PROTOFIRE

February 2026

Smart contract audit
rain.math.float

Content

Project description

This repository contains a Solidity and Yul library that implements decimal floating-point mathematics specifically tailored for deterministic financial applications on the EVM. It utilizes a custom data structure with a 224-bit signed coefficient and a 32-bit signed exponent to ensure that human-readable decimal numbers have exact on-chain representations without the binary precision errors found in standard IEEE 754 implementations. The library deliberately rejects special values like NaN or Infinity in favor of strict errors, ensuring that operations like division by zero or overflows stop execution rather than propagating invalid states. Furthermore, it provides deterministic parsing and formatting guarantees along with support for advanced operations like logarithms and powers via on-chain lookup tables.

Executive summary

Type	Library
Languages	Solidity
Methods	Architecture Review, Manual Review, Unit Testing, Functional Testing, Automated Review
Documentation	README.md
Repositories	https://github.com/rainlanguage/rain.math.float

Reviews

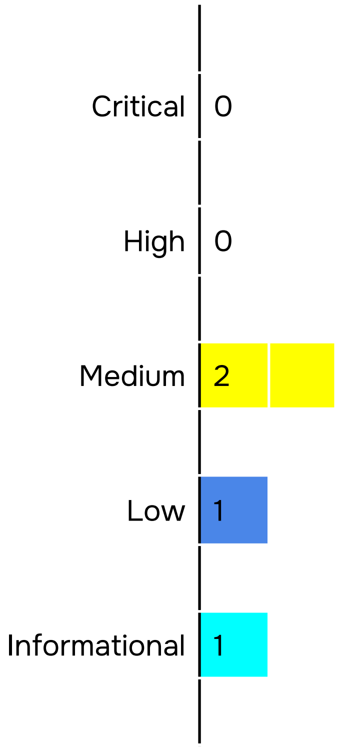
Date	**Repository**	**Commit**
20/01/26	rain.math.float	1a3a88ad580be0d8ddd50e6084c18b612f56107f
05/02/26	rain.math.float	fd7640f6bb06ebf33fd45f5c02f0afc06dcea5d7
10/02/26	rain.math.float	30586873e598cf031f1238e230a5bfe4bd0cfa28

Scope

Contracts
src/generated/LogTables.pointers.sol

src/concrete/DecimalFloat.sol
src/lib/deploy/LibDecimalFloatDeploy.sol
src/lib/parse/LibParseDecimalFloat.sol
src/lib/table/LibLogTable.sol
src/lib/implementation/LibDecimalFloatImplementation.sol
src/lib/format/LibFormatDecimalFloat.sol
src/lib/LibDecimalFloat.sol
src/error/ErrParse.sol
src/error/ErrFormat.sol
src/error/ErrDecimalFloat.sol

Technical analysis and findings



Security findings

**** Critical

No critical severity issue found.

*** High

No high severity issue found.

** Medium

M01 - Incorrect Mathematical Floor Implementation for Negative Numbers

The *floor()* function in LibDecimalFloat.sol incorrectly implements "truncation towards zero" instead of the mathematical definition of "rounding towards negative infinity."

In Solidity, integer division and modulo operations truncate towards zero. The *floor()* function relies on characteristicMantissa, which separates the integer part (characteristic) from the fractional part (mantissa) using these standard operations. Consequently:

1. For positive numbers, it behaves correctly: *floor(1.5) -> 1.0*.
2. For negative numbers, it behaves incorrectly: *floor(-1.5) -> -1.0*.

Mathematically, *floor(x)* is defined as the largest integer less than or equal to *x*. Therefore, *floor(-1.5)* must be *-2.0*, as *-1.0* is greater than *-1.5*. The current implementation violates the invariant *floor(x) <= x* for all negative non-integer values.

Path: src/lib/LibDecimalFloat.sol

Recommendation: Modify the *floor()* function to detect when a negative number has a non-zero fractional part (mantissa) and subtract 1 from the result.

Found in: 1a3a88ad

Status: Fixed at fd7640f6

M02 - Silent Zero Result when Dividing by Non-Maximizable One

In *LibDecimalFloatImplementation.div()*, if the denominator's coefficient is 1 and cannot be maximized (because its exponent is at *type(int256).min*), the scaling logic incorrectly reduces the scalar to 0.

The while (*signedCoefficientBAbs <= scale*) loop allows scale to reduce to zero if *signedCoefficientBAbs* is 1. Subsequently, *mulDiv()* uses this zero scalar, causing the entire division operation to return 0.

This creates a silent failure where dividing by an extremely small number (which should result in a large value or overflow) incorrectly returns zero.

Path: src/lib/implementation/LibDecimalFloatImplementation.sol

Recommendation: Modify the loop condition to strict inequality *signedCoefficientBAbs < scale* or add a guard to ensure scale never drops below 1. Given the logic implies finding a scale smaller than the denominator to prevent overflow, a specific check for *scale == 0* or *scale == 1* handling is needed, or simply preventing scale from becoming zero.

Found in: 1a3a88ad

Status: Fixed at fd7640f6

* Low

L01 - Incorrect Scientific Notation Formatting for Negative Numbers

The format function in DecimalFloat.sol incorrectly forces scientific notation for negative numbers that should be displayed in standard decimal format.

The current implementation determines whether to use scientific notation with the following check:
a.lt(scientificMin) || a.gt(scientificMax)

For a standard negative integer like -500:

1. The comparison -500 < 1e-4 evaluates to true.
2. This triggers the scientific formatting path.
3. The output becomes "-5e2" instead of the expected "-500".

The check fails to account for the fact that large negative numbers (in magnitude) are numerically "smaller" than the positive scientificMin threshold.

Path: src/concrete/DecimalFloat.sol

Recommendation: Update the condition to evaluate the absolute value of the number against the formatting thresholds. This ensures that scientific notation is only triggered when the number's magnitude is too small or too large for standard display.

Found in: 1a3a88ad

Status: Fixed at fd7640f6

Informational

I01 - Hardcoded Deterministic Deployment Proxy Address

The *decimalFloatZoltu()* function performs a low-level call to a hardcoded deterministic deployment proxy address: 0x7A0D94F55792C434d74a40883C6ed8545E406D12.
This address is embedded directly in the function body and cannot be modified or configured.
Hardcoding an external dependency inline inside function logic introduces some maintainability and integration risks:

- Critical configuration is hidden inside implementation details,
- The dependency may be easy to overlook during reviews and integrations.

Path: src/lib/deploy/LibDecimalFloatDeploy.sol

Recommendation: Move the proxy address into a named constant variable with clear documentation.

Found in: 1a3a88ad

Status: Fixed at fd7640f6

General Risks

- Approximation Variance: Operations involving logarithms and exponentiation (log10, pow, sqrt) rely on on-chain lookup tables and linear interpolation. These are approximations rather than exact analytical solutions, meaning distinct round-trip calculations may exhibit small deviations.

- Precision Loss and Rounding: While the library avoids binary floating-point errors, it still operates with finite precision. Operations that produce repeating decimals (like $1/3$) or involve packing values into the 32-byte Float format will undergo rounding (towards zero). This can lead to minor precision loss that may compound over long sequences of operations.

- Gas Consumption: The library implements complex mathematical functions in Solidity and Yul. Operations - especially those involving iterative algorithms or table lookups - are significantly more gas-intensive than standard integer arithmetic. Heavy usage within loops or high-frequency transactions could lead to substantial gas costs.

- Strict Revert Behavior: The library deliberately diverges from standard floating-point behavior by reverting on "nonsense" operations (e.g., division by zero, exponent overflow, logs of negative numbers) instead of returning NaN or Infinity. Systems interacting with this library must rigidly validate inputs, as unhandled invalid values will cause the entire transaction to fail.

Approach and methodology

To establish a uniform evaluation, we define the following terminology in accordance with the OWASP Risk Rating
Methodology:

	Likelihood Indicates the probability of a specific vulnerability being discovered and exploited in real-world scenarios
	Impact Measures the technical loss and business repercussions resulting from a successful attack
	Severity Reflects the comprehensive magnitude of the risk, combining both the probability of occurrence (likelihood) and the extent of potential consequences (impact)

Likelihood and impact are divided into three levels: High H, Medium M, and Low L. The severity of a risk is a blend of these two factors, leading to its classification into one of four tiers: Critical, High, Medium, or Low.

When we identify an issue, our approach may include deploying contracts on our private testnet for validation through testing. Where necessary, we might also create a Proof of Concept PoC to demonstrate potential exploitability. In particular, we perform the audit according to the following procedure:

	Advanced DeFi Scrutiny We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs
	Semantic Consistency Checks We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
	Security Analysis The process begins with a comprehensive examination of the system to gain a deep understanding of its internal mechanisms, identifying any irregularities and potential weak spots.