Evaluation of the DAO Protocol with BLS Aggregate Signature Verification

Jean Gal

Tuesday 14th of May

1 Introduction

The purpose of this report is to present benchmarks for a DAO protocol that incorporates on-chain BLS aggregate signature verification for voting purposes. This protocol is designed to improve the efficiency of voting processes within decentralized autonomous organizations (DAOs). The implementation details and source code are available in the repository at: https://github.com/jjeangal/bls-agg-dao.

2 Benchmark Data

The following table, table 1, presents the gas usage benchmarks for calling the `castVoteWithReasonAndBlsSig` function in the DAO protocol. Each entry represents a different number of aggregated signatures (n).

Number of Signatures (n)	Gas Used
1	206,632
2	233,956
3	261,231
4	288,579
5	315,902
6	343,214
7	370,541
8	397,892
9	425,207
10	452,546
11	479,849
12	507,189

Table 1: Gas usage benchmarks for `castVoteWithReasonAndBlsSig` function.

3 Gas Cost Analysis

The gas cost for validating a BLS signature aggregation can be expressed by the following formula, where k is the number of pairings executed:

$$\text{Pairing Costs} = 34,000 \times k + 45,000 \text{ (0x08 ecPairing)}$$

For two pairings in our `verifySingle` signature of the aggregated keys and signatures for one message, the cost is:

$$34,000 \times 2 + 45,000 = 113,000$$

The gas costs for invoking the function to cast a vote, excluding the signature verification operation, are calculated as follows:

$$\text{Total Gas (1 sig)} - \text{Pairing Cost} = 207,000 - 113,000 = 94,000$$

The additional gas cost for each of the n signatures, which mainly involves looping through the accounts and calculating the vote weights, is approximately 27,330.

4 Comparison with `ecRecover`

To compare with another digital signature algorithm, we'll examine ECDSA, which is a cryptographic algorithm used for public key recovery and is the only other elliptic curve supported by the Ethereum Virtual Machine (EVM). The `ecRecover` function is a pre-compiled contract in Ethereum that implements the ECDSA public key recovery process. It is located at address 0x01 and costs 3,000 gas to execute. In our scenario, we found that our approach outperforms `ecRecover` when there are more than 38 signatures to verify:

$$113,000 < 38 \times 3,000$$

5 Example Calculation

In this section, we present the gas cost calculations for calling the:

`castVoteWithReasonAndBlsSig` function

based on our observations. The gas costs for different values of n, the number of signatures, are illustrated in Appendix A through Figure 2 and 3.

For n=1, the gas cost is calculated as follows:

$$\text{Gas Cost} = 94,000 + 113,000 \ = 207,000 \tag{1}$$

The base cost of the function is 94,000 gas, and the cost of verifying a single BLS signature is 113,000 gas, resulting in a total cost of 207,000 gas.

For n=2, the gas cost is calculated as follows:

$$\text{Gas Cost} = 94,000 + 113,000 + 27,330 \times 1 \ = 234,330 \tag{2}$$

The base cost of the function and the verification of BLS signatures remain constant. However, each additional signature beyond the first incurs an extra cost of 27,330 gas, leading to a total cost of 234,330 gas. This increase is mainly attributed to the operations required to loop over the accounts and calculate their token weights in the protocol.

Our observations show that the gas cost for the `castVoteWithReasonAndBlsSig` function scales linearly with the number of signatures, making it a cost-effective approach for verifying multiple signatures in a single transaction.

6 Graphical Representation

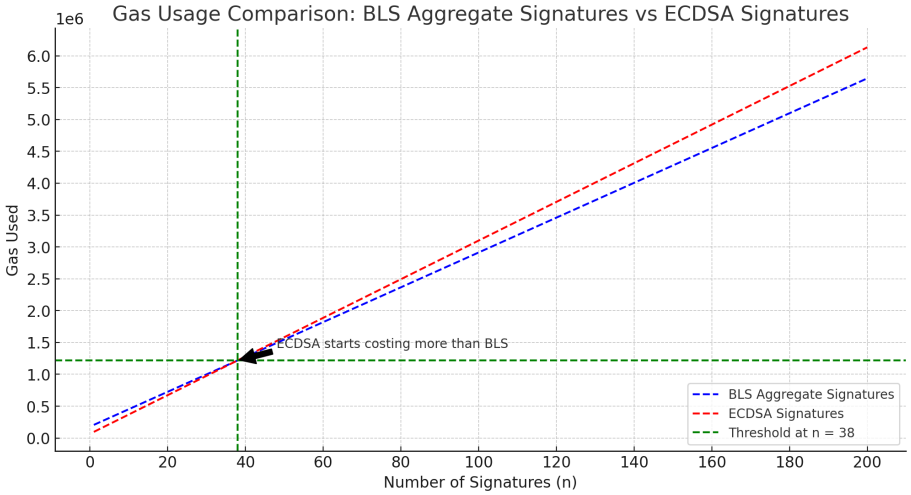

Figure 1: Gas Usage Comparison: BLS Aggregate Signatures vs ECDSA Signatures

7 Future Improvements

The DAO protocol using BLS aggregate signatures on-chain provides a compelling solution for voting scenarios that require signatures. Nevertheless, several important considerations need to be addressed before implementation. The

first step is to generate key pairs and associate them with Ethereum accounts, which can pose challenges. Ideally, enabling direct voting through Ethereum accounts would significantly simplify the process, but this may not be feasible in the near future. In the current scenario, the distribution of key pairs must be meticulously managed to maintain the security and integrity of the voting process.

It is also important to note that even with our signature scheme, calculating the weights of each account can be resource-intensive, as it involves iterating through the accounts. This step is crucial to determine the number of tokens each account holds, influencing their voting weight within the DAO.

Looking ahead, this basic DAO protocol can be expanded to incorporate unique voting mechanisms, such as using NFTs or other emerging solutions, depending on the specific needs of the DAO. Ensuring that each voter has only one vote can significantly enhance both the security and fairness of the voting process. Overall, the use of BLS aggregate signatures in this DAO protocol lays a solid groundwork for future enhancements and improvements.

8 Conclusion

The benchmarks demonstrate that the DAO protocol efficiently handles BLS aggregate signature verification, with a predictable increase in gas usage as the number of aggregated signatures increases. This makes it a suitable choice for secure and scalable voting mechanisms within DAOs. Additionally, it outperforms the `ecRecover` function when there are more than 38 signatures to verify. However, it is important to note that despite these advantages, the protocol can still be computationally intensive. The computation costs, particularly when managing a large number of signatures, might impact its scalability and overall efficiency in larger DAO settings. Future improvements could focus on optimizing these computational demands to enhance performance and feasibility in broader applications.

Special thanks to the contributors at Ethereum Research Forum for their insights on BLS signatures in Solidity, which greatly informed this report.

9 Appendices

Appendix A: Screenshots of transactions with gas usage details

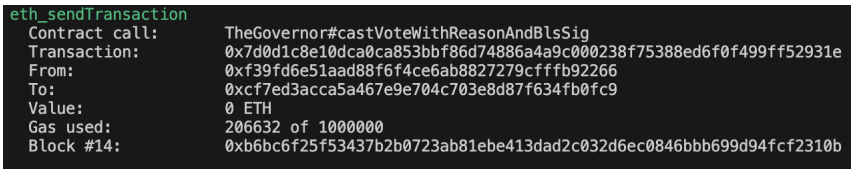

Figure 2: Transaction 1

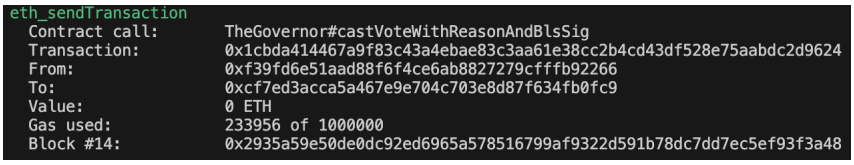

Figure 3: Transaction 2

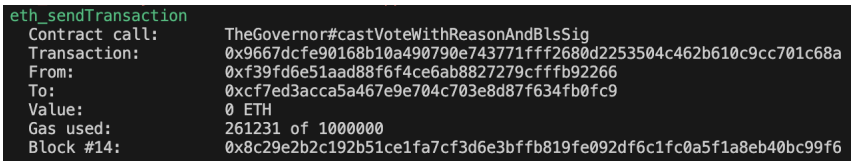

Figure 4: Transaction 3

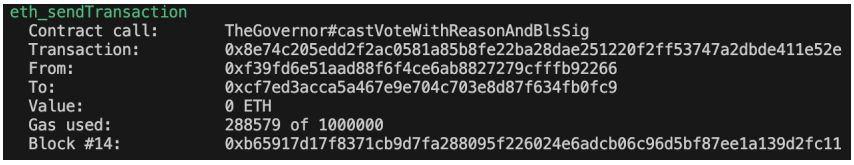

Figure 5: Transaction 4

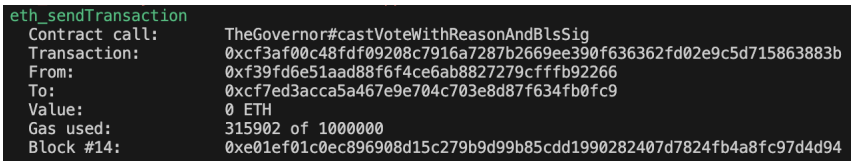

Figure 6: Transaction 5

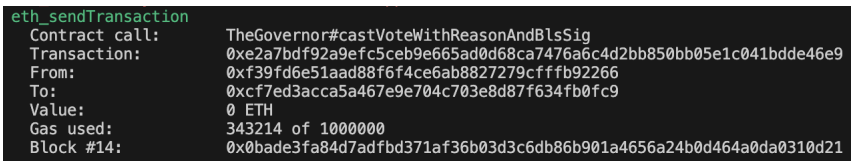

Figure 7: Transaction 6

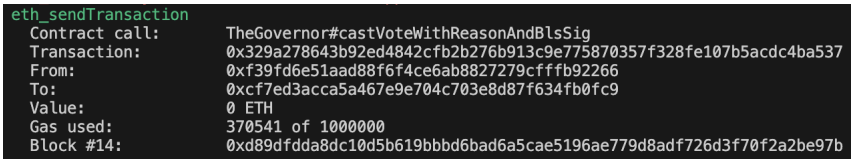

Figure 8: Transaction 7

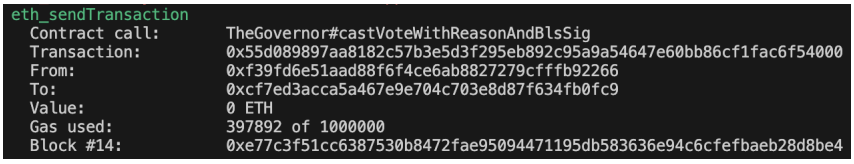

Figure 9: Transaction 8

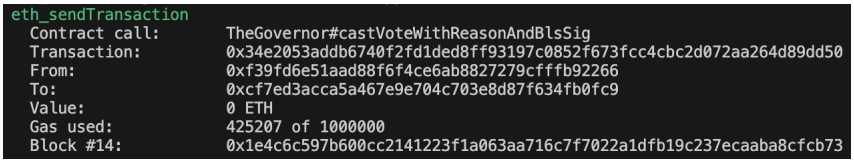

Figure 10: Transaction 9

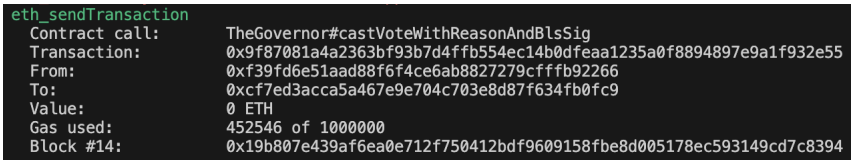

Figure 11: Transaction 10

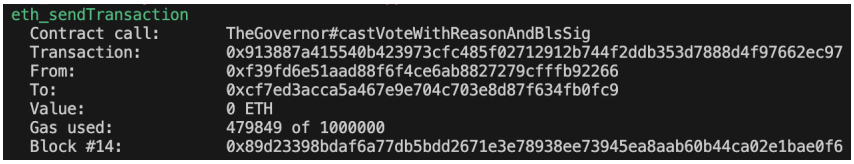

Figure 12: Transaction 11

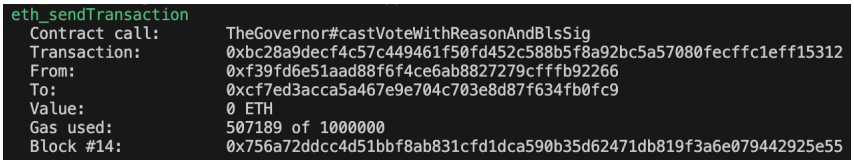

Figure 13: Transaction 12

Appendix B: Source code for the `verifySingle` function

```
function verifySingle(
        uint256[2] memory signature,
        uint256[4] memory pubkey,
        uint256[2] memory message
    ) internal view returns (bool) {
        uint256[12] memory input = [
            signature[0],
            signature[1],
            nG2x1,
            nG2x0,
            nG2y1,
            nG2y0,
            message[0],
            message[1],
            pubkey[1],
            pubkey[0],
            pubkey[3],
            pubkey[2]
        ];
        uint256[1] memory out;
        bool success;
        // Call the precompiled contract at address 8 - pairing
        assembly {
            success := staticcall(
                sub(gas(), 2000), // ensure that there is enough gas left
                8, // pairing precompile address
                input, // Start of input
                384, // 12 * 32 bytes - size of the input
                out, // Write the output
                0x20 // Expect 32 bytes of output
            )
            switch success
            case 0 {
                let size := mload(0x40)
                mstore(size, "Pairing check failed")
                revert(size, 21)
            }
        }
        // Check if the pairing result is true
        if (!success) {
            BLSLibrary._throwSigError(
                BLSLibrary.SignatureError.InvalidSignature,
                0
            );
```

```
        }
        require(success, "");
        return out[0] != 0;
    }
}
```

Appendix C: Detailed calculation steps for gas cost

$$\text{Pairing Costs} = 34,000 \times k + 45,000$$

$$\text{Cast Vote Gas Cost} = 94,000 + 113,000 + 27,330 \times n$$