, B) denotes the 32-byte big-endian x -coordinate of the shared point aB, as returned by the Web Crypto `deriveBits` API—not the 33-byte SEC1 compressed encoding. Both parties must hash the same raw coordinate for their derived keys to agree. The message payload is encrypted with AES-256-GCM using S as the key. Both Alice's and Bob's public keys are stored alongside the ciphertext, so that either party can re-derive the shared secret after key rotation.

Vault encryption. The vault stores secrets—passwords, credentials, and notes—encrypted under the vault key derived from the user's private key (Section 5). The server stores ciphertext alongside user-provided plaintext labels (name and search terms) to enable server-side search without revealing secret content.

7. Federation

Any domain can participate in the KeyPears federation by serving a configuration file at `/.well-known/keypears.json`:

```
{ "apiDomain": "keypears.acme.com" }
```

This file declares the domain's API endpoint. An optional `admin` field names a KeyPears user authorized to manage accounts for the domain. Three deployment patterns are supported:

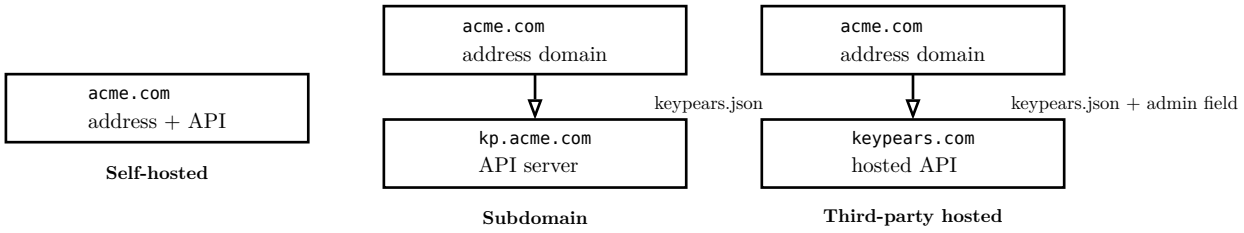

Figure 3: Three federation patterns. In each case, the address domain (`acme.com`) may differ from the API server.

- **Self-hosted:** The domain and API endpoint are the same host.
- **Subdomain:** The API runs on a subdomain (e.g., `kp.acme.com`), keeping the main domain free for other uses.
- **Third-party hosted:** The domain delegates its KeyPears service to another operator (e.g., `keypears.com`). An `admin` field in `keypears.json` names the authorized administrator.

Pull-model message delivery. Cross-domain messages use a pull model rather than server-to-server push. The sender's server stores the ciphertext and issues a pull token with a time-limited expiry. It notifies the recipient's server, which independently resolves the sender's domain via DNS and TLS—it does not trust the notification. The recipient pulls the ciphertext using the token. The pull is idempotent: if the recipient's server fails mid-delivery, it can retry with the same token. Pending deliveries are cleaned up after expiry.

This design provides domain verification without server signing keys. The recipient discovers the sender's API endpoint by resolving the sender's domain itself, via HTTPS. A malicious sender cannot forge another domain's identity because TLS guarantees the `keypears.json` response came from the real domain.

Because the pull happens synchronously during the send, the sender receives immediate confirmation of delivery or an immediate error. There is no outbox queue, no silent retry, and no delayed bounce notification days later—a failure mode familiar to anyone who has used email.

8. Proof of Work

Back proposed Hashcash [2] in 1997 as a proof-of-work scheme to make email spam expensive. The idea was correct, but SMTP has no mechanism to negotiate proof of work, and backwards compatibility prevents

making it mandatory. KeyPears makes proof of work a first-class protocol requirement, building on Hashcash with four adaptations.

Interactive challenges. Hashcash is non-interactive: the sender chooses a start value and the recipient verifies the result. KeyPears uses interactive challenges: the recipient's server issues a challenge signed with HMAC-SHA-256, including a 15-minute expiry. Challenges are stateless—no database entry is created until a valid solution is submitted. This prevents pre-computation attacks.

GPU mining. All proof of work is computed client-side using the `pow5-64b` algorithm, designed for efficient GPU execution via WebGPU. Servers never mine. On a modern laptop GPU (55M hashes/s), a difficulty-70M challenge takes approximately 1–2 seconds and a difficulty-7M challenge completes in under a second.

Configurable difficulty. The protocol does not dictate fixed difficulty levels. Instead, difficulty is set independently at two layers:

- **Server operators** set the difficulty for account creation and login. An operator experiencing spam can raise the account-creation difficulty at any time; an operator under a password-cracking attack can raise the login difficulty. These are operational decisions, not protocol constants.
- **Individual users** set the difficulty for incoming messages: one threshold for first contact (channel opening) and another for subsequent messages. A user receiving unwanted messages can raise their difficulty; a user who values low-friction communication can lower it.

Operators and users are free to choose any value that suits their needs.

Authenticated challenges. Challenge requests require the sender to sign with their P-256 private key (ECDSA). The recipient's server verifies the signature by looking up the sender's public key via federation. Both sender and recipient addresses are bound into the challenge by the server's HMAC-SHA-256. This prevents social-graph probing: an unauthenticated party cannot discover whether two users have communicated.

Solutions are recorded in a spent-token table for replay prevention. Expired entries are cleaned up after their 15-minute window.

9. Security Analysis

9.1. Server Compromise

The server stores only ciphertext (messages, vault entries, encrypted private keys) and hashed credentials (login key hashed with 600,000 additional rounds of PBKDF2-HMAC-SHA-256, using a per-user salt derived deterministically from the user's ID). An attacker who captures the database cannot read any user content.

To impersonate a user, the attacker must recover a valid login key. Brute forcing the login key directly is infeasible—it is a uniformly random 256-bit value, and the search space is 2^{256}. The only realistic attack is a dictionary attack against the user's password: for each candidate password, the attacker computes the full chain (password → password key → login key → stored hash), requiring 1,200,000 rounds of PBKDF2-HMAC-SHA-256 per guess (300,000 for Tier 1, 300,000 for Tier 2b, and 600,000 for the server tier). The per-user salts prevent parallelising a dictionary attack across multiple users—each user's hash must be cracked independently.

9.2. Password Brute-Force

The server-side tier alone performs 600,000 rounds of PBKDF2-HMAC-SHA-256 on every login key, matching the OWASP Password Storage Cheat Sheet recommendation for PBKDF2-HMAC-SHA-256. An offline attack against the stored hash requires 1,200,000 rounds per guess through the full chain. For an 8-character password drawn from lowercase letters and digits ($36^8 \approx 2.8 \times 10^{12}$ candidates), exhaustive search

is computationally infeasible on any realistic hardware budget; longer or more complex passwords increase this cost exponentially. Online attacks are further throttled by the login PoW requirement (7M difficulty per attempt) and by per-IP rate limiting at the infrastructure layer.

9.3. Spam and Sybil Attacks

Every account creation, login, and message requires proof of work, and the difficulty is tunable. The cost of an attack scales linearly with the number of targets and with the difficulty level. For example, at a difficulty of 70M on a modern laptop GPU (55M hashes/s), each action takes approximately 1–2 seconds. Creating 1,000 accounts at this difficulty requires approximately 20 minutes of continuous GPU computation. If an operator is under attack, they can raise the difficulty by an order of magnitude, making the same attack take hours instead of minutes. Recipients who set high message difficulty impose additional per-message costs that make targeted spam impractical.

9.4. Social-Graph Probing

Proof-of-work challenge requests are authenticated: the sender must sign the request with their P-256 private key, and the recipient's server verifies the signature via federation. An unauthenticated party cannot request a challenge, and therefore cannot probe whether two users have a communication channel. Both addresses are signed into the challenge payload by the server's HMAC-SHA-256, preventing cross-conversation reuse.

9.5. Domain Spoofing

The pull model prevents domain spoofing without any additional signing infrastructure. When Bob's server receives a notification from Alice's domain, it does not trust the notification's claimed origin. Instead, it independently resolves Alice's domain via DNS and TLS, fetching `keypears.json` to discover the API endpoint. A malicious server cannot forge another domain's identity because TLS guarantees the response came from the real domain.

9.6. Client Storage Theft

An attacker who compromises a user's client storage obtains the encryption key, which can decrypt the user's P-256 private keys. However, the login key is a cryptographic sibling of the encryption key (derived from the same parent with a different salt), not a child. The attacker cannot derive the login key, cannot impersonate the user on the server, and cannot access the server-side session. The attack surface is limited to decrypting data already present on the compromised device.

Browser-level defense-in-depth is provided by a Content-Security-Policy header that restricts `connect-src` to same-origin, sets `frame-ancestors 'none'` and `base-uri 'self'`, and constrains `default-src`, `script-src`, `style-src`, `img-src`, and `font-src` to same-origin (with `'unsafe-inline'` permitted for scripts and styles as required by the framework, and `'wasm-unsafe-eval'` permitted for the client-side PoW miner). Standard headers round out the policy: `X-Frame-Options: DENY`, `X-Content-Type-Options: nosniff`, and `Strict-Transport-Security`. Together these reduce the exploitability of any XSS or injection bug that might be introduced.

9.7. Limitations

KeyPears does not protect against compromised endpoints (an attacker with access to the running client can read decrypted content), weak passwords (an entropy meter guides users but does not enforce a minimum), or DNS-level attacks such as BGP hijacking (mitigated by DNSSEC where deployed). The protocol does not provide forward secrecy in the Signal sense. All communication is already transported over HTTPS/TLS, so an attacker cannot passively record ciphertext in transit. Forward secrecy protects against an attacker who records encrypted traffic and later compromises a key—but since KeyPears layers E2E encryption inside TLS, this scenario requires compromising both TLS and the user's key. Furthermore, KeyPears messages

persist on the server for later retrieval, so the client must retain decryption keys, limiting the practical benefit of ephemeral key material.

10. Related Work

	PGP	**Signal**	**Matrix**	**KeyPears**
Identity	Email address	Phone number	`@user:domain`	`name@domain`
Federation	Key servers	None	Homeservers	DNS + pull model
E2E encryption	Manual	Automatic	Automatic	Automatic
Spam mitigation	None	Phone reg.	Rate limits	Proof of work
Key management	Manual	Automatic	Automatic	Automatic
Forward secrecy	No	Yes	Yes	No
Open source	Yes	Yes	Yes	Yes

Table 1: Comparison of encrypted communication systems.

PGP (1991) provides strong public-key cryptography but requires users to manage keys, verify fingerprints, and navigate a web of trust. Whitten and Tygar [1] showed that this model is unusable for ordinary people. KeyPears eliminates manual key management entirely: keys are generated automatically, encrypted under the user's password, and exchanged via federation without user intervention.

Signal (2013) solved the usability problem with automatic key management and the Double Ratchet protocol for forward secrecy. However, Signal is centralized: identity is bound to phone numbers controlled by carriers, and a single organization operates all servers. Marlinspike [4] argued that centralization is necessary for rapid iteration. KeyPears accepts slower iteration in exchange for sovereignty: users who own their domain own their identity, and anyone can run a server.

Matrix (2014) is the closest comparison. It is federated, end-to-end encrypted (via Olm and Megolm), and open-source. However, Matrix uses a proprietary address format (`@user:domain`) that is incompatible with email, requiring users to learn and distribute a new identifier. KeyPears uses standard `name@domain` addresses—the same format as email. An organization with existing email addresses can adopt KeyPears without issuing new identifiers. Matrix's architecture is also substantially more complex: room state is maintained as a directed acyclic graph synchronized across homeservers, and the specification spans eight major components. KeyPears makes a deliberate tradeoff toward simplicity: no rooms, no DAG, no state synchronization. The federation layer is a single JSON file and one pull-token API. This limits KeyPears to pairwise communication but dramatically reduces implementation and operational complexity.

Keybase (2014) combined social-proof identity verification with encrypted messaging and a team-based file system. Its design was sound, but it was acquired by Zoom in 2020 and effectively shut down—a cautionary example of centralized hosting for a decentralization-aspirational product. KeyPears avoids this failure mode by design: identity is bound to DNS domains, not to a company, and the protocol can be implemented by anyone.

11. Future Work

Several extensions are planned. **Group messaging** with multi-party key agreement would extend the protocol beyond pairwise communication. **Public-key transparency logs** would provide auditability for key rotations, allowing users to detect unauthorized key changes. A **native mobile client** with hardware-backed key storage would improve security for the encryption key, which is currently cached in client storage.

12. Conclusion

Email demonstrated that federated, human-readable addressing can achieve universal reach without central control. But email's design assumed trusted networks, and four decades of effort have failed to retrofit the two capabilities it lacks: key exchange for encryption, and proof of work for spam resistance. Centralized alternatives solved these problems by abandoning federation, trading sovereignty for convenience. KeyPears takes a different path: a clean-sheet protocol that preserves `name@domain` addressing and DNS-based federation while making Diffie-Hellman key exchange and proof of work mandatory from the start. The result is a system where encryption is the only mode, spam is computationally expensive, and no single entity controls identity.

References

[1] A. Whitten and J. D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," in *Proceedings of the 8th USENIX Security Symposium*, Aug. 1999.

[2] A. Back, "Hashcash - A Denial of Service Counter-Measure," Aug. 01, 2002. [Online]. Available: http://www.hashcash.org/papers/hashcash.pdf

[3] M. Kucherawy and E. Zwicky, "Domain-based Message Authentication, Reporting, and Conformance (DMARC)," *Internet Engineering Task Force*, Art. no. RFC7489, Mar. 2015.

[4] M. Marlinspike, "Reflections: The ecosystem is moving." [Online]. Available: https://signal.org/blog/the-ecosystem-is-moving/