Seal: Decentralized Secrets Management

Mysten Labs

January 2026
Version 1.0

1 Introduction

Blockchains have fundamentally transformed how we think about consensus and data availability, enabling a new paradigm of decentralized applications that operate without trusted intermediaries. Today's blockchain ecosystems support a diverse range of authentication mechanisms, from traditional seed phrases and hardware wallets to modern solutions like zkLogin and Passkeys, providing users with unprecedented control over their digital identities.

However, despite this evolution in authentication, blockchain systems lack a standardized, blockchain-native model for encrypting data and enforcing access to encrypted content. While authentication answers the question of "who you are", encryption addresses the equally critical question of "what you are allowed to decrypt, and under what condition". This gap creates significant limitations for applications that require secure, policy-driven access to sensitive data in decentralized environments.

The need for blockchain-native encryption spans a broad spectrum of use cases, ranging from simple one-to-one applications such as secure storage, to complex scenarios involving time-locked disclosures, threshold access, and policy-gated content. Consider a user encrypting their medical records for personal storage: this represents the simplest case, where only the data owner should ever be able to decrypt the content. At the other end of the spectrum lie decentralized trading platforms, where encrypted orders must remain confidential until a specific onchain event to prevent frontrunning, or voting systems in which ballots remain encrypted until the voting period concludes.

Crucially, as blockchains move toward confidential transactions, decryption key management becomes a first-order usability and security challenge. In programmable private transaction systems, users must be able to reliably recover transaction secrets in order to audit, prove, or re-derive transaction state in order to spend their confidential assets. The absence of robust mechanisms for managing these decryption secrets remains one of the largest barriers to mass adoption of private transactions.

Moreover, encryption policies frequently diverge from signing policies. Threshold and multi-signature wallets are commonly used to control authorization over asset movement, yet the entities authorized to decrypt data are not always the same as those authorized to sign transactions. In such settings, centralized systems become both fragile and inefficient, particularly when decryption must respect decentralized governance, quorum thresholds, or time-based constraints.

Finally, modern account abstraction frameworks increasingly support signing-key rotation and flexible authentication policies. This raises a fundamental question: how should encrypted data be re-encrypted, migrated, or selectively disclosed when signing keys change? Treating encryption and signing as the same concern introduces unnecessary coupling and operational risk.

All of the above scenarios illustrate a fundamental challenge: blockchain applications need encryption systems that can enforce complex, dynamic access policies while preserving the decentralized, trust-minimized properties that make blockchains valuable in the first place.

The Challenges. The absence of widely adopted encryption standards in blockchain ecosystems stems from several interconnected challenges. First, there's uncertainty around which encryption schemes to support. Different applications have vastly different requirements: some need RSA or compatibility with existing systems, while others require ElGamal with specific elliptic-curve-based schemes for advanced cryptographic workflows. The diversity of requirements makes it difficult to define a one-size-fits-all solution.

Second, decentralized key management presents a fundamental ambiguity. Traditional approaches rely on local device storage, but this introduces challenges around device loss, backup, and cross-device access. Some projects have explored custodial services, secure enclaves, or centralized key management systems, but each approach introduces its own trade-offs in terms of usability, trust assumptions, and resilience. This challenge is further complicated by the fact that many modern authentication mechanisms (like zkLogin, Passkeys, and hardware wallets) don't depend on persistent local storage, and their APIs often restrict the use of private keys or mnemonics to signing operations only, preventing their use for encryption purposes.

More decentralized and permissionless solutions based on secure multiparty computation (MPC) have gained attention, with projects like vetKeys or Lit Protocol, distributing trust among committees of parties. While these systems remove centralized points of failure, they typically rely on a single global committee, intertwine authentication and decryption logic, and still require developers to rely on threshold assumptions about permissionless operators.

Third, there is no common language or interface for defining access control policies over encrypted data. What does it mean for someone to have "access" to ciphertext? Should access depend on token ownership, time-based conditions, or multi-party consensus? The absence of a shared policy model forces developers to reimplement this logic for each application, creating fragmentation and limiting interoperability.

Finally, the lack of reference implementations and trusted libraries discourages adoption. Developers who want to add encryption to their blockchain applications often resort to ad hoc solutions: encrypting content with ephemeral keys and sharing those keys through side channels, or implementing bespoke key management systems that don't generalize beyond isolated use cases.

2 Seal: A Decentralized Secrets Management Solution

Seal addresses these fundamental challenges by introducing a Decentralized Secrets Management (DSM) service that integrates Identity-Based Encryption (IBE) with threshold cryptography to enable fine-grained, secure access control over encrypted data. Unlike approaches where users encrypt a secret key under a threshold committee, IBE allows encryption directly to policy-defined identities (such as "user X after timestamp T" or "holder of NFT Y"), enabling dynamic access control without requiring users to manage separate encryption keys for each policy. At a high level, Sui smart contracts define who is allowed to decrypt, and a network of off-chain key servers enforces these rules by jointly managing decryption keys. Designed specifically for decentralized applications built on the Sui blockchain and/or Walrus decentralized storage, Seal supports use cases ranging from secure personal storage to complex policy-gated content while maintaining the trust-minimized properties that make blockchain applications valuable. Projects or receivers can define their desired key server committees per use-case, allowing multiple Seal committees to coexist and enabling flexible trust models tailored to specific application requirements.

Architecture. At its core, Seal introduces a clear architectural separation between policy definition and key management, and uses IBE to connect the two. First, let's recall the algorithms of IBE:

- `Setup`: Generates a master secret key msk and a master public key mpk.
- `Derive`(msk, id): Given a master secret key and an identity id (string or byte array), generates a derived secret key sk for that identity.
- `Encrypt`(mpk, id, m): Given a public key, an identity and a message, returns an encryption c.
- `Decrypt`(sk, c): Given a derived secret key and a ciphertext, compute the message m. Such a scheme is correct if for any id, m and $c =$ `Encrypt`(mpk, id, m), we have that `Decrypt(Derive` $(msk, id), c) = m$.

With these IBE algorithms in place, Seal organizes encryption and decryption around the encryptor/developer, the on-chain policy, and the key servers. The key insight is that IBE identities in Seal are not simple user identifiers, but rather combinations of policy packages and policy-specific identifiers ($packageId \| id$), enabling programmable access control. Seal SDKs provide encrypt and decrypt APIs for storing any secret using Seal.

When encrypting data, the encryptor/developer chooses a policy, identified by a package $packageId$, and constructs a policy-specific identifier id (e.g., an account, a timestamp, or other context). The IBE identity used for encryption is then $packageId \| id$. The encryptor/developer may choose *any set* of key servers and any threshold t to encrypt its data in a way that privacy is guaranteed as long as less than t of the key servers collude, and liveness is guaranteed as long as at least t key servers are responsive. We stress again that the developer can choose any set of key servers and any threshold, depending on its preferred trust assumptions.

Access control logic is defined and enforced using programmable Move smart contracts *on-chain*, ensuring transparency, composability, and verifiability. Policies deployed under package $packageId$ control the namespace of IBE identities $packageId \| *$. These policies can incorporate dynamic conditions such as timestamps, NFT ownership, voting states, or any other blockchain-accessible data.

A key server (or an MPC committee of key servers) holds the IBE master secret key msk and publishes the corresponding public key mpk. The IBE master secret key msk, and thus the ability to derive decryption keys, is never stored on-chain but resides with independent key servers operating *off-chain*. These servers may be standalone nodes operated by independent parties, or organized as decentralized permissioned or permissionless MPC committees. For example, an encryptor might select 5 servers (a mix of institutional, community-run, or MPC committees) with a threshold of 3, meaning any 3 servers can enable decryption. This flexibility allows encryptors to define their own trust assumptions rather than being forced into a one-size-fits-all model.

To decrypt, a user can request a derived key for identity $packageId \| id$. When a key server receives such a request from an authenticated user, it evaluates the policy defined by package $packageId$ for identity id using the latest on-chain state of Sui and the identity of the requesting user. If the policy grants access, the key server returns the derived key to the caller; otherwise, it rejects the request. To maintain privacy, the derived key is encrypted using an ephemeral key chosen by the caller. If a threshold was required, then the user would request keys from at least t of the n key servers, and only after obtaining derived keys from at least t servers, they could decrypt the data using the Seal SDK.

While this design allows different deployments of key servers, the first version of Seal focuses on independent key servers. It allows a key server owner to configure it as permissionless (any policy may be queried) or permissioned (only allowlisted policies may be queried). Future versions of Seal will include realizations of key servers using MPC committees, enclaves, and additional configurations.

Seal's Position in the Sui Stack. Seal integrates cleanly with existing components in the Sui blockchain stack. At the blockchain layer, Seal policies are enforced via smart contracts, leveraging chain-native identities and objects powered by Sui Move. The authentication layer provides the foundation for evaluating decryption access policies using existing identity systems without introducing new authentication primitives. This is particularly valuable for authentication mechanisms like zkLogin and Passkeys, where traditional public key-based encryption may not be feasible since users may not have published public keys before their first transaction. The storage layer, whether Sui or Walrus, handles encrypted payload storage while Seal separates key control from storage, allowing applications to maintain privacy without losing availability.

Sui Nautilus provides a framework for deploying enclaves with reproducible builds and on-chain attestation. However, enclaves face a fundamental challenge: where to securely store long-term cryptographic keys. Traditional approaches rely on external key management systems or require keys to be provisioned during enclave startup, creating security and operational complexities. Seal solves this by allowing enclaves to derive their long-term keys through policy-based access control. An enclave can request a Seal key using its attested identity (PCRs and public key), enabling use cases like encrypted data lakes that can only be processed by specific, verified enclave binaries. The flexibility of Seal policies allows the use of permissioned or permissionless enclaves.

Conversely, Nautilus can offer additional key server deployments by enabling enclave-based key servers. For example, a Seal key server running inside a Nautilus enclave can hold its master key securely while also being backed up by multiple other Seal key servers. This creates a hybrid trust model where users get the security benefits of distributed trust across multiple key servers, combined with the operational efficiency of a single enclave-based server. The enclave's attested identity ensures that only the correct binary can access the master key, while Seal's threshold cryptography provides resilience against enclave compromise or failure. Other combinations of Nautilus and Seal include use cases like poker shuffling, where enclaves can securely manage game state while leveraging Seal for key distribution. Looking forward, Seal's identity-based approach makes it uniquely suited for confidential transaction key management, especially in crypto-agile systems where users may employ multi-sig, passkeys, or zkLogin. In these scenarios, a user's public key may not be known before their first transaction, making traditional public key encryption infeasible. Seal enables such users to receive blinding factors and other cryptographic materials through policy-based access control, addressing a critical challenge for on-chain privacy in systems that support diverse authentication mechanisms.

3 Example Policies

This section showcases how Seal leverages the Move language to define expressive and enforceable access control policies. By encoding logic directly into on-chain programs, Seal enables decentralized and auditable rules for decrypting data. Policies can share code and on-chain data from other contracts, using the same toolchain.

To define an access policy in Seal, developers implement a function with the prefix `seal_approve` within a Move package. This function acts as a gatekeeper for key access: it is called by the key server to determine whether a request should be approved. The `seal_approve` function performs the logic evaluation and receives a key identifier `id`, which corresponds to the policy-specific identifier in the IBE identity $packageId \| id$ (where $packageId$ is the package identifier and id is the policy-specific identifier passed to the function), blockchain state (e.g., the current time from the `Clock` object), etc. If the result is false, the function aborts the transaction with an error code.

Policies are namespaced by packages, segregating policies defined by different contracts. Each policy defines how to construct the IBE key identifier (the `id` used for encryption and decryption). This identifier is derived from contextual data encoded in Move and typically follows a structured format, for example, a `timestamp` for time-lock encryption. The format is intentionally modular, enabling policy- or app-specific semantics for key identifiers.

In the following we present example policies. Additional use cases include event-released encryption, where decryption becomes available when specific on-chain conditions are met (e.g., when more than 100 users enroll in a program, when a game character reaches a particular level or position on the map, when liquidity falls below a threshold such as 10M, when a DAO proposal passes with sufficient votes, or when a crowdfunding campaign reaches its funding goal).

Account Based Access. This policy ensures that only the owner of account x can access the derived key for identity x. This is useful for one-to-one applications like personal storage, private messaging, or private document sharing. For example, anyone can encrypt a message that only Bob can decrypt by using Bob's account address as the IBE identity. The policy can be implemented as follows:

```
entry fun seal_approve(id: vector<u8>, ctx: &TxContext) {
    let caller_bytes = bcs::to_bytes(&ctx.sender());
    assert!(id == caller_bytes, ENoAccess);
}
```

The id argument is the IBE identity requested under the package's namespace and is the only mandatory argument. Additional Move arguments can be used by policies as needed, for example the transaction context.

NFT Gated Access. This policy ensures that only owners of a specific single-owner object type can access the keys governed by a package. This is useful, for example, for NFT-based paywalls. The policy can be implemented as follows:

```
entry fun seal_approve(id: vector<u8>, nft: &NFT) {
    // Ownership of nft is checked by the Move VM.
    // Nothing else needs to be checked here.
}
```

More sophisticated policies can incorporate fields from the NFT object (e.g., creation time for subscription renewal windows).

Time-lock Encryption. This policy allows users to encrypt data to a future point in time. Anyone can decrypt the data once the specified time has passed. This is useful, for example, for secure voting or MEV-aware workflows where data must remain private until an on-chain condition is met. The policy can be implemented as follows:

```
entry fun seal_approve(id: vector<u8>, c: &clock::Clock): bool {
    let mut prepared: BCS = bcs::new(id);
    let t = prepared.peel_u64();
    let leftovers = prepared.into_remainder_bytes();
    (leftovers.length() == 0) && (c.timestamp_ms() >= t)
}
```

Users encode the future time as the identity they use for encryption. The above policy interprets *id* as a $u64$ timestamp and compares it with Sui's on-chain clock. If the specified time has passed, access is granted. Similar policies can use Sui objects created after a particular on-chain event (e.g., when a threshold number of encrypted votes has been cast).

4 Key Server and SDKs

Stateless key server by design. As stated in Section 2, key servers only store the master secret key msk and do not need to keep state across requests or sessions. Instead, they derive decryption keys on demand from msk.

To determine whether a key request satisfies the relevant policy, the key server evaluates (off-chain, with the help of a Sui full node) the associated `seal_approve` Move function on the Sui blockchain using the latest state of the chain. The result determines whether the key can be released, ensuring that policy checks are enforced without any state changes or gas usage on-chain, since the evaluation is performed as a read-only dry run.

This approach allows for simple horizontal scalability and robustness: each key server operates independently and can verify whether a request meets the access policy without requiring shared state or user-specific metadata.

How data is encrypted. To encrypt data in Seal, the encryptor uses Seal SDKs to construct a policy-specific key identifier id (which, together with the package identifier $packageId$, forms the IBE identity $packageId \| id$ as described in Section 2) that encodes the access conditions (e.g., a timestamp for time-lock using `bcs::to_bytes(T)`). The encryptor then selects a set of key servers and a threshold value, specifying how many servers must cooperate for decryption. Using Seal SDKs, the encryptor performs a KEM/DEM encryption (detailed in Section 5): (i) generates a random symmetric key `k_sym` and encrypts the data with it using AES-256-GCM or HMAC-256-CTR, (ii) uses the TSS-BF-KEM scheme to encrypt `k_sym` under the chosen $packageId \| id$, creating separate ciphertexts for each selected key server using Boneh-Franklin IBE with BLS12-381 curves, and (iii) packages everything into a single object containing the policy identifier, threshold parameters, and the encrypted data. The resulting ciphertext can be stored anywhere since the decryption keys are held off-chain by the key servers. This design ensures that the encryptor never needs to communicate with key servers during encryption: the policy logic and key derivation happen entirely during the decryption phase.

How data is decrypted. Clients submit a policy invocation bundle to key servers consisting of a minimal Programmable Transaction Block (*PTB*, `txBytes`) that calls the policy entry function (e.g., `<package>::module::seal_approve`) with the policy's inputs, and any required references to on-chain objects used by the policy (e.g., `Clock`, `NFT`). Evaluation proceeds as follows:

1. If the policy requires the caller identity, validate an attestation bound to the request (e.g., a chain-native signature); otherwise may skip.

2. Decode `txBytes` and ensure it contains only a `MoveCall` command to a function with the prefix `seal_approve`.

3. Fetch the latest on-chain state for any objects referenced in the PTB and perform a read-only dry run using `dry_run_transaction_block`. Success is defined as "no abort" of the function; a policy failure manifests as an abort.

4. If all checks pass, the server derives and returns its IBE key share for $packageId \| id$; otherwise the request is rejected. (To maintain consistency of identities across package upgrades, the value of *packageId* used for key derivation is always set to the value of the first published version of the package.)

5 Cryptography

Seal integrates Encrypted BLS signatures and Threshold Secret-shared Identity-Based Encryption (IBE) to offer a practical solution balancing rigorous security properties with operational efficiency. We define the cryptographic primitives explicitly, including key generation, encryption, signing, and verification algorithms, and provide formal proofs of their correctness and security guarantees.

Specifically, we present an ElGamal-based Encrypted BLS signature scheme that ensures unforgeability and privacy, relying on standard cryptographic assumptions in the Random Oracle Model. Additionally, our Threshold Secret-shared IBE scheme combines Boneh-Franklin IBE, specifically the KEM-DEM hybrid encryption adaptation due to [LQ05], with threshold secret sharing to achieve strong security in the Universally Composable framework [Can01]. Our formal proofs demonstrate that Seal meets these robust security definitions, establishing a foundation for secure blockchain protocols requiring both high performance and stringent security standards.

TSS-BF-KEM. The Threshold Secret-Shared Boneh-Franklin Key Encapsulation Mechanism (TSS-BF-KEM) is the core cryptographic primitive that enables Seal's threshold decryption capabilities. It combines three key components:

- **Boneh-Franklin IBE**: Provides identity-based encryption where any string can serve as a public key, eliminating the need for public key infrastructure. In Seal, the policy-specific identifier `id` serves as the IBE identity.
- **Threshold Secret Sharing**: Uses Shamir's secret sharing to split the symmetric encryption key `k_sym` into shares, such that any t out of n shares can reconstruct the original key, but fewer than t shares reveal no information about the key.
- **Key Encapsulation Mechanism (KEM)**: Separates the encryption of the symmetric key from the encryption of the actual data, providing better efficiency and enabling the threshold sharing of the key material.

The TSS-BF-KEM process works as follows: The encryptor/developer first samples a symmetric key `k_sym`, then splits it into n shares using Shamir's secret sharing with threshold t. Each share is then encrypted using Boneh-Franklin IBE under the policy identity `id`, but using the master public key of a different key server. This creates n separate ciphertexts, one for each key server. During decryption, a user must obtain at least t of these ciphertexts and the corresponding IBE private keys from the key servers to reconstruct `k_sym` and decrypt the original data.

This design provides several key advantages: (1) **Key Independence**: Each key server maintains its own IBE master key, eliminating the need for complex distributed key generation protocols; (2) **Flexible Trust Models**: Users can choose their own set of key servers and threshold values based on their trust assumptions; (3) **Privacy**: The actual data remains encrypted with the symmetric key, and only the key shares are distributed across servers; (4) **Efficiency**: The KEM/DEM structure allows for efficient encryption of large data while keeping the threshold operations on small key material.

Encrypted BLS Signatures. While TSS-BF-KEM handles the encryption and threshold decryption of data, Encrypted BLS signatures provide the secure mechanism for key derivation in Seal's architecture. The extracted keys corresponding to identities can be viewed as partial BLS signatures on the requested identity.

The security challenge is ensuring that no eavesdropper can learn the partial signatures and aggregate them without proper authentication. To address this, Seal employs an encryption scheme where the requester sends a signed augmented ElGamal public key in their request. This augmentation enables an untrusted verifier to verify the encrypted partial signatures and aggregate them (in the case of MPC scenarios) without decrypting them, while ensuring that only the authenticated requester can decrypt the partial signatures and construct the full signature.

This design ensures that key servers can securely distribute IBE private keys while maintaining both privacy and verifiability properties.

Future Work. Building on the foundation established by our current TSS-BF-KEM and Encrypted BLS implementations, we plan to extend Seal's cryptographic primitives to support additional encryption schemes and security models. This will allow users to choose their own trust assumptions and security requirements based on their specific use cases, while maintaining Seal's core architectural principles.

Key areas of future development include: (1) **MPC Key Server**: Implementing MPC-based key server setups to provide a unified client experience with different security and liveness guarantees, and (2) **Post-Quantum Security**: Developing post-quantum secure TSS-IBE schemes to ensure long-term security against quantum attacks.

5.1 Encrypted BLS Service

In the Seal system, there are key servers which act as authorities for IBE keys. Clients obtain sub-keys which are extracted corresponding to queried ID's. We use an adaptation of the Boneh-Franklin IBE [BF01], for which the extracted sub-key for an ID is essentially a BLS [BLS01] signature on the ID. An untrusted aggregator service may help the client obtain sub-keys from several key servers. Crucially, we don't want the aggregators to have access to these sub-keys. In order to hide the sub-keys in transport from key servers to clients, the clients generate ephemeral encryption keys and send them as part of their request. The key servers hide the sub-key responses with the ephemeral key.

Definition 1 (Encrypted BLS)**.** *An* Encrypted_BLS *scheme is a tuple of algorithms* ($\texttt{keygen}$, $\texttt{ephkey}$, $\texttt{encsign}$, $\texttt{verify}$, $\texttt{decrypt}$)*:*

- $\texttt{keygen}(\lambda) \rightarrow (pk, sk)$*: takes the security parameter* λ *as input and outputs BLS keys* (pk, sk)*.*
- $\texttt{ephkey}(\lambda) \rightarrow (ephpk, ephsk)$*: outputs ephemeral keys* $(ephsk, ephpk)$*.*
- $\texttt{encsign}(sk, ephpk, ID) \rightarrow encsig$*: takes a secret key* sk*, an ephemeral public key* $ephpk$ *and an* ID *and outputs an encrypted signature* $encsig$ *on* ID*.*
- $\texttt{encverify}(pk, ephpk, ID, encsig) \rightarrow \{0, 1\}$*: takes* $(pk, ephpk, ID, encsig)$*, and checks whether the encrypted BLS signature verifies.*
- $\texttt{decrypt}(ephsk, encsig) \rightarrow sig$*: takes* $(ephsk, encsig)$*, and outputs a BLS signature* sig*.*
- $\texttt{verify}(pk, ID, sig) \rightarrow \{0, 1\}$*: takes* (pk, ID, sig)*, and checks whether the BLS signature verifies.*

The ephemeral encryptions have a design that enables aggregators to verify them without accessing the underlying signatures. The encryption is useful even when no aggregator is used. In particular, the encryption protects against a malicious key server or an eavesdropper that does a replay attack and sends the same user request to other key servers.

Definition 2 (Correctness)**.** *If* $\texttt{keygen}$*,* $\texttt{ephkey}$*,* $\texttt{encsign}$*, and* $\texttt{decrypt}$ *operations of an* Encrypted_BLS *scheme follow protocol, then* $\texttt{encverify}$ *and* $\texttt{verify}$ *both return* 1*.*

Definition 3 (EncVerify Soundness)**.** *An adversary has negligible advantage in winning the following game against a challenger:*

1. *Challenger samples* $(ephpk, ephsk) \leftarrow \texttt{ephkey}(\lambda)$ *and sends* $ephpk$ *to Adversary.*
2. *Adversary outputs* $(pk, ID, encsig)$*.*
3. *Challenger computes* $sig \leftarrow \texttt{decrypt}(ephsk, encsig)$

4. *Adversary wins if* **`verify`**(pk, ID, sig) *doesn't hold but* **`encverify`**$(pk, ephpk, ID, encsig)$ *holds.*

This definition ensures that a valid *encsig* (i.e. one that passes the `encverify` check) will necessarily contain the encryption of a valid signature *sig* (i.e. the encrypted *sig* passes the `verify` check). This implies that service is able to verify the consistency of a key server's response without getting access to the final signature.

The main security properties of Encrypted BLS, including unforgeability and privacy, are formalized later as part of the overall ideal functionality of the system in Section 5.3.

5.1.1 ElGamal-based Construction

We now describe an Encrypted BLS service based on ElGamal encryption. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be groups of prime order q, let g_1, g_2 be generators of $\mathbb{G}_1, \mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable bilinear map. Let $H : \{0,1\}^* \to \mathbb{G}_1$ be a hash-to-curve function. This construction enables aggregators to homomorphically aggregate encrypted signatures, an operation that enables MPC committees for secret shared BLS keys with a single committee public key.

`keygen`(λ): sample $sk \leftarrow \mathbb{Z}_q$ and set $pk \leftarrow g_2^{sk}$.

`ephkey`(λ): Sample $x \leftarrow \mathbb{Z}_q$ and set $ephsk \leftarrow x$ and $ephpk \leftarrow (g_1^x, g_2^x)$.

`encsign`$(sk, ephpk, ID)$: Sample $r \leftarrow \mathbb{Z}_q$. Parse $ephpk$ as $(ephpk_1, ephpk_2)$. Output $encsig \leftarrow (g_1^r, ephpk_1^r \cdot H(ID)^{sk})$.

`encverify`$(pk, ephpk, ID, encsig)$: Check if $e(encsig_2, g_2) = e(H(ID), pk) \cdot e(encsig_1, ephpk_2)$.

`decrypt`$(ephsk, encsig)$: Output $sig \leftarrow encsig_2/(encsig_1)^{ephsk}$.

`verify`(pk, ID, sig): Check if $e(sig, g_2) = e(H(ID), pk)$.

It is easy to see that the construction is correct. We will prove it's main security goal in Section 5.3, but state and prove an additional security guarantee here:

Theorem 1 (EncVerify Soundness)**.** *The ElGamal-based Encrypted BLS scheme satisfies EncVerify soundness.*

Proof. Assume that `encverify` holds with $ephkey = (g_1^x, g_2^x)$ supplied by Challenger and $(pk, ID, encsig)$ supplied by the Adversary. This means:

$$e(encsig_2, g_2) = e(H(ID), pk) \cdot e(encsig_1, g_2^x) = e(H(ID), pk) \cdot e((encsig_1)^x, g_2)$$

Therefore,

$$e(encsig_2/(encsig_1)^x, g_2) = e(H(ID), pk) \implies e(sig, g_2) = e(H(ID), pk)$$

Thus `verify` is also satisfied. □

5.2 Threshold Secret-shared IBE

Definition 4 (Threshold Secret-shared IBE)**.** *A* TSS_IBE *scheme is a tuple of algorithms (*`keygen`*,* `extract`*,* `admissible`*,* `encrypt`*,* `decrypt`*):*

`keygen`$(\lambda) \to (sk, pk)$: *Generate IBE keypairs for a server, given a security parameter.*

`extract`$(sk, ID) \to sk_{ID}$: *Generate an IBE key for ID corresponding to server sk.*

`admissible`$(\vec{sk}_{ID}, \vec{pk}, ID, t) \to \{0,1\}$: *Let n be the dimension of the vector $\vec{pk}$, which is a vector of public keys. Return 1 if $\vec{sk}_{ID}$ is also of dimension n and there are at least t indices $i \in [n]$, such that $(\vec{sk}_{ID})_i$ is an extracted secret key for ID corresponding to $(\vec{pk})_i$ and the rest $n - t$ of them equal to $\perp$.*

$\texttt{encrypt}(\vec{pk}, t, ID, m, aad) \rightarrow c = (\vec{pk}, t, c')$**:** *Return the encryption of a message m with ID characterizing the target recipient. We assume that c includes the ciphertext c', the vector of public keys used to construct it, $\vec{pk}$ and the intended threshold t. The aad field is an optional way to uniquely label ciphertexts. This is helpful in situations where we want to prevent replay of encryptions, such as encrypted votes.*

$\texttt{decrypt}(\vec{sk}_{ID}, ID, c, aad) \rightarrow m$**:** *Decrypt a ciphertext $c.c'$ given an admissible subset of the user secret keys for the corresponding key servers, that is, admissible$(\vec{sk}_{ID}, c.\vec{pk}, ID, c.t) = 1$.*

Definition 5 (Correctness of TSS_IBE)**.** *Correctness holds if for all $\lambda, n, t, \vec{pk}$ with $|\vec{pk}| = n, ID, \vec{sk}_{ID}$, the following holds: if*

$$\begin{array}{ccl} (sk_i, pk_i) & \leftarrow & \textit{keygen}(\lambda), \text{ for all } i \in [n] \\ c & \leftarrow & \textit{encrypt}(\vec{pk}, t, ID, m, aad), \text{ such that } (\vec{pk})_i \in pk_{[n]} \text{ for all } i \in [n] \\ sk_{ID,i} & \leftarrow & \textit{extract}(sk_i, ID), \text{ for all } i \in [n] \\ 1 & \leftarrow & \textit{admissible}(\vec{sk}_{ID}, \vec{pk}, ID, t) \\ m' & \leftarrow & \textit{decrypt}(\vec{sk}_{ID}, ID, c, aad) \end{array}$$

then we must have $m = m'$.

5.2.1 Construction TSS-BF-KEM-CCA

In this section, we build a TSS_IBE based on Boneh-Franklin IBE, a threshold secret sharing scheme TSS over $\mathbb{F}_{2^\lambda}$, a symmetric labeled encryption scheme SYMENC with message and ciphertext domains $\{0,1\}^*$, and Random Oracles $H_1 : \{0,1\}^* \rightarrow \mathbb{G}_1, H_2 : \{0,1\}^* \rightarrow \{0,1\}^\lambda, H_3 : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$.

KeyGen(λ)**:** Given bilinear groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, g_1, g_2, e)$,

- Sample $sk \leftarrow \mathbb{Z}_q^n$.
- Compute $pk \leftarrow g_2^{sk}$.
- Output (pk, sk).

Extract(i, sk_i, ID)**:** Output $H_1(ID)^{sk_i}$.

Encrypt$(\vec{pk}, t, ID, m, aad)$**:** Given a vector of public keys $\vec{pk}$, which might repeat some pk's to represent weight, a threshold t, an ID, a plaintext m, and an additional authenticated data field aad,

- Sample $k \leftarrow \{0,1\}^\lambda, r \leftarrow \mathbb{Z}_q$
- Sample $(k_1, \ldots, k_n) \leftarrow \mathsf{TSS}.\texttt{share}(k, n, t)$
- Compute $nonce \leftarrow g_2^r$
- Compute $c_i' = k_i \oplus H_2(i, (\vec{pk})_i, H_1(ID), nonce, e(H_1(ID), (\vec{pk})_i^r))$ for all $i \in [n]$
- $(k_r, k_{sym}) \leftarrow H_3(k, \vec{pk}, t, c'_{[n]})$.
- $c_r \leftarrow r \oplus k_r$
- $sem = \mathsf{SYMENC}.\texttt{encrypt}(k_{sym}, m, aad)$ where aad is the encryption label.
- Output $(\vec{pk}, t, nonce, c_r, c'_{[n]}, sem)$

Admissible$(\vec{sk}_{ID}, \vec{pk}, ID, t)$**:** Given a set of secret keys extracted for ID,

- Set count $= 0$
- For all $i \in [|\vec{pk}|]$, do:
  - if $((\vec{sk}_{ID})_i \neq \perp)$, then:
    - if $e((\vec{sk}_{ID})_i, g_2) \neq e(H_1(ID), (\vec{pk})_i)$, then return 0.
    - else set $count \leftarrow count + 1$
- If $count \neq t$, then return 0, else return 1

Decrypt$(\vec{sk}_{ID}, ID, (\vec{pk}, t, nonce, c_r, c'_{[n]}, sem), aad)$**:** Given a set of t secret keys extracted for ID, a ciphertext, and an additional authenticated data field aad,

- If $\texttt{admissible}(\vec{sk}_{ID}, \vec{pk}, ID, t) = 0$, then return $\perp$.
- Compute $k_i \leftarrow c'_i \oplus H_2(i, (\vec{pk})_i, H_1(ID), nonce, e(sk_{ID,i}, nonce))$, for all $(\vec{sk}_{ID})_i \neq \perp$
- Compute $k \leftarrow \mathsf{TSS}.\texttt{reconstruct}(\{k_i\}_{(\vec{sk}_{ID})_i \neq \perp}, n, t)$
- $(k_r, k_{sym}) \leftarrow H_3(k, \vec{pk}, t, c'_{[n]})$.
- Compute $r \leftarrow c_r \oplus k_r$
- Assert $nonce = g_2^r$
- Compute $k_{[n]} \leftarrow \mathsf{TSS}.\texttt{extend}(\{k_j\}_{(\vec{sk}_{ID})_j \neq \perp})$
- Assert $k_i = c'_i \oplus H_2(i, (\vec{pk})_i, H_1(ID), nonce, e(H_1(ID), (\vec{pk})_i^r))$, for all $(\vec{sk}_{ID})_i = \perp$
- Output $\mathsf{SYMENC}.\texttt{decrypt}(k_{sym}, sem, aad)$

In particular, the SYMENC is implemented in Seal with two options: HMAC-CTR and AES-GCM as follows:

Algorithm 1 AuthenticatedEncrypt

```
 1: Input: $M \in \{0,1\}^*$, $aad \in \{0,1\}^*$, $k \in \{0,1\}^\lambda$, $mode \in \{\mathsf{AES}, \mathsf{HMAC}\}$
 2: Output: $sem$
 3: if $mode = \mathsf{AES}$ then
 4:     $sem \leftarrow \mathsf{AES\_256\_GCM}.encrypt(k, m, aad)$          ▷ AES-GCM encryption
 5: else
 6:     Parse $m_{[l]} \leftarrow M$
 7:     for $i \in [l]$ do
 8:         $c_i \leftarrow m_i \oplus \mathsf{HMAC\_SHA3\_256}(k, \text{“}enc\text{”} \mid i)$
 9:     end for                                                     ▷ Stream cipher encryption
10:     $mac \leftarrow \mathsf{HMAC\_SHA3\_256}(k, \text{“}mac\text{”} \mid aad \mid c_{[l]})$       ▷ Compute MAC
11:     $sem \leftarrow (c_{[l]}, mac)$
12: end if
13: return $sem$
```

Algorithm 2 AuthenticatedDecrypt

```
 1: Input: $sem$, $aad \in \{0,1\}^*$, $k \in \{0,1\}^\lambda$, $mode \in \{\mathsf{AES}, \mathsf{HMAC}\}$
 2: Output: $M \in \{0,1\}^*$ or $\perp$
 3: if $mode = \mathsf{AES}$ then
 4:     $M \leftarrow \mathsf{AES\_256\_GCM}.decrypt(k, sem, aad)$          ▷ AES-GCM decryption
 5:     if $M = \perp$ then
 6:         return $\perp$                                              ▷ Authentication failed
 7:     end if
 8: else
 9:     Parse $(c_{[l]}, mac) \leftarrow sem$
10:     $mac' \leftarrow \mathsf{HMAC\_SHA3\_256}(k, \text{“}mac\text{”} \mid aad \mid c)$         ▷ Recompute MAC
11:     if $mac' \neq mac$ then
12:         return $\perp$                                              ▷ Authentication failed
13:     end if
14:     for $i \in [l]$ do
15:         $m_i \leftarrow c_i \oplus \mathsf{HMAC\_SHA3\_256}(k, \text{“}enc\text{”} \mid i)$
16:     end for                                                     ▷ Stream cipher decryption
17:     $M \leftarrow m_{[l]}$
18: end if
19: return $M$
```

Theorem 2. *TSS-BF-KEM-CCA satisfies correctness.*

Proof. The correctness of IBE decryption follows immediately from Boneh-Franklin. Decrypting with an admissible set of extracted secret keys allows computation of t shares of n Shamir secret shares of k. Thus k can be correctly reconstructed.

Since *nonce* is a binding commitment to r and the share consistency checks mirror the encryption process, these checks will also pass.

Now given the decryption correctness of SYMENC and the correct reconstruction of k, we get the correct decryption to the plaintext m. □

5.3 Ideal Functionality and Protocol

This section presents the ideal functionality and protocol for verifiable threshold secret sharing identity-based encryption (vetssIBE), which enables secure key derivation and encryption in a distributed setting. The construction allows multiple servers to jointly derive encryption keys for specific identities while maintaining security guarantees. The protocol Π_{tssIBE} is an integrated standalone protocol that directly implements the vetssIBE functionality. We present the ideal functionality specification, the concrete protocol implementation, and a simulator that demonstrates UC-security under the co-BDH assumption in the Random Oracle Model.

The ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ provides verifiable threshold secret sharing identity-based encryption (vetssIBE) in the UC framework. It enables multiple servers to individually derive encryption keys for specific identities while maintaining security guarantees. The functionality supports three main operations: (1) **Key Derivation**: Servers can derive encryption keys for specific identities on behalf of users, (2) **Encryption**: Any party can encrypt messages for specific identities using given public keys, and (3) **Decryption**: Users can decrypt messages if they have derived the necessary keys from the required servers. The design ensures that only users with proper key derivations can decrypt messages, while maintaining verifiability and security properties.

The functionality maintains several key data structures to track the state of the system:

- **Honest MPKs**: MPK stores the public keys of the honest servers.
- **Key Derivation Tracking**: EKD tracks which identities are currently being derived for each user-server pair, enabling proper simulation in case of corrupt user requesting for a key that was used in a simulated encryption.
- **Ciphertext Storage**: $CTXT$ stores all encrypted messages with their associated metadata (public keys, threshold, ciphertext, identity, message, AAD) for proper decryption and consistency.

Intuition behind the functionality. The ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ formalizes a core security property of the system that ensures confidentiality in a distributed threshold setting. At its heart, the functionality guarantees that for any identity for which no corrupt party has obtained an extracted key, the adversary cannot learn the underlying message, even when observing encrypted ciphertexts. This security property is fundamental to the system's design and forms the basis for secure key distribution in multi-server environments.

The security guarantee is enforced via a TSS-BF-KEM (Threshold Secret Sharing Boneh-Franklin Key Encapsulation Mechanism) scheme. This construction operates in a t-out-of-n threshold setting, where decryption requires possession of at least t out of n extracted keys from different servers. The threshold structure ensures that if corrupt users have not obtained the required minimum number of extracted keys, then ciphertexts encrypted to the full set of n servers cannot be decrypted, even by a coalition of corrupt parties. This provides robust security against partial corruption scenarios where some but not all servers may be compromised. Our Boneh-Franklin IBE modification provides non-malleability, ensuring that adversaries cannot transform valid ciphertexts into other valid ciphertexts for related messages or identities. This property is captured by the ideal functionality's requirement that decryption only succeeds for ciphertexts that were legitimately created and stored in $CTXT$. The functionality maintains state tracking through several interconnected data structures, as explained above. A central ciphertext table stores all encrypted message pairs along with their associated metadata, including the public keys used for encryption, the target identity, the original message, and any additional authenticated data. This table serves as the record for determining legitimate decryption operations and prevents inconsistencies that could arise.

A particularly challenging scenario in the security proof arises when a corrupt user requests a key that was previously used in a simulated encryption operation. In this case, the functionality must reveal the decryption to the simulator to maintain indistinguishability, but this creates a complex technical challenge: the simulator must be able to equivocate previously simulated ciphertexts to open correctly to the newly supplied messages. This equivocation process requires sophisticated programming of the random oracles and careful management of the symmetric encryption components to ensure that the previously generated ciphertexts can be made to decrypt to the correct messages without the adversary detecting any inconsistencies in the simulation. This functionality is an integration of the vetKeys [CCN+23] $\mathcal{F}_{bls}$ and $\mathcal{F}_{vetibe}$ functionalities and adapted to independent signers and our TSS-BF-KEM scheme, rather than MPC-based signers. The formal specification of the ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ is given in Figure 1.

Ideal Functionality $\mathcal{F}_{\mathsf{tssIBE}}$

- On $(sid, \texttt{"init"})$ from honest $\mathcal{S}_i$:

  Send $(sid, \texttt{"init"}, \mathcal{S}_i)$ to Sim. If mpk_i isn't defined, wait for a response mpk_i from Sim and store mpk_i. If MPK is not initialized, initialize $EKD, MPK, CTXT \leftarrow \emptyset$. Add mpk_i to MPK. Output $(sid, \texttt{"output-mpk"}, mpk_i)$ to $\mathcal{S}_i$.

- On $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id, \mathcal{U})$ from honest $\mathcal{S}_i$:

  Add $(mpk_i, id, \mathcal{U})$ to EKD and send $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id, \mathcal{U})$ to Sim.

  Output $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id)$ to $\mathcal{U}$.

- On $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, m, aad)$ from $\mathcal{P}$:

  Let $I = \{i \in [|\vec{mpk}|] : (\vec{mpk})_i \notin MPK \text{ OR } \exists \text{ corrupt } \mathcal{U}' \text{ with } (mpk_i, id, \mathcal{U}') \in EKD\}$.

  If $|I| \geq t$, send $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, m, aad)$ to Sim.

  Otherwise send $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, |m|, aad)$ to Sim.

  Wait for response C from Sim such that $\nexists(\vec{mpk}, t, C, id, \cdot, aad) \in CTXT$. Add $(\vec{mpk}, t, C, id, m, aad)$ to $CTXT$ and output $(sid, \texttt{"encrypt"}, \vec{mpk}', id, m, aad, C)$ to $\mathcal{P}$.

- On $(sid, \texttt{"decrypt"}, \vec{\mathcal{S}}, \vec{mpk}, t, id, C, aad)$ from $\mathcal{U}$:

  Let $S_{\mathcal{U}} = \{\mathcal{S}_i \in \vec{\mathcal{S}} : (mpk_i, id, \mathcal{U}) \in EKD\}$ and $S_{\text{corrupt}} = \{\mathcal{S}_i \in \vec{\mathcal{S}} : \exists \text{ corrupt } \mathcal{U}' \text{ with } (mpk_i, id, \mathcal{U}') \in EKD \text{ OR } mpk_i \notin MPK\}$.

  If $\mathcal{U}$ is honest and $|S_{\mathcal{U}}| < t$, then ignore.

  Else if $\mathcal{U}$ is corrupt and $|S_{\text{corrupt}}| < t$, then ignore.

  Else if $\exists(\vec{mpk}, t, C, id, m, aad) \in CTXT$, then output $(sid, \texttt{"decrypt"}, C, id, m, aad)$ to $\mathcal{U}$.

  Else, send $(sid, \texttt{"decrypt"}, \vec{\mathcal{S}}, \vec{mpk}, t, C, id, aad)$ to Sim and await a response m. Add $(\vec{mpk}, t, C, id, m, aad)$ to $CTXT$ and output $(sid, \texttt{"decrypt"}, C, id, m, aad)$ to $\mathcal{U}$.

Figure 1: The ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ for verifiable threshold secret sharing identity-based encryption (vetssIBE) in the UC framework. The functionality supports key derivation, encryption, and decryption operations while maintaining security guarantees through state tracking in MPK, EKD, and $CTXT$.

Protocol Construction. The protocol Π_{tssIBE} is constructed by composing two cryptographic primitives: *Encrypted BLS* (described in Section 5.1.1) and *TSS-BF-KEM* (Threshold Secret Sharing Boneh-Franklin Key Encapsulation Mechanism, described in Section 5.2.1). While these components serve distinct purposes, they are integrated into a unified protocol rather than being composed as separate hybrid functionalities. This unified design simplifies the security analysis by avoiding the complexity of composing multiple ideal functionalities, and it allows the extracted keys from Encrypted BLS to be directly used as IBE secret keys in TSS-BF-KEM without additional transformation layers.

The *Encrypted BLS* primitive is used in the "ekderive" operation to securely deliver IBE extracted keys from servers to users. Specifically, when a user $\mathcal{U}$ requests an extracted key for identity id from server $\mathcal{S}_i$, the protocol follows the ElGamal-based Encrypted BLS construction: (1) $\mathcal{U}$ generates an ephemeral transport key pair $tpk = (g_1^{tsk}, g_2^{tsk})$ where tsk is sampled from $\mathbb{Z}_q$, (2) $\mathcal{S}_i$ computes the IBE extracted key as $\sigma = H_1(id)^{sk_i}$ (which is essentially a BLS signature on id), (3) $\mathcal{S}_i$ encrypts this key using ElGamal encryption to produce $es = (g_1^r, tpk_1^r \cdot \sigma)$ where r is sampled from $\mathbb{Z}_q$, and (4) $\mathcal{U}$ decrypts to recover $K_{i,id} = e_2/e_1^{tsk} = \sigma$. This encrypted transport mechanism ensures that the extracted keys remain confidential during transmission, protecting against eavesdroppers and malicious aggregators, while enabling verifiability through the `encverify` check.

The *TSS-BF-KEM* primitive handles the actual encryption and threshold decryption of messages. The "encrypt" operation directly invokes $\mathsf{TSSBFKEM.encrypt}(\vec{mpk}, t, id, m, aad)$ to produce a ciphertext C that encapsulates a symmetric key k using a threshold secret sharing scheme, then encrypts the message m under that key. The "decrypt" operation uses the extracted keys $\vec{K_{id}} = \{K_{i,id}\}_{i\in[n]}$ (obtained via the Encrypted BLS mechanism) to invoke $\mathsf{TSSBFKEM.decrypt}(\vec{K_{id}}, id, C, aad)$, which reconstructs the symmetric key from at least t shares and decrypts the message.

The integration of these primitives is seamless: the extracted keys $K_{i,id}$ obtained through Encrypted BLS are exactly the IBE secret keys $sk_{ID,i} = H_1(ID)^{sk_i}$ required by TSS-BF-KEM for decryption. This design ensures that users who have successfully derived keys from at least t servers can decrypt ciphertexts, while maintaining security guarantees even when some servers or aggregators are corrupt. The protocol Π_{tssIBE} is therefore an *integrated standalone protocol* that directly implements the vetssIBE functionality, rather than being a hybrid composition of separate ideal functionalities. The complete protocol specification is given in Figure 2.

Protocol Π_{tssIBE}

- On $(sid, \texttt{"init"})$:

  Server $\mathcal{S}_i$ samples $sk \leftarrow \mathbb{Z}_q$, computes $pk \leftarrow g_2^{sk}$, and stores (pk, sk). Sends $(sid, \texttt{"pk"}, pk)$ and outputs $(sid, \texttt{"output-mpk"}, pk)$.

- On $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id, \mathcal{U})$:

  $\mathcal{U}$ samples $tsk \leftarrow \mathbb{Z}_q$ and stores tsk, computes $tpk \leftarrow (g_1^{tsk}, g_2^{tsk})$ and sends $(sid, \texttt{"tpk"}, tpk)$ to $\mathcal{S}_i$.

  $\mathcal{S}_i$ parses $(tpk_1, tpk_2) \leftarrow tpk$ and checks whether $e(tpk_1, g_2) = e(g_1, tpk_2)$; if not, it ignores this input. Otherwise it computes $\sigma \leftarrow H(id)^{sk}$, samples $r \leftarrow \mathbb{Z}_q$, computes $es \leftarrow (g_1^r, tpk_1^r \cdot \sigma)$, and sends $(sid, \texttt{"encsig"}, id, tpk, es)$ to $\mathcal{U}$.

  $\mathcal{U}$ parses $(e_1, e_2) \leftarrow es$ and sets $K_{i,id} \leftarrow e_2/e_1^{tsk}$. It stores $(mpk_i, id, K_{i,id})$ in its state, and outputs $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id)$.

- On $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, m, aad)$:

  $\mathcal{P}$ computes $C \leftarrow \mathsf{TSSBFKEM.encrypt}(\vec{mpk}, t, id, m, aad)$ and outputs $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, m, aad, C)$.

- On $(sid, \texttt{"decrypt"}, \vec{\mathcal{S}}, \vec{mpk}, t, id, C, aad)$:

  $\mathcal{U}$ checks if it has $(mpk_i, id, K_{i,id})$ in its state for all $\mathcal{S}_i \in \vec{\mathcal{S}}$. If not, it ignores.

  Else, it computes $m \leftarrow \mathsf{TSSBFKEM.decrypt}(\vec{K_{id}}, id, C, aad)$ and outputs $(sid, \texttt{"decrypt"}, C, id, m, aad)$ to $\mathcal{U}$.

Figure 2: The protocol Π_{tssIBE} implementing verifiable threshold secret sharing identity-based encryption (vetssIBE). The protocol integrates Encrypted BLS for secure key derivation and TSS-BF-KEM for threshold encryption and decryption.

5.4 Security Proof

We now establish the security of the protocol Π_{tssIBE} by proving that it UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ under standard cryptographic assumptions. The proof proceeds in two main steps: first, we construct a simulator Sim that interacts with the ideal functionality and the environment, demonstrating that any attack in the real world can be simulated in the ideal world. Second, we show that any distinguisher between the real and ideal executions can be used to solve the co-BDH (co-Bilinear Diffie-Hellman) problem, which is the same assumption used in Boneh-Franklin IBE.

Theorem 3 (UC-Security of vetssIBE). *The protocol Π_{tssIBE} UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ in the Random Oracle Model. That is, for any PPT environment $\mathcal{E}$ and any PPT adversary $\mathcal{A}$ corrupting a subset of parties, there exists a PPT simulator Sim such that:*

$$\Pr[REAL_{\Pi_{\mathsf{tssIBE}},\mathcal{A},\mathcal{E}}(\lambda) = 1] - \Pr[IDEAL_{\mathcal{F}_{\mathsf{tssIBE}},\mathsf{Sim},\mathcal{E}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{negl}(\lambda)$ is a negligible function in the security parameter λ.

Proof. The proof proceeds by constructing a simulator Sim that interacts with the ideal functionality $\mathcal{F}_{\mathsf{tssIBE}}$ and the environment $\mathcal{E}$ in the ideal world, such that no PPT environment can distinguish between the real execution of Π_{tssIBE} and the ideal execution with $\mathcal{F}_{\mathsf{tssIBE}}$.

Consider the following simulator that interacts with $\mathcal{E}$ and $\mathcal{F}_{\mathsf{tssIBE}}$ in the ideal world.

- On $(sid, \texttt{"init"}, \mathcal{S}_i)$ from $\mathcal{F}_{\mathsf{tssIBE}}$:

  If this is the first such call, Sim samples $sk_i \leftarrow \mathbb{Z}_q$ and computes $pk_i \leftarrow g_2^{sk_i}$. It stores (pk_i, sk_i) and returns pk_i to $\mathcal{F}_{\mathsf{tssIBE}}$. Otherwise, it returns the public key pk_i stored in its state.

- On $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, m, aad)$ from $\mathcal{F}_{\mathsf{tssIBE}}$:

  Sim runs the honest $\texttt{"encrypt"}$ interface of Π_{tssIBE} to produce a ciphertext C and sends C back to $\mathcal{F}_{\mathsf{tssIBE}}$.

- On $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, |m|, aad)$ from $\mathcal{F}_{\mathsf{tssIBE}}$:

  Sim executes as follows:

  – sample $r \leftarrow \mathbb{Z}_q$
  – compute $nonce \leftarrow g_2^r$
  – sample $c_r \leftarrow \{0,1\}^\lambda$
  – sample $c'_i \leftarrow \{0,1\}^\lambda$ for $i \in [n]$
  – sample $k_{sym} \leftarrow \mathsf{SYMENC}.\texttt{keygen}()$
  – compute $sem \leftarrow \mathsf{SYMENC}.\texttt{encrypt}(k_{sym}, 0^{|m|}, aad)$
  – set $C \leftarrow (\vec{pk}, t, nonce, c_r, c'_{[n]}, sem)$,
  – and add (C, id, aad, r) to $CTXT$.

  Finally, it sends C to $\mathcal{F}_{\mathsf{tssIBE}}$.

- On $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id, \mathcal{U})$ from $\mathcal{F}_{\mathsf{tssIBE}}$ for an honest user $\mathcal{U}$:

  Sim lets $\mathcal{S}_i$, and $\mathcal{U}$ follow the $\texttt{"ekderive"}$ interface on Π_{tssIBE} protocol.

- On $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id, \mathcal{U})$ from $\mathcal{F}_{\mathsf{tssIBE}}$ for a corrupt user $\mathcal{U}$:

  Sim receives $(sid, \texttt{"tpk"}, tpk)$ from $\mathcal{U}$, parses $(tpk_1, tpk_2) \leftarrow tpk$ and checks whether $e(tpk_1, g_2) = e(g_1, tpk_2)$; if not, it ignores this input. Otherwise, Sim computes $\sigma \leftarrow H_1(id)^{sk_i}$, samples $r \leftarrow \mathbb{Z}_q$, computes $es \leftarrow (g_1^r, tpk_1^r \cdot \sigma)$, and sends $(sid, \texttt{"encsig"}, id, tpk, es)$.

  Sim goes over all $(C, id, aad, r) \in CTXT$ to

  – parse $C \rightarrow (\vec{pk}, t, nonce, c_r, c'_{[n]}, sem)$,
  – derive the server set $\vec{\mathcal{S}}$ corresponding to the public keys in $\vec{pk}$,

  - send an input $(sid, \texttt{"decrypt"}, \vec{\mathcal{S}}, \vec{pk}, t, id, C, aad)$ to $\mathcal{F}_{\mathsf{tssIBE}}$ to obtain a response $(sid, \texttt{"decrypt"}, C, id, m, aad)$,
  - program random k_{sym}, such that $\mathsf{SYMENC.decrypt}(k_{sym}, sem, aad) = m$
  - sample $k \leftarrow \{0,1\}^{\lambda}$
  - sample $k_{[n]} \leftarrow \mathsf{TSS.share}(k, n, t)$
  - compute $k_r \leftarrow r \oplus c_r$
  - compute $T_i \leftarrow e(H_1(id), (\vec{pk})_i)^r$ for all $i \in [n]$
  - program the maps for random oracles H_2, and H_3 as

$$\forall i \in [n] : H_2[i, (\vec{pk})_i, H_1[id], nonce, T_i] \leftarrow c'_i \oplus k_i$$

$$H_3[k, \vec{pk}, t, c'_{[n]}] \leftarrow (k_r, k_{sym})$$

  - and to delete (C, id, aad, r) from $CTXT$.

  Note that programming the random oracles this way always works unless Sim aborted earlier. Also note that, by programming the random oracles in this way, the "decrypt" interface of Π_{tssIBE} correctly decrypts C under id to m.

- On $(sid, \texttt{"decrypt"}, \vec{\mathcal{S}}, \vec{pk}, t, C, id, aad)$ from $\mathcal{F}_{\mathsf{tssIBE}}$:

  Sim uses the "decrypt" interface on Π_{tssIBE} protocol to decrypt C under id. If successful, it sends m back to $\mathcal{F}_{\mathsf{tssIBE}}$. Otherwise, it sends $\perp$.

co-BDH reduction. We show that a distinguisher $\mathcal{D}$ that can distinguish between the real execution of Π_{tssIBE} and the simulator Sim can be turned into an algorithm $\mathcal{B}$ that solves the co-BDH problem. On input a co-BDH challenge $(U_1, V_1, V_2, W_2) \in \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_2$ with the goal of outputting $e(g_1, g_2)^{\alpha\beta\gamma}$ for hidden α, β, γ such that $U_1 = g_1^{\alpha}$, $V_1 = g_1^{\beta}$, $V_2 = g_2^{\beta}$, and $W_2 = g_2^{\gamma}$, algorithm $\mathcal{B}$ runs $\mathcal{E}$ and $\mathcal{A}$ in an environment like the one provided by Sim above, except that:

- For the init of each server $\mathcal{S}_i$, it uses $mpk_i \leftarrow V_2^{\theta_i}$ for a random $\theta_i \leftarrow \mathbb{Z}_q$.
- It simulates the random oracle $H_1(id)$, by choosing $r \leftarrow \mathbb{Z}_q$, storing $H_1[id] \leftarrow (g_1^r, r)$, and returning g_1^r, except for one randomly guessed id^*, it returns $H_1[id^*] \leftarrow U_1$ instead.
- On input $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id, \mathcal{U})$ for any user where $id \neq id^*$:
  - $\mathcal{B}$ receives $(sid, \texttt{"tpk"}, tpk)$ from $\mathcal{U}$,
  - parses $(tpk_1, tpk_2) \leftarrow tpk$ and checks whether $e(tpk_1, g_2) = e(g_1, tpk_2)$; if not, it ignores this input.
  - Otherwise it looks up the stored $(g_1^r, r) \leftarrow H_1[id]$, computes $\sigma \leftarrow V_1^{r\theta_i}$, samples $r' \leftarrow \mathbb{Z}_q$, computes $es \leftarrow (g_1^{r'}, tpk_1^{r'} \cdot \sigma)$, and sends $(sid, \texttt{"encsig"}, id, tpk, es)$.

  If $\mathcal{U}$ is corrupt, then $\mathcal{B}$ patches the ciphertexts just as Sim does.
- On input $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id^*, \mathcal{U})$: $\mathcal{B}$ aborts if $\mathcal{U}$ is corrupt. Otherwise, it does the following:
  - sample $u \leftarrow \mathbb{Z}_q$ and computes $\tilde{tpk} \leftarrow (V_1 \cdot g_1^u, V_2 \cdot g_2^u)$,
  - send $(sid, \texttt{"tpk"}, \tilde{tpk})$ from $\mathcal{U}$.
  - sample $s \leftarrow \mathbb{Z}_q$, computes $es \leftarrow (g_1^s \cdot U_1^{-\theta_i},\ g_1^{us} \cdot U_1^{-\theta_i u} \cdot V_1^{-\theta_i s})$,
  - send $(sid, \texttt{"encsig"}, id, \tilde{tpk}, es)$ to $\mathcal{F}_{\mathsf{tssIBE}}$.
- On input $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, m, aad, \mathcal{P})$:

  On one randomly chosen input $(sid, \texttt{"encrypt"}, \vec{mpk}, t, id, \ell, aad)$ from $\mathcal{F}_{\mathsf{tssIBE}}$, it gives up if $id \neq id^*$; otherwise, it sets $C^* \leftarrow (\vec{mpk}, t, nonce, c_r, c'_{[n]}, sem)$ where
  - $nonce \leftarrow W_2$,

  - $c_r \leftarrow \{0,1\}^\kappa$ is a uniform κ-bit string,
  - $\forall i \in [n]:\ c'_i \leftarrow \{0,1\}^\kappa$ independent uniform strings,
  - $sem \leftarrow \{0,1\}^\ell$ an independent uniform string of length ℓ,

  adds (C^*, id^*, aad, s) to $CTXT$ for $s \leftarrow \{0,1\}^\kappa$, and responds C^* to $\mathcal{F}_{\mathsf{tssIBE}}$.

  For all other inputs, $\mathcal{B}$ follows the same strategy as Sim does.

- On $(sid, \texttt{"decrypt"}, \vec{mpk}, t, C = (\vec{mpk}, t, nonce, c_r, c'_{[n]}, sem), id, aad)$ from $\mathcal{F}_{\mathsf{tssIBE}}$, $\mathcal{B}$ does not decrypt using any secret key, but instead checks whether there exist $((k, \vec{pk}, t, c'_{[n]}), (k_r, k_{sym})) \in H_3$, such that $g_2^{k_r \oplus c_r} = nonce$. Return $\perp$ if no such entry. If so, then it computes $r \leftarrow c_r \oplus k_r$, decrypts all the encrypted shares with r and then checks if the key shares form a valid secret sharing of k. Otherwise, it returns $\perp$. If all checks pass, it returns $m \leftarrow \mathsf{SYMENC}.\texttt{decrypt}(k_{sym}, sem, aad)$ to $\mathcal{F}_{\mathsf{tssIBE}}$; if not, or if no such entry was found, it returns $m \leftarrow \perp$ to $\mathcal{F}_{\mathsf{tssIBE}}$.

The only way that this decryption method produces a different outcome than the real decryption of Π_{tssIBE} is if, during decryption, no entry $((k, \vec{mpk}, t, c'_{[n]}), (k_r, k_{sym}))$ exists in H_3, but later a new random-oracle query to H_3 on $(k, \vec{mpk}, t, c'_{[n]})$ creates exactly such an entry with $g_2^{k_r \oplus c_r} = nonce$. With at most q_D decryption queries and q_H random-oracle queries, this happens with probability at most $q_H\, q_D / 2^\kappa$.

To cause the original simulator Sim to abort, $\mathcal{A}$ must make a query to H_2 so that there exists $(C, id, aad, r) \in CTXT$ with

$$H_2[i^*, (\vec{mpk})_{i^*}, H_1[id], nonce,\ e(H_1(id), (\vec{mpk})_{i^*})^r] \text{ is queried.}$$

Algorithm $\mathcal{B}$'s strategy is that the offending query occurs for the tuple $(C^*, id^*, aad, s) \in CTXT$, in which case we would have that

$$T^{1/\theta_i} := e(H_1(id^*), (\vec{mpk})_i)^{\log_{g_2} nonce} = e(U_1, V_2)^{\log_{g_2} W_2} = e(g_1, g_2)^{\alpha\beta\gamma},$$

which is the solution to $\mathcal{B}$'s co-BDH challenge. Since $\mathcal{B}$ cannot test which random-oracle query is the offending one, it simply outputs the key of a randomly chosen entry, scaled by its corresponding θ_i, among its H_2 queries, giving it a probability of at least $1/q_H$ to guess correctly. That query also has to trigger abortion for the tuple $(C^*, id^*, aad, s) \in CTXT$, however, which happens with probability $1/q_E$ if $\mathcal{B}$ didn't give up prematurely.

There are two reasons that cause $\mathcal{B}$ to give up prematurely. The first is if an honest server $\mathcal{S}_i$ calls $\mathcal{F}_{\mathsf{tssIBE}}$ with $(sid, \texttt{"ekderive"}, \mathcal{S}_i, id^*, \mathcal{U})$ for a corrupt $\mathcal{U}$. The other reason for $\mathcal{B}$ to give up early is if the randomly chosen "encrypt" input is for $id \neq id^*$. This can indeed happen, but since $\mathcal{A}$'s view is independent of $\mathcal{B}$'s choice of id^* as long as it doesn't give up, we have a probability at least $1/q_H$ that $id = id^*$.

Overall, $\mathcal{B}$ therefore outputs the solution to the co-BDH problem with probability at least $1/(q_E \cdot q_H^2)$. □

On the symenc scheme. For the UC proof we require programmability of the symmetric encryption scheme. As described in Section 5.2.1, Seal offers two different symmetric encryption schemes: HMAC-CTR-based and AES-GCM-based. The HMAC-CTR-based scheme can be written succinctly as:

$$sem \leftarrow \mathsf{SYMENC}.\texttt{encrypt}(k, M = m_{[l]}, aad) = (c_i = m_i \oplus H(k, \texttt{"enc"}|i)))_{i\in[l]}, H(k, \texttt{"mac"}|aad|c_{[l]}))$$

To program this to decrypt to $M' = m'_{[l]}$ under an independently random key k', we program the RO as follows:

$$H(k', \texttt{"enc"}|i) \leftarrow m'_{[i]} \oplus m_{[i]} \oplus H(k, \texttt{"enc"}|i)$$

$$H(k', \texttt{"mac"}|aad|c_{[l]})) \leftarrow H(k, \texttt{"mac"}|aad|c_{[l]}))$$

For the AES-GCM-based scheme, we are not able to prove UC security due to the lack of programmability. To maintain UC security, the simulator must be able to equivocate: it must retroactively program the symmetric encryption scheme so that the previously simulated ciphertext C decrypts to the real message m under the newly revealed key material. In contrast, a game-based security definition can avoid this programmability requirement by imposing restrictions on the adversary's behavior. Specifically, we can define security so that the adversary *loses* the security game if it ever requests an extracted key for an identity *id* that was used in any encryption challenge query. This restriction is natural in the context of identity-based encryption: it corresponds to the standard adaptive chosen ciphertext attack (IND-ID-CCA) security notion, where the adversary cannot query the key extraction oracle for the challenge identity. Under such a definition, the simulator never needs to equivocate simulated ciphertexts, because the adversary is prevented from obtaining keys that would allow decryption of challenge ciphertexts. This approach provides a more practical security guarantee for schemes like AES-GCM that do not admit efficient programmability, at the cost of a weaker (but still standard) security model compared to full UC security.

5.5 Share Consistency Guarantee

Seal allows decryption of ciphertexts with varying sets of decryption keys. The TSSIBE-SHARECON property ensures that the decryption result is the same regardless of which set is used, including adversarial ones.

Definition 6 (TSSIBE-SHARECON)**.** *Adversary has negligible advantage in the following game with a challenger.*

1. *Receive* $pk_{[n]}$ *from the Adversary*
2. *Receive* $ID, c, \vec{sk}_{ID}, \vec{sk}'_{ID}$ *from Adversary*
3. *Adversary wins if*
   (a) $\vec{sk}_{ID} \neq \vec{sk}'_{ID}$ *and both have exactly* t *entries which are not* $\perp$*.*
   (b) $\texttt{admissible}(\vec{sk}_{ID}, \vec{pk}, ID, t) = \texttt{admissible}(\vec{sk}'_{ID}, \vec{pk}, ID, t) = 1$.
   (c) $\texttt{decrypt}(\vec{sk}_{ID}, c, aad, ID) \neq \texttt{decrypt}(\vec{sk}'_{ID}, c, aad, ID)$*, including the case that one of these is* $\perp$*.*

Theorem 4. *TSS-BF-KEM unconditionally satisfies TSSIBE-SHARECON.*

Proof. If both decryptions fail then the result holds. Now assume the decryption with $\vec{sk}_{ID}$ returns M. Roughly, the intuition is that the *nonce* unconditionally binds to r, and given that value, the encapsulated key is deterministically derived.

Formally, Let the ciphertext be

$$c = (\vec{pk}, t, nonce, c_r, c_{[n]}, sem),$$

Assume the decryption using vector $\vec{sk}_{ID}$, succeeds and outputs M. We show that decryption with any other admissible vector $\vec{sk}'_{ID}$, must also succeed and output the same M.

Since decryption with vector $\vec{sk}_{ID}$ succeeds, this implies:

1. Each share k_i, such that $\vec{sk}_{ID,i} \neq \perp$ is recovered as:
$$k_i = c_i \oplus H_2(i, (\vec{pk})_i, H_1(ID), nonce, e(sk_{ID,i}, nonce))$$
$$= c_i \oplus H_2(i, (\vec{pk})_i, H_1(ID), nonce, e(H_1(ID), \mathsf{pk}_i)^r).$$
2. The t shares k_i interpolate the unique degree-$(t-1)$ polynomial $p(x)$, yielding $k = p(0)$.
3. Let $(k_r, k_{sym}) \leftarrow H_3(k, \vec{pk}, t, c'_{[n]})$
4. Randomness r is recovered via:
$$r = c_r \oplus k_r.$$
5. The nonce check $g_2^r = nonce$ passes.

6. For all j, such that $\vec{sk}_{ID,j} = \perp$, we have:

$$k_j = c_j \oplus H_2(j, (\vec{pk})_j, H_1(ID), nonce, e(H_1(ID), pk_j^r))$$

$$= c_j \oplus H_2(j, (\vec{pk})_j, H_1(ID), nonce, e(sk_{ID,j}, nonce))$$

7. By the share consistency check, it is guaranteed that these k_j's are same as those uniquely determined by k_i's. Therefore they interpolate the same k.

8. The payload sem is successfully decrypted with k_{sym} to yield M.

Now we analyze decryption with the vector $\vec{sk}'_{ID}$:

1. For each j, such that $\vec{sk}'_{ID,j} \neq \perp$, compute:

$$k_j = c_j \oplus H_2(j, (\vec{pk})_j, H_1(ID), nonce, e(sk_{ID,j}, nonce))$$

$$= c_j \oplus H_2(j, (\vec{pk})_j, H_1(ID), nonce, e(H_1(ID), \mathsf{pk}_j^r)).$$

   which is same as those computed in Step 6 above.

2. The t shares k_j interpolate the same k as above, since share consistency passed for $\vec{sk}_{ID}$.

3. Randomness r is same as above, since k_r is same. So nonce check passes. In addition, share consistency also passes, since it passed for $\vec{sk}_{ID}$ with the same r.

4. The authenticated encryption is decrypted to recover the same message M, since k_{sym} is same.

Therefore, if decryption for the vector $\vec{sk}_{ID}$ passes and outputs M, so does for vector $\vec{sk}'_{ID}$. $\square$

5.6 Comparison to vetKeys

Our core cryptography is similar to vetKeys [CCN+23], but differs in specific details of IBE instantiation, threshold sharing, targeting multiple independent IBE keys, and has a different security analysis.

An important distinction is the way we handle threshold decryption. We allow the threshold servers to generate their IBE keys independently, rather than secret share those with the help of a setup DKG, which increases operational complexity.

- While vetKeys uses a setup DKG which results in a single encryption public key, we allow independent public keys for servers.

- We follow a KEM-DEM structure for the encryption, where a symmetric key is threshold shared and encrypted to the independent public keys. The message itself is symmetric encrypted using the key.

- We batch the independent encryptions to use common randomness and achieve $\approx 50\%$ reduction in KEM size. However, as opposed to constant size ciphertexts in vetKeys, our KEM size still scales linearly with the number of servers as a cost for key independence.

- An important security consideration in our model is **share consistency**: in on-chain scenarios public keys and secret keys can be adversarially generated, and the adversary can also dictate which subset of secret keys to use for decryption in a threshold setting. We ensure that the result of decryption is the same, including its failure, regardless of which subset is chosen. This property is crucial for maintaining deterministic behavior in adversarial environments.

References

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229, Santa Barbara, CA, USA, August 19–23, 2001. Springer Berlin Heidelberg, Germany.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer Berlin Heidelberg, Germany.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CCN$^+$23] Andrea Cerulli, Aisling Connolly, Gregory Neven, Franz-Stefan Preiss, and Victor Shoup. vetKeys: How a blockchain can keep many secrets. Cryptology ePrint Archive, Report 2023/616, 2023.

[LQ05] Benoît Libert and Jean-Jacques Quisquater. Identity based encryption without redundancy. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 2005: 3rd International Conference on Applied Cryptography and Network Security*, volume 3531 of *Lecture Notes in Computer Science*, pages 285–300, New York, NY, USA, June 7–10, 2005. Springer Berlin Heidelberg, Germany.