Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Auditor Scan

Scan ID	36
Date	February 18, 2026
Organization	igor53627
Repository	evm-lwe-math
Branch	main
Commit Hash	5849af3c...0b9b2e80

Contracts in scope

src/LWEPacking.sol

src/LibLWE.sol

Code Statistics

Findings	Contracts Scanned	Lines of Code
5	2	445

Findings Summary

Total Findings

🟥 High Risk (0)	🟦 Info (0)
🟧 Medium Risk (0)	🟩 Best Practices (0)
🟨 Low Risk (5)	

Code Summary

This protocol provides a suite of gas-optimized Solidity libraries for performing on-chain cryptographic operations based on the Learning With Errors (LWE) lattice-based scheme. It is designed as a foundational utility for other smart contracts that require advanced cryptographic primitives, such as those used in Fully Homomorphic Encryption (FHE) or certain Zero-Knowledge Proof systems. The protocol is not a standalone application but a set of reusable building blocks.

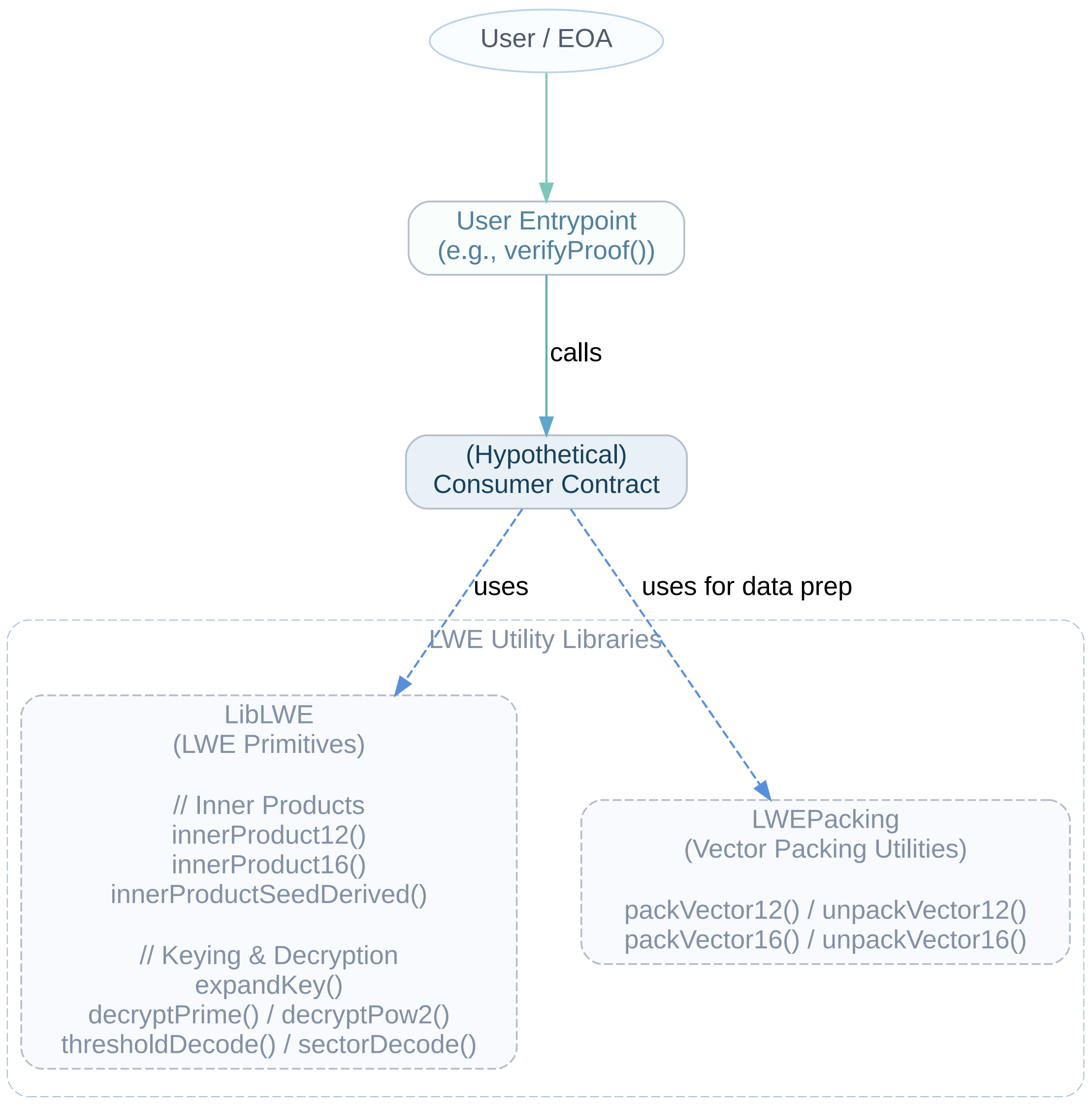
The core components are:
- `LWEPacking` **Library:** This library offers utilities for data compression. It contains functions to pack large vectors of small integers (12-bit or 16-bit) into `uint256` words and unpack them back. This is essential for reducing the gas costs associated with storing and transmitting cryptographic data on the blockchain.
- `LibLWE` **Library:** This library implements the fundamental LWE cryptographic operations. It leverages the packed data formats for efficiency and uses inline assembly for maximum gas optimization. Its key features include calculating inner products of vectors, deriving vectors on-the-fly from a compact seed to save calldata, expanding keys from seeds, and performing decryption and decoding steps to recover plaintext data from LWE ciphertexts.

The architecture is entirely library-based, with all functions marked as `internal`. This ensures that the cryptographic logic can be securely and efficiently integrated into other contracts without being exposed as a standalone, stateful entity.

Entry Points and Actors

This protocol is composed entirely of `internal` library functions. As such, it does not expose any external entry points that can be called by actors to modify state. It is intended to be used as a dependency by other contracts.

Code Diagram



1 of 5 Findings src/LibLWE.sol

`thresholdDecode(diff,threshold)` can exclude an integer that lies inside the documented `(q/4, 3q/4)` band when `threshold` is passed as `floor(q/4)` for `q % 4 != 0` (e.g., `q=65521`)

● Low Risk

`thresholdDecode` implements the “true band” check as strict inequalities against `threshold` and `3*threshold`:

```
function thresholdDecode(uint256 diff, uint256 threshold)
    internal
    pure
    returns (uint256 bit)
{
    assembly {
        bit := and(gt(diff, threshold), lt(diff, mul(3, threshold)))
    }
}
```

The docstring states: "For `q/4` threshold: true band is `(q/4, 3q/4)`".
For moduli where `q % 4 != 0` (notably the documented prime modulus `q=65521`), callers commonly compute `threshold` as `q/4` using integer division:
- `q = 65521 = 4*16380 + 1`
- `threshold = q/4 = 16380` (floor)
- `3*threshold = 49140`

The implemented check becomes `diff > 16380 && diff < 49140`, which accepts integer `diff` values `16381..49139`.
But the documented mathematical band `(q/4, 3q/4)` corresponds to `(16380.25, 49140.75)`, which includes integer values `16381..49140`.

As a result, `diff = 49140` lies inside the documented band (since `49140 < 49140.75`) but is excluded by the implementation when `threshold` is computed as `floor(q/4)`. This is a truncation/precision edge case that can flip the decoded bit exactly at that boundary value under the documented interpretation.

Severity Note:
- Callers compute threshold as floor(q/4) for q % 4 != 0, as suggested by the docstring shorthand.
- The bit produced by thresholdDecode is used in a way where a single false negative can cause a user-visible rejection, but not fund movement.

2 of 5 Findings

src/LibLWE.sol

expandKey reverts for invalid q even when numWords=0, contradicting expected empty result

● Low Risk

The black-box expectation is that calling expandKey with numWords=0 returns an empty array. However, the implementation validates q before allocating the output and will revert if q == 0 or q > 65536, even though no elements are produced when numWords==0. This contradicts the black-box invariant and can cause unexpected reverts in callers that rely on expandKey being a no-op for zero-length output.

```
function expandKey(bytes32 keySeed, uint256 numWords, uint256 q)
    internal
    pure
    returns (uint256[] memory s)
{
    require(q > 0 && q <= 65536, "q must fit in 16-bit lanes"); // Reverts even when
numWords == 0
    s = new uint256[](numWords);
    assembly { /* loop skipped when numWords==0 */ }
}
```

Because the revert is triggered prior to observing that numWords==0, any configuration or upstream misuse that provides an out-of-range q will fail hard instead of yielding the expected empty array, violating the stated invariant and potentially inducing denial-of-service behavior in dependent flows that conditionally use zero-length expansion.

Severity Note:
- There exist call sites that treat expandKey(numWords=0) as a valid no-op and do not pre-validate q.
- numWords can be zero due to configuration or user input in some flows.
- A revert at this point aborts the broader transaction rather than being isolated.

3 of 5 Findings

src/LWEPacking.sol

Loop contains `require` / `revert` statements

● Low Risk

The contract contains loops (specifically `for (uint256 k = 0; k < 16; k++)`) that include `require` or `revert` statements. This pattern is problematic because a single invalid item within the loop can cause the entire transaction to fail, preventing the processing of valid items. This occurs in 3 separate loop instances within the contract. A better approach would be to allow the loop to continue processing all items and collect failed items for post-processing, enabling partial success and better error handling.

4 of 5 Findings   src/LWEPacking.sol   src/LibLWE.sol

Magic Numbers Instead Of Constants

● Low Risk

Multiple magic number literals are used throughout the codebase instead of being defined as named constants. This reduces code readability and maintainability.

In src/LWEPacking.sol: The magic number `16` is used repeatedly in shift calculations (e.g., `uint256 shift = (15 - posInWord) * 16;`). This literal appears 13 times across various functions and should be extracted into a named constant for clarity and easier maintenance.

In src/LibLWE.sol: The magic number `3` is returned directly in multiple locations instead of being defined as a named constant. This value appears 2 times and should be extracted into a constant to improve code clarity.

5 of 5 Findings   src/LWEPacking.sol   src/LibLWE.sol

PUSH0 Opcode Compatibility Issue

● Low Risk

The contracts use Solidity compiler version `^0.8.20`, which defaults to the Shanghai EVM version. This version includes the PUSH0 opcode in the generated bytecode. However, not all EVM-compatible chains support the PUSH0 opcode, particularly Layer 2 solutions and other non-mainnet chains. If deployment is intended on chains that do not support PUSH0, the bytecode generation will fail. Ensure the appropriate EVM target version is explicitly specified in the compiler configuration to match the deployment target chain.