Introduction

This post presents a novel zero-knowledge proof scheme called the ZK Median, which is designed to improve the security of TWAP (Time-Weighted Average Price) oracles in DeFi (Decentralized Finance) protocols. We propose a proof-of-concept implementation of the ZK Median that uses ZK circuits to provide a tamper-resistant median value that can be used as a fail-safe for DeFi protocols that want to onboard emerging tokens with low liquidity or for DeFi protocols that have a need for cryptographic security inherited from the properties of a SNARK protocol.

It's important to note that the Solidity implementation of the scheme is not yet started, and the proof-of-concept implementation is still in the experimental stage. As the author acknowledges, there is a nontrivial chance that the overall scheme may not be robust, and there is at least one critical security flaw presently that requires a solution. Nonetheless, the ZK Median presents a promising approach to enhancing the security of TWAP oracles, and it may inspire other researchers to develop similar solutions to the problem of oracle manipulation in DeFi protocols.

Problem

TWAP (Time-Weighted Average Price) oracles are widely used in DeFi (Decentralized Finance) protocols to determine the price of assets. However, these oracles are vulnerable to various forms of manipulation, such as liquidity attacks, which can artificially inflate or deflate the price of an asset, leading to inaccurate or unreliable price information. This, in turn, can force DeFi protocols to make incorrect financial decisions, resulting in false liquidations, losses of funds, or other negative consequences for users. Therefore, there is a pressing need to develop more secure and tamper-resistant oracles that can provide accurate and reliable price information for DeFi protocols.

ZK Median

The ZK Median is a zero-knowledge proof scheme that can be used to enhance the security of TWAP (Time-Weighted Average Price) oracles in DeFi (Decentralized Finance) protocols. The scheme involves adding a wrapper contract to the TWAP oracle that either accesses N prices reports directly or records N checkpoint values of the prices at an arbitrary interval. Once N data points are available for a given interval, a zero-knowledge proof can be constructed to prove the median of the unsorted array of prices:

1. There is an unsorted array \mathbf{x}

of length N, which was constructed faithfully according to the logic of the checkpoint smart contract. The proof can be verified in one of two modes: (i) the array values are retrieved from contract storage and used as public inputs to an on-chain verifier; or (ii) the array is represented as a single hash value by incrementally forming a hash chain using an EVM implementation of the Poseidon hash function, and that value is used in the on-chain verifier. In either case, the prover cannot arbitrarily choose a price array as input.

1. There is an N x N square matrix \mathbf{A}

, which, when multiplied by column vector \mathbf{x}

, produces column vector \mathbf{y}

, such that \mathbf{y} = \mathbf{Ax}

. \mathbf{A}

is an invertible permutation matrix, but not necessarily unique because of the possibility of repeat price values. \mathbf{A}

contains only binary values.

1. The values in \mathbf{y}

are sorted, that is, {y_i <= y_{i+1} \ \forall \ i \in 0..\mathbf{N-1}}

. Again, we avoid strictly using <

because of the possibility of repeat price values.

1. The public output \mathbf{m}

of the circuit is the middle value of \mathbf{y}

. The proof shows that \mathbf{y}_{\lfloor N/2 \rfloor} = \mathbf{m}

, where N is a circuit-compile-time static value that must be odd.

The ZK Median provides a tamper-resistant median value that can be used by DeFi protocols to ascertain strictly on-chain asset prices. The median value \mathbf{m}

is resistant to manipulation attacks, so long as the values in \mathbf{y}

are constrained to only be inserted once-per-block or some other acceptably low amount-per-block. The ZK Median can operate in an as-needed fail-safe mode, where if an untrusted liquidity pool reports prices outside of an acceptable range of tolerance, then a ZK Median response is scheduled instead. Operations within the contract relying on that price oracle may be temporarily paused for a brief period until the ZK Median proof is calculated and delivered on-chain.

Proof of Concept

A proof-of-concept implementation of the ZK Median scheme can be found in the GitHub repository[zk-medianizer](). The repository includes the circuits used in the implementation, which can be found in the circuits

directory, as well as the unit tests, which can be found in the test

directory.

The proof-of-concept implementation relies on the [circomlib-matrix]() library, which is used to perform matrix multiplication and sort the resulting vector. The implementation demonstrates the feasibility of the ZK Median scheme and provides a starting point for future research and development. However, as the author notes, the implementation is still in the experimental stage and requires further refinement to ensure robustness and efficiency.

Conclusion

In conclusion, the ZK Median scheme is a promising approach to enhancing the security and resilience of TWAP oracles. Although the use of zero-knowledge proofs offers potential cost savings, the gas costs of the scheme may still be prohibitive, and further research and development are required to optimize the implementation. In addition, the static array length, as well as the fact that the number of constraints associated with the permutation matrix grow proportional to the square of N, are potential drawbacks that need to be better understood.

However, the ZK Median scheme has the potential to prevent same-block oracle manipulations entirely, which can increase the security of DeFi protocols. The scheme is designed to be built on top of an AMM TWAP oracle and offers increased resilience to oracle manipulation, making it a useful tool for protocols that want to build financial products with the price data of emerging tokens with low liquidity.

The ZK Median scheme is a high-security construction that comes at a likely high cost. As such, we do not recommend it for pools that have enough liquidity to be resilient to manipulation attacks. Nevertheless, for the gas cost of running this scheme, secure price oracles may be constructed even on low liquidity pools while still enjoying the other properties of the TWAP algorithm, but over an interval of blocks, and with increased resilience to manipulation.

Overall, we hope that this attempt at a ZK Median Oracle circuit will inspire further research, discussion, and development in the field of secure and resilient oracles.

References

This entire experiment was inspired by a tweet: “terribly bad idea: instead of computing median, send a computational* integrity proof that you computed median and this is the output” - twitter(.)com/martriay

I used ChatGPT to improve the quality of this document.

Author’s Note

We acknowledge that the constraints on the square matrix in this initial implementation are insufficient to prove that \mathbf{A}

is a permutation matrix. The circuit constrains each vector to be a unit vector, but that does not prove the invertibility of the matrix. This is a critical security flaw in the presented implementation, as it allows for the possibility of arbitrary price selection.

To address this issue, a better way to secure the scheme would be to prove that the determinant of \mathbf{A}

is either -1 or 1, which would allow for the removal of the unit vector checks. This method would provide more assurance that the matrix \mathbf{A}

is a permutation matrix, and that the median \mathbf{m}

is correctly computed. If anyone were to continue work on this implementation, implementing the matDeterminant component would be a great place to start.

pragma circom 2.0.3;

include "../node_modules/circomlib/circuits/comparators.circom"; include "../node_modules/circomlib-matrix/circuits/matMul.circom"; include "./matDeterminant.circom";

template SquareSortV2(n) { / *inputs* / // raw values to be sorted signal input unsortedValues[n];

```
// determinant of permutation matrix
signal input determinantOfA;

// permutation matrix
signal input permutationMatrixA[n][n];

/*
    outputs
*/
// sorted values will be assigned from the result of the matrix multiplication
signal output sortedValues[n];

/*
    constraints generation
*/
// instantiate matDeterminant component
component determinant = matDeterminant(n);
for (var i = 0; i < n; i++) {
    for (var j = 0; j < n; j++) {
        // constraint: elements in permutation matrix are binary
        permutationMatrixA[i][j] * (1 - permutationMatrixA[i][j]) === 0;

        // feed matDeterminant component inputs
        determinant.in[i][j] <== permutationMatrixA[i][j];
    }
}

// constraint: the provided value for determinantOfA is the determinant of permutationMatrixA
determinant.out === determinantOfA;

// constraint: the value of the determinant is either 1 or -1
(determinantOfA + 1) * (determinantOfA - 1) === 0;

// instantiate matMul component
component m = matMul(n, n, 1);
for (var i = 0; i < n; i++) {
    // feed matMul vector inputs
    m.b[i][0] <== unsortedValues[i];
    for (var j = 0; j < n; j++) {
        // feed matMul matrix inputs
        m.a[i][j] <== permutationMatrixA[i][j];
    }
}

// verify permutated array is sorted
component isSorted[n - 1];
for (var i = 0; i < n - 1; i++) {
    isSorted[i] = LessEqThan(252);
    isSorted[i].in[0] <== m.out[i][0];
    isSorted[i].in[1] <== m.out[i+1][0];
    isSorted[i].out === 1;
}

// assign the sorted values to the template's output
for (var i = 0; i < n; i++) {
    sortedValues[i] <== m.out[i][0];
}

}
```

Preliminary analysis of the critically flawed implementation revealed that using the Solidity-verifier-optimal hash method of verifying input data for an unsorted array of 69 values, the circuit used 43,215 constraints. The non-Solidity-verifier-optimal circuit with direct inputs used 26,658 constraints for the same size array. It is likely that the determinant method of proving \mathbf{A}

is a permutation matrix will more than double the number of constraints, which still falls within a reasonable range.

$ circom --r1cs test/circuits/median_optimal_test.circom // Output: template instances: 79 non-linear constraints: 43215 linear constraints: 0 public inputs: 0 public outputs: 1 private inputs: 4831 private outputs: 0 wires: 43148 labels: 104279 Written successfully: ./median_optimal_test.r1cs Everything went okay, circom safe

$ circom --r1cs test/circuits/median_test.circom // Output: template instances: 7 non-linear constraints: 26658 linear constraints: 0 public inputs: 0 public outputs: 1 private inputs: 4830 private outputs: 0 wires: 26591 labels: 51285 Written successfully: ./median_test.r1cs

It’s worth noting again that this is an experimental implementation, and there is a nontrivial chance that the overall scheme may not be robust. Further research and development are necessary to refine and optimize the implementation, and to ensure its efficiency, scalability, and security.