Verifiably Private Email

Closing the Plaintext Gap in Encrypted Email Services via Trusted Execution Environments

Andrew Lee
andrew@bmail.ag

Mark Karpelès
mark@bmail.ag

Version 1.0
March 2026

Abstract

Existing encrypted email services require users to *trust* the service operator not to read email during a critical plaintext window: the interval between when external SMTP mail arrives and when it is encrypted with the recipient's public key. This trust assumption has proven insufficient—service operators have been compelled by legal authorities to modify their systems to capture metadata (Lomas, 2021), and could in principle be compelled to capture plaintext content during this window. We present an architecture that eliminates this trust assumption by combining Intel SGX trusted execution environments (Costan and Devadas, 2016) with DNS-Based Authentication of Named Entities (Hoffman and Schlyter, 2012). Our contributions are threefold: (1) we move SMTP reception and TLS termination entirely inside an SGX enclave, ensuring plaintext email exists only within attested, operator-inaccessible memory; (2) we bind the enclave's TLS identity to DNS via DANE/TLSA records, enabling sending mail servers to cryptographically verify they are communicating directly with the enclave; and (3) we introduce per-message *enclave receipts*—cryptographic proofs that each message was processed exclusively by attested enclave code. Combined with a Key Transparency mechanism (Melara et al., 2015) and the OPAQUE password-authenticated key exchange protocol (Jarecki et al., 2018), the resulting system provides guarantees that are *cryptographically verifiable* rather than merely *policy-asserted*. Companion papers address verifiable non-logging of client metadata (Lee and Karpelès, 2026b) and unlinkable payment authorization (Lee and Karpelès, 2026a).

Contents

1 Introduction **3**
- 1.1 The Promise and Failure of Encrypted Email . . . 3
- 1.2 The Proton Precedent . . . 3
- 1.3 Our Contribution . . . 3

2 Threat Model **4**
- 2.1 Adversary Capabilities . . . 4
- 2.2 Trust Assumptions . . . 4
- 2.3 Security Goals . . . 5

3 Cryptographic Design **5**
- 3.1 OPAQUE Password-Authenticated Key Exchange . . . 5
- 3.2 Message Encryption . . . 6
- 3.3 Forward Secrecy via Epoch Keys . . . 6

4 SGX Enclave Architecture **7**
4.1 The Plaintext Gap . . . 7
4.2 Enclave-Isolated SMTP . . . 7
4.3 DANE/TLSA Binding . . . 7
4.4 Remote Attestation and Reproducible Builds . . . 7
4.5 Sealing Strategy . . . 8

5 Per-Message Enclave Receipts **8**

6 Key Transparency **8**

7 Spam Filtering in the Enclave **9**
7.1 Constraints . . . 9
7.2 Classification Pipeline . . . 9
7.3 Runtime Model Training . . . 9

8 Encrypted Search **9**

9 System Architecture **9**

10 Limitations and Future Work **10**

11 Related Work **10**

12 Conclusion **10**

1 Introduction

1.1 The Promise and Failure of Encrypted Email

The past decade has seen the emergence of encrypted email services that promise to protect user communications from surveillance. Services such as ProtonMail (Yen et al., 2014), Tutanota (Tutao GmbH, 2011), and others encrypt stored email with user-controlled keys and employ zero-access encryption architectures.

However, these services share a fundamental architectural vulnerability that we term the **plaintext gap**: the interval between when an external email arrives via SMTP and when it is encrypted with the recipient's public key. During this window, the email exists in cleartext on the service provider's infrastructure. The provider *promises* not to read or log this plaintext, but this promise is (a) **unverifiable**—users have no way to confirm the server code matches the published source; (b) **legally fragile**—courts can compel providers to modify their systems to capture data during this window; and (c) **operationally risky**—a compromised server, malicious insider, or coerced operator can silently capture plaintext.

1.2 The Proton Precedent

In 2021, ProtonMail complied with a Swiss court order to log the IP address and browser fingerprint of a French climate activist (Lomas, 2021). While ProtonMail correctly noted that email *content* remained encrypted, the incident demonstrated that metadata and operational data are vulnerable to legal compulsion. More critically, it revealed that the architecture *permitted* such modifications—the operator could alter server behavior without users' knowledge or consent.

The fundamental issue is not that ProtonMail acted in bad faith, but that the architecture *required trust in the operator*. Any system where the operator *can* comply with a surveillance order *will eventually be compelled to do so*. As Levison (2013) demonstrated, the alternative to compliance is destruction of the service itself.

1.3 Our Contribution

We present an architecture where the operator *cannot* access user data—not as a matter of policy, but as a matter of cryptographic and hardware-enforced guarantees:

1. **Enclave-Isolated SMTP Reception**: External email is received and encrypted inside an Intel SGX enclave (Costan and Devadas, 2016). TLS terminates within the enclave; plaintext never exists in operator-accessible memory.

2. **DANE-Bound Enclave Identity**: The enclave's TLS certificate is published via DANE/TLSA DNS records (Hoffman and Schlyter, 2012). Sending mail servers that support DANE can verify they are communicating directly with the attested enclave, not an operator-controlled proxy.

3. **Per-Message Enclave Receipts**: Each processed message is accompanied by a cryptographic receipt proving it was handled by an enclave with a specific, reproducibly-built MRENCLAVE measurement.

4. **Key Transparency**: A publicly auditable Merkle tree of user public keys (Melara et al., 2015) prevents the operator from silently substituting keys.

5. **OPAQUE Authentication**: Users authenticate via the OPAQUE protocol (Jarecki et al., 2018), ensuring the server never observes passwords or password-equivalent values.

6. **SGX API Gateway**: The API gateway runs inside an SGX enclave with TLS termination, ensuring client IP addresses exist only in attested, non-logging enclave memory. This component is detailed in a companion paper (Lee and Karpelès, 2026b).

2 Threat Model

2.1 Adversary Capabilities

We consider an adversary with the following capabilities: (1) **full infrastructure control**—the adversary is the service operator, or has fully compromised the operator's infrastructure; (2) **legal compulsion**—the adversary can obtain court orders compelling the operator to modify systems or produce data; (3) **network observation**—the adversary can observe all network traffic to and from the service; and (4) **persistent access**—the adversary has ongoing, not merely point-in-time, access.

2.2 Trust Assumptions

Our security guarantees rely on the following assumptions. First, **SGX hardware integrity**: Intel's SGX implementation correctly isolates enclave memory from all software, including the operating system and hypervisor. We acknowledge known side-channel attacks (Van Bulck et al., 2018; Murdock et al., 2020; Borrello et al., 2022) and assume current microcode patches are applied. Second, **cryptographic soundness**: the primitives used (X25519, ML-KEM-768 (National Institute of Standards and Technology, 2024), XChaCha20-Poly1305, Ed25519, SHA-256, HKDF) are secure against computationally bounded adversaries. The X25519 + ML-KEM-768 hybrid construction we employ for message encryption (Section 3.2) is IND-CCA secure as long as *at least one* of the two component KEMs remains secure: a future cryptographically relevant quantum adversary that breaks the discrete-log problem on Curve25519 still faces ML-KEM-768's lattice assumptions, and a hypothetical break of ML-KEM-768 still leaves X25519 ECDH intact. Third, **DNSSEC integrity**: DNS records authenticated via DNSSEC (Arends et al., 2005) are not forged. Fourth, **client device integrity**: the user's device is not compromised.

2.3 Security Goals

Goal	Guarantee
Confidentiality of stored email	Encrypted at rest with user's public key; private key wrapped with OPAQUE export key. Server never possesses decryption capability.
Inbound transit confidentiality	Plaintext exists only inside attested SGX enclave. TLS terminates inside enclave. Operator cannot access enclave memory.
Internal email confidentiality	True end-to-end: sender encrypts with recipient's public key. No plaintext gap.
Password confidentiality	OPAQUE protocol (Jarecki et al., 2018): server stores OPRF record, never observes password.
Key integrity	Key Transparency log (Melara et al., 2015): unauthorized changes detectable.
Verifiability	Remote attestation + reproducible builds: anyone verifies exact running code.
User anonymity	SGX API gateway (Lee and Karpelès, 2026b): no IP logging. For cryptocurrency payments, Ed25519 Schnorr blind signatures (Lee and Karpelès, 2026a) provide mathematical unlinkability between payment and account. Stripe card payments are offered as a convenience alternative where billing identity is inherently known to the payment processor.

Table 1: Security goals and guarantees.

3 Cryptographic Design

3.1 OPAQUE Password-Authenticated Key Exchange

We employ the OPAQUE protocol (Jarecki et al., 2018) for user authentication, implemented via the `bytemare/opaque` library (RFC 9807, using Ristretto255/SHA-512). OPAQUE is an asymmetric password-authenticated key exchange (aPAKE) in which the server never learns the user's password—not even during registration.

During registration, the client and server execute the OPAQUE registration protocol, producing a server-side *registration record* containing no password-equivalent information, and a client-side *export key* k_{export} derived deterministically from the password. The client generates a hybrid encryption keypair (X25519 for classical ECDH, ML-KEM-768 for post-quantum key encapsulation) and an Ed25519 signing keypair:

$$(pk_{\text{x}}, sk_{\text{x}}) \leftarrow \text{X25519.Generate()} \tag{1}$$

$$(pk_{\text{kem}}, sk_{\text{kem}}) \leftarrow \text{ML-KEM-768.Generate()} \tag{2}$$

$$(pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \text{Ed25519.Generate()} \tag{3}$$

The combined public encryption key $pk_{\text{enc}} = (pk_{\text{x}}, pk_{\text{kem}})$ is what other senders use to encrypt messages to this user.

The private keys are encrypted with the export key:

$$c_{\text{priv}} = \text{XChaCha20-Poly1305}(k_{\text{export}},\ sk_{\text{x}} \| sk_{\text{kem}} \| sk_{\text{sig}}) \tag{4}$$

The server stores $(pk_{\text{x}}, pk_{\text{kem}}, pk_{\text{sig}}, c_{\text{priv}}, \text{registration\_record})$ but can decrypt neither c_{priv} nor derive k_{export}.

During login, OPAQUE produces a mutually authenticated session key k_{session} and the export key k_{export} (client-side only). The client retrieves and decrypts c_{priv}. Key recovery is handled via a BIP-39 mnemonic (Palatinus and Rusnak, 2013); no recovery email or phone number is collected.

3.2 Message Encryption

For each message, we employ a post-quantum hybrid encryption scheme that combines X25519 ECDH with ML-KEM-768 (National Institute of Standards and Technology, 2024) key encapsulation. The message key wrap is derived from the concatenation of both shared secrets, so an attacker must break *both* primitives to recover the per-message key. A versioned envelope (leading byte `0x02`) commits to the hybrid construction in plaintext so that recipients can later parse the envelope without ambiguity.

Algorithm 1 Hybrid Message Encryption (X25519 + ML-KEM-768)

$k_m \xleftarrow{\$} \{0,1\}^{256}$ ▷ Random per-message key
$(ek_{\text{pub}}, ek_{\text{priv}}) \leftarrow \text{X25519.Generate}()$ ▷ Ephemeral X25519 keypair (32 B pub)
$s_{\text{x}} \leftarrow \text{X25519}(ek_{\text{priv}}, pk_{\text{x}})$ ▷ Classical ECDH shared secret (32 B)
$(ct_{\text{kem}}, s_{\text{pq}}) \leftarrow \text{ML-KEM-768.Encaps}(pk_{\text{kem}})$ ▷ Lattice KEM: ct 1088 B, s_{pq} 32 B
$\text{envelope} \leftarrow \texttt{0x02} \,\|\, ek_{\text{pub}} \,\|\, ct_{\text{kem}}$ ▷ Versioned envelope, 1121 B total
$k_{\text{wrap}} \leftarrow \text{HKDF-SHA256}(s_{\text{x}} \,\|\, s_{\text{pq}},\ \text{salt} = ek_{\text{pub}} \,\|\, ct_{\text{kem}},\ \text{info} = \text{“message-key-wrap-v2”})$
$c_k \leftarrow \text{XChaCha20-Poly1305}(k_{\text{wrap}}, k_m, \text{AAD} = \text{envelope})$ ▷ Envelope binds version + both KEMs
$c_{\text{body}} \leftarrow \text{XChaCha20-Poly1305}(k_m, \text{body})$
$c_{\text{subj}} \leftarrow \text{XChaCha20-Poly1305}(k_m, \text{subject})$
return $(\text{envelope}, c_k, c_{\text{body}}, c_{\text{subj}})$

Decryption. The recipient parses the envelope, runs $s_{\text{x}} = \text{X25519}(sk_{\text{x}}, ek_{\text{pub}})$ and $s_{\text{pq}} = \text{ML-KEM-768.Decaps}(sk_{\text{kem}}, ct_{\text{kem}})$ in either order, re-derives k_{wrap} via HKDF, and unwraps c_k to recover k_m. Because k_{wrap} is derived from $s_{\text{x}} \,\|\, s_{\text{pq}}$, an adversary must compromise *both* the X25519 secret and the ML-KEM-768 secret to recover k_m; loss of either one alone leaves k_{wrap} a uniform random value from the adversary's perspective. “Harvest now, decrypt later” attacks against today's traffic therefore additionally require a future cryptanalytic break of ML-KEM-768, not just a quantum break of X25519.

3.3 Forward Secrecy via Epoch Keys

To limit the impact of key compromise, we implement epoch-based key rotation.

Definition 3.1 (Epoch Key Chain)**.** Let $E = \{e_1, e_2, \ldots, e_n\}$ be a sequence of epochs. For each epoch e_i, a fresh keypair (pk_i, sk_i) is generated. The previous epoch's private key is encrypted under the new epoch's public key: $c_{i-1} = \text{Encrypt}(pk_i, sk_{i-1})$.

Messages in epoch e_i are encrypted with pk_i. If sk_i is compromised, messages from epochs $e_j > e_i$ remain secure, as the new keypair is independent. This provides partial forward secrecy without the cost of re-encrypting all historical messages.

4 SGX Enclave Architecture

4.1 The Plaintext Gap

In conventional encrypted email systems, inbound external email follows this path:

$$\text{Sender MTA} \xrightarrow{\text{SMTP/TLS}} \text{Server} \xrightarrow{\text{plaintext in memory}} \text{Encrypt}(pk_{\text{recipient}}) \rightarrow \text{Store ciphertext}$$

The plaintext exists in server memory between TLS decryption and public-key encryption. The server operator has full access to this memory.

4.2 Enclave-Isolated SMTP

We move the entire plaintext-handling pipeline inside an Intel SGX enclave, built using the EGo framework (Edgeless Systems, 2021):

$$\text{Sender MTA} \xrightarrow{\text{SMTP/TLS}} \boxed{\text{SGX: TLS} \rightarrow \text{SMTP} \rightarrow \text{Filter} \rightarrow \text{Encrypt}(pk_{\text{r}})} \xrightarrow{\text{ciphertext only}} \text{Store}$$

Property 4.1 (Enclave Isolation)**.** The enclave boundary ensures: (1) the TLS private key is generated inside the enclave and sealed to the enclave's identity—it never exists in host-accessible memory; (2) the decrypted SMTP message exists only in enclave-protected memory (EPC), which is hardware-encrypted and inaccessible to the host operating system, hypervisor, and DMA devices; (3) only the encrypted message, a folder assignment, and an enclave receipt cross the enclave boundary.

The enclave binary contains: TLS termination for inbound SMTP; an RFC 5321 SMTP server using the `go-smtp` library (Binet, 2019); SPF (Kitterman, 2014), DKIM (Crocker et al., 2011), and DMARC (Kucherawy and Zwicky, 2015) verification; a Go-native Bayesian spam classifier (Section 7); Key Transparency proof verification; hybrid encryption (Section 3.2); and receipt signing (Section 5).

4.3 DANE/TLSA Binding

To prevent the operator from intercepting SMTP connections before they reach the enclave, we bind the enclave's TLS identity to DNS. The enclave generates its TLS keypair internally at first boot. The TLS public key hash is published as DANE/TLSA records (Hoffman and Schlyter, 2012) for both SMTP (port 25) and HTTPS (port 443):

```
_25._tcp.mx.example.com.  IN TLSA 3 1 1 <sha256(smtp_enclave_tls_pubkey)>
_443._tcp.api.example.com. IN TLSA 3 1 1 <sha256(gw_enclave_tls_pubkey)>
```

An MTA-STS policy (Margolis et al., 2018) enforces TLS for all inbound connections, and DNSSEC (Arends et al., 2005) signs the DANE records to prevent DNS tampering.

Sending MTAs that support DANE—including Postfix, Exim, and major providers such as Gmail—verify the TLS certificate against the TLSA record. Since the corresponding private key exists *only inside the enclave*, the operator cannot present a valid certificate from a non-enclave service.

4.4 Remote Attestation and Reproducible Builds

Any party can verify the enclave is running the expected code by requesting an attestation quote from the enclave, which the Intel CPU signs with a measurement (MRENCLAVE) of the enclave binary, the signer's identity (MRSIGNER), and user-defined data (the enclave's TLS and signing public keys). The verifier checks this quote against Intel's attestation infrastructure

and compares MRENCLAVE against the hash produced by a reproducible build of the published source code.

If the operator modifies the enclave code—for example, to add logging—MRENCLAVE changes, and attestation against the published hash fails. For attestation to be meaningful, the mapping from source code to MRENCLAVE must be deterministic. We achieve this via deterministic build environments (Nix or pinned Docker images) with EGo signing.

4.5 Sealing Strategy

The enclave persists state (TLS private key, spam model weights) using **MRSIGNER-based sealing**: data is encrypted to the signer's identity rather than the enclave's identity, allowing code updates without losing sealed state. This is acceptable because all signed binaries are reproducibly built from public source code—any user can verify what code a given MRSIGNER has signed.

5 Per-Message Enclave Receipts

Each message processed by the enclave produces a cryptographic receipt.

Definition 5.1 (Enclave Receipt)**.** An enclave receipt R for a message m is a tuple:

$$R = (mre, pk_{\text{sig}}, H(m), H(c), H_{\text{sender}}, \delta_{\text{tls}}, \delta_{\text{dkim}}, \delta_{\text{spf}}, \delta_{\text{dmarc}}, s, f, t, \sigma) \tag{5}$$

where mre is the MRENCLAVE measurement, pk_{sig} is the enclave's Ed25519 signing key, $H(m)$ is the SHA-256 hash of the original raw message, $H(c)$ is the SHA-256 hash of the encrypted output (computed *after* encryption), $H_{\text{sender}} = \text{SHA-256}(\text{sender\_address} \| \text{daily\_salt})$ is a privacy-preserving hash of the sender's email address using a daily-rotated salt, δ values are authentication verification results (SPF, DKIM, DMARC — each "pass", "fail", or "none"), s is the spam score, f is the folder assignment, t is the timestamp, and $\sigma = \text{Ed25519.Sign}(sk_{\text{sig}}, R \setminus \{\sigma\})$ is the signature over all other fields in canonical byte order.

A user's client verifies a receipt by: (1) verifying σ under pk_{sig}; (2) confirming pk_{sig} matches the signing key in a valid attestation quote; (3) confirming MRENCLAVE matches the reproducible build hash; and (4) confirming $H(c)$ matches the SHA-256 of the received ciphertext. This chain proves: "the ciphertext I received was produced by enclave code with measurement X, built from source code Y, and that enclave was the only entity that processed the plaintext."

6 Key Transparency

If the operator substitutes a user's public key with one for which the operator holds the private key, the enclave will encrypt messages to the wrong key. Key Transparency prevents this.

We maintain an append-only Merkle tree (Melara et al., 2015) where each leaf represents a key binding: $\text{leaf}_i = H(\text{user\_id} \| \text{domain} \| \text{epoch} \| H(pk_{\text{enc}}))$. The tree root is signed by the KT service at each epoch. When a sender (or the SMTP enclave) needs a recipient's public key, the server returns the key along with a Merkle inclusion proof and signed root. The requester verifies the proof; if invalid, it refuses to encrypt and alerts the user.

Each client periodically fetches its own inclusion proof and verifies its key has not been changed without authorization. Third-party auditors verify tree consistency (no forks, append-only). The SMTP enclave verifies recipient public keys against the KT log before encrypting, including the KT epoch in its receipt.

7 Spam Filtering in the Enclave

7.1 Constraints

Spam filtering requires access to message content, which only exists inside the enclave. The filter must be pure Go (EGo only executes Go binaries), compact enough for the EPC memory budget, and effective without operator access to plaintext. External filtering systems (SpamAssassin, rspamd) cannot be used as they are implemented in Perl and C/Lua respectively.

7.2 Classification Pipeline

The enclave implements a multi-layer pipeline: (1) an *authentication layer* performing SPF (Kitterman, 2014), DKIM (Crocker et al., 2011), and DMARC (Kucherawy and Zwicky, 2015) verification; (2) a *reputation layer* with IP-based DNSBL queries and reverse DNS validation; (3) *header heuristics* analyzing structural properties (missing Message-ID, date anomalies, charset irregularities); (4) a *naïve Bayes classifier* operating on hashed feature vectors; and (5) *URL analysis* with blocklist checking. Each layer produces a weighted score. The aggregate determines folder assignment (inbox vs. junk). All spam is *delivered* to the junk folder—the filter classifies but never rejects.

7.3 Runtime Model Training

Property 7.1 (MRENCLAVE Stability)**.** The Bayesian model weights are *runtime data*, not compiled into the enclave binary. Training updates the weights in memory and periodically seals them via MRSIGNER-based sealing. The training *algorithm* is attested (part of MRENCLAVE), but the learned *weights* are not. This allows the model to improve continuously without changing the enclave's attestation measurement.

Users mark messages as spam or not-spam in their client; the client extracts anonymized feature vectors locally and submits them to the enclave's training API. The enclave updates its Bayesian model weights in memory and periodically seals them to persistent storage. Weights survive restarts and code updates via MRSIGNER sealing. Similarly, heuristic scoring thresholds are loaded from a signed configuration file at startup, allowing tuning without binary changes.

8 Encrypted Search

We evaluated Symmetric Searchable Encryption (SSE) schemes and determined they are incompatible with our security goals. All practical SSE constructions leak access patterns, search patterns, and volume patterns. Published leakage-abuse attacks (Cash et al., 2015; Grubbs et al., 2017; Oya and Kerschbaum, 2021) demonstrate these leakages are exploitable, particularly for email where content distributions are predictable. ORAM-based mitigations (Goldreich and Ostrovsky, 1996) impose 100–1000× performance overhead.

We adopt client-side search exclusively: clients download encrypted messages progressively, decrypt them on the main thread with batched indexing (50 messages per batch, yielding to the UI between batches), and build a trigram-based search index persisted in IndexedDB encrypted with AES-256-GCM (key derived via HKDF from the user's export key). Search queries extract trigrams from terms and phrases, intersect trigram matches, and apply sender and phrase filters entirely in the browser with zero information leakage to the server.

9 System Architecture

The system comprises stateless, horizontally-scalable services: an SMTP inbound enclave (EGo SGX), an API gateway enclave (EGo SGX, detailed in Lee and Karpelès (2026b)), an SMTP outbound enclave (EGo SGX, responsible for DKIM signing with enclave-sealed private keys), standard Go services for authentication (OPAQUE), mail workers, Key Transparency, and blind signature payment authorization (Lee and Karpelès, 2026a), backed by PostgreSQL (metadata), MinIO (encrypted blobs), NATS JetStream (message queue), and Redis (session cache). The system supports multi-domain multi-tenancy from inception, with all tables scoped by `tenant_id` for future horizontal sharding via Citus or CockroachDB. For a target of 2 million users (∼40M inbound messages/day), 10 SGX SMTP enclave instances provide 2,000–5,000 msgs/sec throughput.

10 Limitations and Future Work

Outbound email. When a user sends email to an external recipient, the recipient's server expects cleartext SMTP. This is an inherent limitation of email interoperability; for internal (user-to-user) messages, true end-to-end encryption is maintained.

SGX side-channels. Intel SGX has been subject to multiple side-channel attacks (Van Bulck et al., 2018; Murdock et al., 2020; Borrello et al., 2022). While known attacks are patched via microcode, undiscovered vulnerabilities remain possible. We mitigate by abstracting the TEE interface to support migration to AMD SEV-SNP or ARM CCA, and by using transparency logs that would retrospectively detect exploitation.

DANE adoption. Not all sending MTAs support DANE. For senders that do not check TLSA records, the TLS certificate binding is not enforced. As DANE adoption grows, this gap narrows.

EPC memory. SGX v2 hardware (e.g., Azure DCsv3) provides up to 256 GB EPC, significantly relaxing memory constraints for enclave workloads.

11 Related Work

ProtonMail (Yen et al., 2014) and Tutanota (Tutao GmbH, 2011) provide encrypted email with zero-access encryption but rely on operator trust during the inbound plaintext window. OpenPGP (Callas et al., 2007) and S/MIME (Schaad et al., 2019) provide end-to-end encryption between cooperating parties but do not protect email from non-participating senders. CONIKS (Melara et al., 2015) and Google's Key Transparency (Google, 2017) have been proposed for messaging; we apply Key Transparency to email, integrating verification into the SGX enclave. Previous work has explored SGX for secure services including containers (Arnautov et al., 2016), databases (Priebe et al., 2018), and contact discovery (Marlinspike, 2017). Our contribution is the specific application to SMTP reception with DANE binding and per-message receipts.

12 Conclusion

We have presented an architecture for verifiably private email that closes the plaintext gap inherent in existing encrypted email services. By combining Intel SGX enclaves via EGo (Edgeless Systems, 2021) with DANE/TLSA binding (Hoffman and Schlyter, 2012), per-message enclave receipts, Key Transparency (Melara et al., 2015), and OPAQUE authentication (Jarecki et al., 2018), we achieve security guarantees that are cryptographically and hardware-enforced rather than policy-asserted.

The critical distinction from prior work is *verifiability*: users do not trust the operator's promises—they verify the operator's code. Remote attestation, reproducible builds, and transparency logs create an auditable chain from source code to running service. The operator *cannot* access plaintext email, not because they choose not to, but because the architecture makes it impossible.

Combined with the SGX API gateway for verifiable non-logging (Lee and Karpelès, 2026b) and blind signature payments for unlinkable billing (Lee and Karpelès, 2026a), the system provides comprehensive privacy guarantees across content, metadata, network identity, and billing—advancing encrypted email from "trust us" to "verify us."

Verifiable Privacy

Provably private infrastructure for the open internet.

`bmail.ag`

References

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (2005). DNS security introduction and requirements. RFC 4033, IETF.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). SCONE: Secure Linux containers with Intel SGX. In *OSDI*.

Binet, S. e. (2019). go-smtp: An SMTP client and server library for Go. `https://github.com/emersion/go-smtp`.

Borrello, P., Kogler, A., Schwarzl, M., Lipp, M., Gruss, D., and Schwarz, M. (2022). AEPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security Symposium*.

Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and Thayer, R. (2007). OpenPGP message format. RFC 4880, IETF.

Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. (2015). Leakage-abuse attacks against searchable encryption. In *ACM Conference on Computer and Communications Security*, pages 668–679.

Costan, V. and Devadas, S. (2016). Intel SGX explained. Technical report, IACR Cryptology ePrint Archive.

Crocker, D., Hansen, T., and Kucherawy, M. (2011). DomainKeys Identified Mail (DKIM) signatures. RFC 6376, IETF.

Edgeless Systems (2021). EGo: A framework for building confidential apps in Go. `https://github.com/edgelesssys/ego`.

Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473.

Google (2017). Key transparency. `https://github.com/google/keytransparency`.

Grubbs, P., Sekniqi, K., Bindschaedler, V., and Ristenpart, T. (2017). Leakage-abuse attacks against order-revealing encryption. In *IEEE Symposium on Security and Privacy*.

Hoffman, P. and Schlyter, J. (2012). The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. RFC 6698, IETF.

Jarecki, S., Krawczyk, H., and Xu, J. (2018). OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In *Advances in Cryptology — EUROCRYPT 2018*, pages 456–486. Springer.

Kitterman, S. (2014). Sender policy framework (SPF) for authorizing use of domains in email, version 1. RFC 7208, IETF.

Kucherawy, M. and Zwicky, E. (2015). Domain-based message authentication, reporting, and conformance (DMARC). RFC 7489, IETF.

Lee, A. and Karpelès, M. (2026a). Unlinkable payment authorization via blind signatures in trusted execution environments. Technical report, Verifiable Privacy.

Lee, A. and Karpelès, M. (2026b). Verifiable non-logging in network services via attested API gateways. Technical report, Verifiable Privacy.

Levison, L. (2013). Lavabit: The story of a shutdown. *The Guardian*.

Lomas, N. (2021). ProtonMail logged IP address of French activist after order by Swiss authorities. *TechCrunch*.

Margolis, D., Risher, M., Ramakrishnan, B., Brotman, A., and Jones, J. (2018). SMTP MTA strict transport security (MTA-STS). RFC 8461, IETF.

Marlinspike, M. (2017). Technology preview: Private contact discovery for Signal. Signal Blog.

Melara, M. S., Blankstein, A., Bonneau, J., Felten, E. W., and Freedman, M. J. (2015). CONIKS: Bringing key transparency to end users. In *USENIX Security Symposium*.

Murdock, K., Oswald, D., Garcia, F. D., Van Bulck, J., Gruss, D., and Piessens, F. (2020). Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*.

National Institute of Standards and Technology (2024). Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report FIPS 203, U.S. Department of Commerce.

Oya, S. and Kerschbaum, F. (2021). Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security Symposium*.

Palatinus, M. and Rusnak, P. (2013). BIP-39: Mnemonic code for generating deterministic keys. Bitcoin Improvement Proposal.

Priebe, C., Vaswani, K., and Costa, M. (2018). EnclaveDB: A secure database using SGX. In *IEEE Symposium on Security and Privacy*.

Schaad, J., Ramsdell, B., and Turner, S. (2019). S/MIME version 4.0 message specification. RFC 8551, IETF.

Tutao GmbH (2011). Tutanota: Encrypted email service. `https://tutanota.com`.

Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Gras, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*.

Yen, A., Yan, J., and Koch, W. (2014). ProtonMail: End-to-end encrypted email. Technical report, Proton Technologies AG.