FEVERTOKENS

Private Packages in the FeverTokens Package-Oriented Framework

November 2025

Abstract

This note introduces a standard approach for integrating Private Packages into the FeverTokens Package-Oriented Framework (POF). By extending the existing POF architecture, we introduce methodologies to design Private Packages that manage encrypted state, cryptographic commitments, or trusted off-chain execution. This approach preserves the framework's strict modularity while enabling a new class of hybrid applications where public transparency and institutional privacy coexist synchronously within the same smart contract address.

Three complementary privacy methodologies are presented: Fully Homomorphic Encryption (FHE) for on-chain encrypted computation, Zero-Knowledge Proofs (ZKP) for cryptographic state verification, and Trusted Execution Environments (TEE) for off-chain confidential processing with on-chain attestation.

Architectural Context & Adaptation

The FeverTokens framework relies on rigid separation of concerns to ensure upgradeability and auditability. It utilizes a five-file architecture: Internal Interface, External Interface, Storage, Internal Logic, and Package Facade.

In introducing privacy, we do not alter this foundation. Instead, we adapt the Storage and Logic components to handle non-public data types. Because the POF utilizes the Diamond Standard (EIP-2535), all packages (public or private) share the same execution context and storage space. This allows developers to build hybrid applications that leverage the trust of public transparent ledgers, the security of cryptographic privacy, and the flexibility of trusted execution, all within a unified and composable architecture.

A - Key Engineering Principles for Private Packages

Storage: Uses EIP-7201 namespaced storage slots to hold Encrypted Handles (FHE), State Roots (ZKP), or Attestation Records (TEE) rather than raw public values.

Logic: Operations are performed homomorphically on-chain (FHE), verified cryptographically via proofs (ZKP), or executed off-chain with attestation verification (TEE).

Composability: Private packages remain standard facets, allowing them to interoperate seamlessly with existing public packages.

B - Methodology I: Fully Homomorphic Encryption (FHE)

Concept

This approach utilizes FHE-enabled EVM environments. State variables are stored as encrypted types (e.g., euint32, ebool). Computation is performed on-chain over these ciphertexts, resulting in new ciphertexts without ever decrypting the data.

Important: FHE library naming differs between ecosystems. Fhenix uses the FHE namespace, while Zama/Inco networks use TFHE. The examples below use Fhenix syntax; adaptation for other networks requires updating the import path and library prefix.

Framework Implementation

A. Storage (PrivateFHEStorage.sol)

Storage slots hold encrypted handles. Note the EIP-7201 compliant namespace prefix.

Solidity

```
pragma solidity ^0.8.20;


// Fhenix import - use TFHE for Inco/Zama
import {FHE, euint32, ebool} from "@fhenixprotocol/contracts/FHE.sol";


library PrivateFHEStorage {
    struct Layout {
        euint32 privateCounter;
        mapping(address => euint32) encryptedBalances;
```

```
    }

    // EIP-7201 compliant storage slot with "eip7201:" prefix
    bytes32 constant STORAGE_SLOT = keccak256(
        abi.encode(uint256(keccak256("eip7201:company.storage.PrivateFHE")) - 1)
    ) & ~bytes32(uint256(0xff));

    function layout() internal pure returns (Layout storage l) {
        bytes32 slot = STORAGE_SLOT;
        assembly { l.slot := slot }
    }
}
```

B. Internal Logic (PrivateFHEInternal.sol)

Logic flow replaces standard branching (if/else) with oblivious multiplexing (select) to prevent access pattern leakage. ACL grants are mandatory for encrypted values to persist across transactions.

Solidity

```
pragma solidity ^0.8.20;

import {PrivateFHEStorage} from "./PrivateFHEStorage.sol";
import {FHE, euint32, ebool, inEuint32} from "@fhenixprotocol/contracts/FHE.sol";

abstract contract PrivateFHEInternal {
    using PrivateFHEStorage for PrivateFHEStorage.Layout;

    function _incrementPrivate(inEuint32 calldata encryptedAmount) internal {
        PrivateFHEStorage.Layout storage l = PrivateFHEStorage.layout();

        // Convert input ciphertext to FHE type
        euint32 amount = FHE.asEuint32(encryptedAmount);

        // Homomorphic addition: Encrypted + Encrypted = Encrypted
        l.privateCounter = FHE.add(l.privateCounter, amount);

        // CRITICAL: Grant contract access to persist the value
```

```
        FHE.allowThis(l.privateCounter);
    }

    function _conditionalLogic(euint32 threshold) internal {
        PrivateFHEStorage.Layout storage l = PrivateFHEStorage.layout();

        // Returns an encrypted boolean
        ebool condition = FHE.gt(l.privateCounter, threshold);

        // Use FHE.select for branching logic without revealing the branch
        l.privateCounter = FHE.select(
            condition,
            l.privateCounter,
            FHE.asEuint32(0)
        );

        // Grant access for persistence
        FHE.allowThis(l.privateCounter);
    }

    function _grantUserAccess(address user) internal {
        PrivateFHEStorage.Layout storage l = PrivateFHEStorage.layout();
        // Allow user to decrypt their own balance off-chain
        FHE.allow(l.encryptedBalances[user], user);
    }
}
```

C - Methodology II: Zero-Knowledge Proofs (ZKP)

Concept

In this model, the contract does not store the actual state (balances, votes). Instead, it stores a State Commitment (Merkle Root). The user computes transitions off-chain and submits a ZK Proof. The package acts strictly as a Verifier.

Important: Groth16 verifiers (generated by snarkjs/circom) use separate elliptic curve point arrays, not a single bytes blob. The interface below reflects the actual snarkjs output. PLONK verifiers use a different signature with `bytes memory proof`.

Framework Implementation

A. Storage (PrivateZKStorage.sol)

Manages the state root, nullifiers, and verifier address.

Solidity

```
pragma solidity ^0.8.20;

library PrivateZKStorage {
    struct Layout {
        bytes32 stateRoot;                    // Merkle root of private state
        mapping(bytes32 => bool) nullifiers;  // Prevent proof reuse
        address verifier;                     // Groth16 verifier contract
    }

    // EIP-7201 compliant storage slot
    bytes32 constant STORAGE_SLOT = keccak256(
        abi.encode(uint256(keccak256("eip7201:company.storage.PrivateZK")) - 1)
    ) & ~bytes32(uint256(0xff));

    function layout() internal pure returns (Layout storage l) {
        bytes32 slot = STORAGE_SLOT;
        assembly { l.slot := slot }
    }
}
```

B. Verifier Interface (IGroth16Verifier.sol)

The Groth16 verifier interface uses separate arrays for curve points A (G1), B (G2), and C (G1). The public signals array size is circuit-dependent.

Solidity

```
pragma solidity ^0.8.20;

/// @notice Groth16 verifier interface (snarkjs-generated)
/// @dev Public input array size [N] depends on circuit design
interface IGroth16Verifier {
    function verifyProof(
        uint256[2] calldata _pA,      // G1 point (x, y)
        uint256[2][2] calldata _pB,   // G2 point over extension field
        uint256[2] calldata _pC,      // G1 point (x, y)
        uint256[4] calldata _pubSignals  // [oldRoot, newRoot, nullifier, publicSignal]
    ) external view returns (bool);
}
```

C. Internal Logic (PrivateZKInternal.sol)

Integrates the Groth16 verifier with proper proof structure.

Solidity

```
pragma solidity ^0.8.20;

import {PrivateZKStorage} from "./PrivateZKStorage.sol";
import {IGroth16Verifier} from "./interfaces/IGroth16Verifier.sol";

abstract contract PrivateZKInternal {
    using PrivateZKStorage for PrivateZKStorage.Layout;

    event StateUpdated(bytes32 indexed newRoot, bytes32 indexed nullifier);

    function _updateState(
        uint256[2] calldata pA,
        uint256[2][2] calldata pB,
        uint256[2] calldata pC,
        bytes32 newRoot,
        bytes32 nullifier,
```

```
        bytes32 publicSignal
    ) internal {
        PrivateZKStorage.Layout storage l = PrivateZKStorage.layout();

        require(!l.nullifiers[nullifier], "Nullifier already used");

        // Construct public inputs array
        uint256[4] memory pubSignals = [
            uint256(l.stateRoot),
            uint256(newRoot),
            uint256(nullifier),
            uint256(publicSignal)
        ];

        // Verify Groth16 proof
        bool valid = IGroth16Verifier(l.verifier).verifyProof(
            pA, pB, pC, pubSignals
        );
        require(valid, "Invalid ZK proof");

        // Update state
        l.nullifiers[nullifier] = true;
        l.stateRoot = newRoot;

        emit StateUpdated(newRoot, nullifier);
    }
}
```

D - Methodology III: Trusted Execution Environments (TEE)

Concept

Trusted Execution Environments provide hardware-isolated computation where sensitive data can be processed without exposure to the host system. TEE-based privacy differs fundamentally from FHE and ZKP: computation occurs off-chain in a secure enclave, with the blockchain serving as the trust anchor for attestation verification and result commitment.

This approach is particularly suitable for computationally intensive operations that would be prohibitively expensive on-chain (FHE) or require complex circuit design (ZKP). The trade-off is trust delegation to hardware manufacturers (Intel for SGX, AWS for Nitro Enclaves) rather than pure cryptographic guarantees.

Supported TEE Platforms:

- Intel SGX: Creates memory-isolated enclaves with CPU-level encryption. Used by Secret Network, Oasis Network, and iExec.
- AWS Nitro Enclaves: Virtualization-based isolation on EC2 instances. No persistent storage or external networking; communicates only via secure VSOCK channel.
- AMD SEV: Memory encryption for virtual machines, relevant for cloud-based blockchain infrastructure.

Framework Implementation

A. Storage (PrivateTEEStorage.sol)

Stores attestation records, registered enclaves, and computation results. The contract acts as a registry and verification layer for TEE-executed computations.

Solidity

```
pragma solidity ^0.8.20;


library PrivateTEEStorage {
    struct EnclaveInfo {
        bytes32 mrEnclave;        // Measurement of enclave code (PCR0)
        bytes32 mrSigner;         // Measurement of enclave signer
        uint256 registeredAt;
        bool isActive;
    }


    struct ComputationResult {
        bytes32 resultHash;       // Hash of computation output
        bytes32 inputHash;        // Hash of inputs for verification
        address enclave;          // Enclave that produced the result
        uint256 timestamp;
        bool verified;
    }
```

```
    struct Layout {
        mapping(address => EnclaveInfo) registeredEnclaves;
        mapping(bytes32 => ComputationResult) results;
        mapping(bytes32 => bool) usedNonces;
        address attestationVerifier;  // On-chain attestation verifier
    }

    bytes32 constant STORAGE_SLOT = keccak256(
        abi.encode(uint256(keccak256("eip7201:company.storage.PrivateTEE")) - 1)
    ) & ~bytes32(uint256(0xff));

    function layout() internal pure returns (Layout storage l) {
        bytes32 slot = STORAGE_SLOT;
        assembly { l.slot := slot }
    }
}
```

B. Attestation Verifier Interface (IAttestationVerifier.sol)

The attestation verifier validates that computation occurred in a genuine TEE. For Intel SGX, this involves ECDSA verification; for AWS Nitro, it requires P-384/SHA-384 certificate chain validation.

Solidity

```
pragma solidity ^0.8.20;

/// @notice Interface for TEE attestation verification
/// @dev Implementation varies by TEE platform (SGX vs Nitro)
interface IAttestationVerifier {
    struct Attestation {
        bytes quote;              // Raw attestation quote/document
        bytes signature;          // Attestation signature
        bytes certificateChain;   // Certificate chain for verification
        bytes32 reportData;       // User-defined data in attestation
    }

    /// @notice Verify an attestation document
    /// @param attestation The attestation to verify
```

```
    /// @param expectedMrEnclave Expected enclave measurement
    /// @return valid True if attestation is valid and matches expected enclave
    function verifyAttestation(
        Attestation calldata attestation,
        bytes32 expectedMrEnclave
    ) external view returns (bool valid);

    /// @notice Extract report data from verified attestation
    /// @param attestation The attestation to extract from
    /// @return reportData The user-defined report data
    function extractReportData(
        Attestation calldata attestation
    ) external pure returns (bytes32 reportData);
}
```

C. Internal Logic (PrivateTEEInternal.sol)

The internal logic manages enclave registration and result submission with attestation verification.

Solidity

```
pragma solidity ^0.8.20;

import {PrivateTEEStorage} from "./PrivateTEEStorage.sol";
import {IAttestationVerifier} from "./interfaces/IAttestationVerifier.sol";

abstract contract PrivateTEEInternal {
    using PrivateTEEStorage for PrivateTEEStorage.Layout;

    event EnclaveRegistered(address indexed enclave, bytes32 mrEnclave);
    event ComputationSubmitted(bytes32 indexed taskId, bytes32 resultHash);

    error EnclaveNotRegistered();
    error AttestationFailed();
    error NonceAlreadyUsed();

    function _registerEnclave(
```

```
    address enclaveAddress,
    bytes32 mrEnclave,
    bytes32 mrSigner,
    IAttestationVerifier.Attestation calldata attestation
) internal {
    PrivateTEEStorage.Layout storage l = PrivateTEEStorage.layout();

    // Verify the attestation proves this is a genuine enclave
    bool valid = IAttestationVerifier(l.attestationVerifier)
        .verifyAttestation(attestation, mrEnclave);
    if (!valid) revert AttestationFailed();

    // Register the enclave
    l.registeredEnclaves[enclaveAddress] = PrivateTEEStorage.EnclaveInfo({
        mrEnclave: mrEnclave,
        mrSigner: mrSigner,
        registeredAt: block.timestamp,
        isActive: true
    });

    emit EnclaveRegistered(enclaveAddress, mrEnclave);
}

function _submitComputationResult(
    bytes32 taskId,
    bytes32 resultHash,
    bytes32 inputHash,
    bytes32 nonce,
    IAttestationVerifier.Attestation calldata attestation
) internal {
    PrivateTEEStorage.Layout storage l = PrivateTEEStorage.layout();

    // Prevent replay attacks
    if (l.usedNonces[nonce]) revert NonceAlreadyUsed();
    l.usedNonces[nonce] = true;

    // Verify caller is a registered enclave
    PrivateTEEStorage.EnclaveInfo storage enclave =
```

```
            l.registeredEnclaves[msg.sender];
        if (!enclave.isActive) revert EnclaveNotRegistered();

        // Verify fresh attestation with result commitment
        bytes32 expectedReportData = keccak256(
            abi.encodePacked(taskId, resultHash, inputHash, nonce)
        );
        bytes32 actualReportData = IAttestationVerifier(l.attestationVerifier)
            .extractReportData(attestation);
        require(expectedReportData == actualReportData, "Report data mismatch");

        // Store verified result
        l.results[taskId] = PrivateTEEStorage.ComputationResult({
            resultHash: resultHash,
            inputHash: inputHash,
            enclave: msg.sender,
            timestamp: block.timestamp,
            verified: true
        });

        emit ComputationSubmitted(taskId, resultHash);
    }
}
```

Security Considerations

Hardware Trust: TEE security relies on hardware manufacturer integrity. While unlikely, vulnerabilities like Plundervolt have demonstrated that SGX can be compromised. Distributed key management across multiple TEE nodes mitigates single-point-of-failure risks.

Attestation Freshness: Attestations should include nonces or timestamps to prevent replay attacks. The contract must verify that attestation report data matches the submitted computation parameters.

Side-Channel Attacks: Code running in TEEs should be designed to minimize timing variations and memory access patterns that could leak information.

Unified Composability: The Hybrid Workflow

The strongest argument for this integration is Synchronous Composability. Because all Facets share the `address(this)` context, any Private Package, whether FHE, ZKP, or TEE-based, can read the storage layout of a Public Package directly in a single atomic transaction.

This eliminates the need for asynchronous cross-chain messaging or complex callback architectures usually required for privacy solutions.

Methodology Comparison

Aspect	**FHE**	**ZKP**	**TEE**
Trust Model	Cryptographic	Cryptographic	Hardware
Computation	On-chain (encrypted)	Off-chain (prover)	Off-chain (enclave)
Gas Cost	Very High	Medium (verification)	Low-Medium
Latency	Block time	Proving time (10-60s)	Near real-time
Best For	Simple operations on encrypted data	Proving membership, eligibility	Complex computation, ML inference

Example Scenario: Hybrid Multi-Method Workflow

Consider a confidential credit scoring system:

1. Public Phase: User submits a loan application (public record).
2. TEE Phase: Credit scoring algorithm executes in enclave using off-chain data.
3. ZKP Phase: User proves score exceeds threshold without revealing exact score.
4. FHE Phase: Loan terms are computed homomorphically based on encrypted score range.

Solidity

```
pragma solidity ^0.8.20;


import {GovernanceStorage} from "../governance/GovernanceStorage.sol";
import {PrivateZKInternal} from "./PrivateZKInternal.sol";
```

```
import {PrivateTEEStorage} from "./PrivateTEEStorage.sol";

abstract contract HybridCreditInternal is PrivateZKInternal {

    function _processLoanApplication(
        bytes32 applicationId,
        uint256[2] calldata pA,
        uint256[2][2] calldata pB,
        uint256[2] calldata pC,
        bytes32 newRoot,
        bytes32 nullifier
    ) internal {
        // 1. READ PUBLIC STATE (Direct Access)
        GovernanceStorage.Layout storage gov = GovernanceStorage.layout();
        require(gov.lendingEnabled, "Lending currently disabled");

        // 2. VERIFY TEE COMPUTATION (Check attestation result exists)
        PrivateTEEStorage.Layout storage tee = PrivateTEEStorage.layout();
        require(
            tee.results[applicationId].verified,
            "TEE credit score not computed"
        );

        // 3. EXECUTE ZK VERIFICATION (Prove eligibility)
        _updateState(pA, pB, pC, newRoot, nullifier, applicationId);

        // 4. PUBLIC ACTION (Record approval)
        emit LoanApproved(applicationId, msg.sender);
    }
}
```

Conclusion

This proposal demonstrates that the FeverTokens framework requires no structural changes to support advanced privacy. By treating privacy as a "Storage and Logic" attribute rather than an infrastructure change, the POF can accommodate three complementary privacy methodologies:

- FHE for on-chain encrypted computation with cryptographic guarantees
- ZKP for efficient state verification without revealing underlying data
- TEE for complex off-chain computation with hardware-attested integrity

Extending the POF to support Private Packages will allow developers to build hybrid applications that leverage the trust of public transparent ledgers, the security of cryptographic privacy, and the flexibility of trusted execution, all within a unified and composable architecture.