Seal Cryptography Specification and Implementation Audit

Dominik Apel[1] João Azevedo[1] Pyrros Chaidos[1,3] Bernardo David[1,4]
Jakov Mitrovski[1,2]

[1] Common Prefix
[2] Technical University of Munich
[3] University of Athens
[4] IT University of Copenhagen

June 12, 2025
Last update: August 12, 2025

1 Overview

1.1 Introduction

Mysten Labs commissioned Common Prefix to audit a part of the SEAL decentralized secrets management (DSM) service. SEAL enables users to encrypt messages while specifying a set of conditions for their decryption. When encrypting, the user selects a set of servers to entrust with decrypting the ciphertext and also chooses the number of servers that need to reply before decryption is possible (i.e. the threshold value). Encryption is performed with no involvement by the servers themselves, and produces a multi-part ciphertext. To decrypt, the user contacts the required number of servers to obtain decryption subkeys specialized to the decryption condition. Technically, each server holds a master key for an Identity Based Encryption (IBE) scheme similar to Boneh-Franklin [BF01]. A server will check that the condition is true before producing any subkeys. Collecting the mandated number of subkeys allows decryption to go through. Subkeys are structured to be checkable even in encrypted form, a valuable property if they are collected through intermediaries. That is, subkeys are BLS signatures on a message describing the decryption condition.

The primary objectives of the audit were to assess security, adherence to the provided specification, performance optimizations, and code quality. The audit focused on the specification and the TypeScript implementation of the main encryption and decryption functions of the Threshold Secret-shared IBE primitive. The encryption, decryption, and encrypted validation of subkeys for the Encrypted BLS Service were part of the specification but not present in the TypeScript codebase, as these operations are handled in a different part of the protocol implementation.

1.2 Audited Files

Audit start commit: [8f2ee1e]
Latest audited commit: [a5fe4ec]

1. Cryptography in Seal v2 - June 12, 2025 (shared privately)
   sha-256 744f8d15f6a338546b6c4b1962f6343673113e04a2286a4a86e6be56e48c96dd
2. bls12381.ts
3. decrypt.ts
4. kdf.ts
5. ibe.ts
6. encrypt.ts
7. dem.ts
8. elgamal.ts
9. shamir.ts

1.3 Disclaimer

This audit does not give any warranties on the bug-free status of the given code, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. We always recommend proceeding with several independent audits and a public bug bounty program to increase confidence in the security of the project.

The scope of this audit was strictly limited to the specification document and the TypeScript files listed above. Any other code, including files that import, interact with, or rely on the audited components, as well as external dependencies and third-party libraries, was considered out of scope of this audit. Importantly, the audit did not include the Encrypted BLS Service, as its implementation falls outside the audited TypeScript codebase.

1.4 Executive Summary

The audited specification document and implementation code were of high quality and free of any high-severity findings. The specification document included formal proofs for the claimed security properties and a pseudocode version of the SEAL primitive, providing a firm basis for the implementation audit. Several findings are related to the intended use of the primitive (e.g., not implementing the share consistency check present in the specification) and the choice of the security model (static corruption model). These deviations do not contradict the CCA security of the

IBE primitive but may become relevant depending on the application. Several other findings are related to the handling of zero polynomials. This has little relevance in the domain of Shamir Secret Sharing, but could surface in edge cases or when the code is reused in other domains with different assumptions. The remaining findings are informational and primarily relate to a non-critical specification mismatch, code clarity, and deviations from best practices.

1.5 Findings Severity Breakdown

Our findings are classified under the following severity categories, according to their impact and their likelihood of leading to an attack.

Level	**Description**
High	Logical errors or implementation bugs that are easily exploited. In the case of contracts, such issues can lead to any kind of loss of funds.
Medium	Issues that may break the intended logic, are deviations from the specification, or can lead to DoS attacks.
Low	Issues harder to exploit (exploitable with low probability), can lead to poor performance, clumsy logic, or seriously error-prone implementation.
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable, and easier to maintain.

2 Findings

2.1 High

None found

2.2 Medium

M01 Deviation from specification: Missing consistency check

Affected Code: decrypt.ts (line 91)

Summary: The implementation is missing the share consistency check used to achieve the share consistency property in the specification. However, the nonce check is present, preventing any tampering with the ciphertexts.

Suggestion: We suggest adding documentation that explains why the share consistency property is not required in the current codebase and notes the limitations that other implementations should be aware of when using this code as a reference. A note should be added to the IND-CCA proof, which states that the checks are not used and thus not necessary for achieving the property. Another option is to mandate a robust secret sharing scheme with a $t = \frac{n}{3} - 1$ threshold.

Status: Resolved [bdbb922]

2.3 Low

L01 Out-of-bounds memory access when multiplying polynomials

Affected Code: shamir.ts (line 163)

Summary: The `mul` method in the `Polynomial` class directly accesses entries in the `coefficients` array. This can cause out-of-bounds array access when one of the polynomials is the zero polynomial.

For example, when calling `Polynomial.zero().mul(somePolynomial)`, the code will try to access `this.coefficients[0]` where `this.coefficients` is an empty array.

Suggestion: Consider replacing the direct array access with the safe `getCoefficient` method as shown in the line below:

```
sum = sum.add(this.getCoefficient(j).mul(other.getCoefficient(i - j)));
```

Status: Resolved [ffc6e4b]

L02 Asymmetric behavior and out-of-bounds access when checking equality

Affected Code: shamir.ts (lines 249-254)

Summary: The `equals` method has unexpected behavior when comparing with the zero polynomial. For example, `Polynomial.zero().equals(Polynomial.one())` incorrectly returns **true**, while `Polynomial.one().equals(Polynomial.zero())` throws an error due to accessing `other.coefficients[0]` on an empty array. The root cause is that the method compares degrees first (both have degree 0), then tries to compare coefficients without using the safe `getCoefficient` method.

Suggestion: We suggest modifying the `equals` method similarly to the provided code snippet:

Code Listing 1.1: Equals method suggestion

```
1       equals(other: Polynomial): boolean {
2           if (this.coefficients.length !==
                other.coefficients.length) {
3               return false;
4           }
5           return this.coefficients.every((c, i) =>
                c.equals(other.getCoefficient(i)));
6       }
```

This compares the number of entries in the coefficients array (which distinguishes zero from non-zero polynomials) and uses the safe `getCoefficient` method to prevent out-of-bounds access.

Status: Resolved [ffc6e4b]

2.4 Informational

I01 Deviation from best practices

Affected Code: bls12381.ts (line 83)

Summary: The `hashToCurve` method in the `G2Element` class is currently an instance method, but it would be more appropriate as a static method. Since it generates a new `G2Element` from input data rather than operating on an existing instance.

Suggestion: We suggest making the `hashToCurve` method static to follow best practices.

Status: Resolved [d4d58b4]

I02 Non-critical deviation from specification: Require that exactly a threshold number of shares are provided

Affected Code: decrypt.ts (line 41)

Summary: The current implementation only checks that `inKeystore.length < encryptedObject.threshold` and throws an error if not enough shares are available. However, the specification requires an exact match between the number of shares provided and the threshold value, rather than just ensuring sufficient shares are available. More importantly, if more shares than the threshold are provided and some of them are incorrect, the Shamir secret sharing reconstruction will produce an incorrect result that may appear valid. For example, suppose the threshold is three and we provide four shares, three of which are correct and one of which is incorrect. In that case, the `combine` function will interpolate a wrong polynomial and return a wrong secret, leading to decryption failure.

Suggestion: We suggest changing the condition from `inKeystore.length < encryptedObject.threshold` to `inKeystore.length !== encryptedObject.threshold` to enforce exact threshold matching. This ensures specification compliance.

Status: Acknowledged

I03 Non-critical deviation from specification: Missing admissibility check

Affected Code: decrypt.ts (line 51)

Summary: The implementation is missing the secret key admissibility check used to validate the set of provided secret keys against their corresponding verification keys. This does not contradict any security properties, as their definitions assume the provided keys are correct. However, depending on the application (e.g. an on-chain decryption or decryption within a SNARK), the lack of such checks may enable the creation of ciphertexts that fail to decrypt using correct keys but are able to be "pretend decrypted" using specially crafted (but invalid) placeholders.

Suggestion: We suggest adding documentation that explains why the admissibility check is not required in the current codebase and notes the limitations that other implementations should be aware of when using this code as a reference.

Status: Resolved [a9c1964]

I04 "Plain" encryption strategy provides no encryption

Affected Code: dem.ts (line 98)

Summary: The "Plain" encryption strategy does not actually perform encryption - it just returns the key directly. This is confusing and potentially dangerous as it gives the false impression of encryption.
Suggestion: We suggest removing the "Plain" encryption strategy entirely as it may lead to confusion about whether data is encrypted.
Status: Resolved [f5954f3, a689625]

I05 Bypass checks with negative `threshold`, redundant check `keyServers.length > MAX_U8`

Affected Code: encrypt.ts (lines 51-54)
Summary: The check `keyServers.length > MAX_U8` is redundant because the combination of `keyServers.length < threshold` and `threshold > MAX_U8` already ensures that `keyServers.length` cannot exceed `MAX_U8`. Additionally, the check `threshold === 0` only prevents zero thresholds but allows negative threshold values, which are invalid for threshold-based secret sharing.
Suggestion: Consider removing the redundant check `keyServers.length > MAX_U8` as it is logically impossible to satisfy this condition when the other checks fail. We also suggest changing `threshold === 0` to `threshold <= 0` which will make `encrypt` to fail if the threshold is negative.
Status: Partially Resolved [f5f330a]

I06 Deviation from key management best practices

Affected Code: encrypt.ts (line 108)
Summary: The `encrypt` function returns the encryption key to the caller, which moves the responsibility of key disposal to the caller, which creates a security risk.
Suggestion: Remove the key from the return value and securely dispose of it within the function. If the key is needed for backup purposes, consider a separate secure backup mechanism instead of returning it directly.
Status: Acknowledged

I07 Potential hash collisions in exported key derivation functions

Affected Code: kdf.ts (lines 46, 85-87)

Summary: The `kdf` and `deriveKey` functions accept `index` and `threshold` parameters of type number but cast internally to `Uint8`. This could lead to collisions in the key derivation process if values that are not in `Uint8` range are passed. The same problem can arise when a float is passed as threshold or index to `kdf` and `deriveKey`, since casting the parameter to a `Uint8Array` will perform implicit rounding. Additionally, the `deriveKey` function joins the `keyServers` array and includes the result in the hash preimage. This allows for collisions if the attacker can craft malicious arrays that contain elements of unequal lengths.

Suggestion: We suggest adding range validation for the `threshold` and the `index` parameters in both functions to prevent potential collisions, as well as checking whether they are integers using `Number.isInteger()`. In addition, convert the elements of the `keyServers` array to a fixed length before joining them. This prevents the attacker from finding collisions even if the attacker is to craft the `keyServers` array freely.

Status: Resolved [a4e7009, a689625]

I08 Duplicate code

Affected Code: shamir.ts (lines 220-240)

Summary: The `combine` method implements polynomial interpolation and evaluation at zero. This duplicates logic that already exists in the `interpolate` and `evaluate` methods. The current implementation is more efficient, but it creates code duplication and reduces maintainability.

Suggestion: Consider using the existing methods for better maintainability (e.g., `Polynomial.interpolate(coordinates).evaluate(0)`). Alternatively, the current, more efficient implementation can be kept, but we suggest extracting the validation logic into a method to avoid duplicate code.

Status: Acknowledged

I09 Missing default cases in switch statements

Affected Code:

- encrypt.ts (line 129)
- kdf.ts (line 56)

Summary: Multiple switch statements lack default cases, preventing consistent error handling:

- `encrypt.ts`: Switch statement for `kemType` values lacks a default case.
- `kdf.ts`: Switch statement for `KeyPurpose` values lacks a default case, causing the function to return `undefined` for invalid values.

Suggestion: Add default cases to all switch statements to handle unexpected values consistently. For the `kemType` switch, throw an error for unsupported values. For the `KeyPurpose` switch, either throw an error for invalid values or return a safe default value.

Status: Resolved [a4e7009]

I10 JSDoc inconsistencies across exported functions and methods

Summary: Multiple exported functions and methods have inconsistent or missing JSDoc documentation:

- `ibe.ts`: `encapBatched` function JSDoc states "returns a list of keys, 32 bytes each" but actually returns GTElement[] (576 bytes each).
- `ibe.ts`: `decrypt` function JSDoc does not match the arguments passed to the function.
- `encrypt.ts`: `encrypt` function JSDoc documents encryptionInput as "AesGcmEncryptionInput or Plain" but omits Hmac256Ctr, which is also valid.
- `dem.ts`: `EncryptionInput` interface missing JSDoc documentation.
- `shamir.ts`: Consider adding documentation to the `Polynomial` class, especially to `div` which could be confused with division between polynomials.

Suggestion: We suggest standardizing JSDoc documentation across all exported functions and methods by:

- Correcting return type descriptions to match actual implementations.
- Adding missing JSDoc for undocumented functions/methods.
- Ensuring parameter documentation is complete and accurate.
- Validating that enum values match documented types.
- Adding input validation documentation where applicable.

Status: Resolved [bdbb922, fd17f17]

I11 Model Restriction: Adaptive corruptions

Summary: The security definitions for Encrypted BLS and TSS-BF-KEM-CCA are modeled against a static set of corrupted servers. A dynamic model may be more appropriate for real-world scenarios where there are adaptive corruptions. The schemes may even be adaptively secure under certain conditions, but this is not analyzed.

Suggestion: The adaptive setting is hard to provide a general solution for. It may be worth investigating a delayed corruption model featuring role transfers with erasures.

Status: Acknowledged

I12 Scope Restriction: Encrypted BLS service

Summary: The Encrypted BLS Service is featured in the specification but is not included in the audited codebase, as it is implemented on a different layer. The elgamal.ts file is not related to the Elgamal encryption of shares.

Suggestion: No action needed.

Status: Resolved

References

BF01. Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

About Common Prefix

Common Prefix is a blockchain research and development company. We believe blockchains will re-organize the world's economy to achieve planetary–scale efficient resource allocation and governance. We do the science and engineering to bring the blockchain field towards mainstream adoption and real-world impact, tackling foundational problems such as scalability, interoperability, and usability. We specialize in all aspects of blockchain science, from the low-level consensus of Layer-1s to the high-level tokenomics of DeFi applications. We have in-house scientist experts in game theory, incentives, auctions, cryptography, zero knowledge, multiparty computation, light clients, wallets, signature schemes, DeFi, and smart contracts from top universities worldwide. We are a team with multichain expertise, with engineers working in ecosystems ranging from Bitcoin and Ethereum, to Cosmos, Cardano, and Solana. We work with blockchain-first–only industry partners who push the boundaries of what is possible. We help them design, analyze, implement, deploy, and commercialize rigorous protocols from first principles, with provable security and pragmatism in mind.