Solidity SimpleDAO Smart Contract Gas Optimization Report

Contract name: SimpleDAO.sol

Solidity version: 0.8.13

Auditor: Alejandro Durá

Date: January 2026

1. Summary

This report presents the gas optimization analysis of SimpleDAO.sol.

Some inefficiencies were detected, specially in variable packaging and variable lengths. Many of them are all related to storage access.

In some functions, their consumption was increased because of new security improvements, which adds new logic and storage access. So in this case there is no possible gas optimization because of new functionality needed to achieve the security goals.

During the report you will see the cost differences between V1 version and V2 version.

All gas measures were performed using **test/gas/SimpleDAO_GasCostTest.t.sol** and **test/gas/SimpleDAO_Original_GasCostTest.t.sol** files. They can be found in the github repository.

2. Scope and methodology

Scope

- **Contracts:** SimpleDAO_Original.sol and SimpleDAO.sol.
- Difference between both versions costs.

Methodology

- We performed a manual code review to identify possible optimizations, whether in variable access or in the logic.
- We performed storage access evaluations.
- Gas consumption was measured using: Foundry **forge –gas-report,** Solidity version 0.8.13. We always take **the average cost**.
- Two test files were created to measure the gas cost in all functions of both versions: **SimpleDAO_Original_GasCostTest.t.sol** and **SimpleDAO_GasCostTest.t.sol.**

3. Optimization Summary

ID	Location	Optimization	Impact
G-01	Proposal struct	Variable length optimization	Medium
G-02	Proposal struct	Variable packaging	High
G-03	Proposal struct, function and event parameters	String and byte chains	High

4. Detailed Optimizations

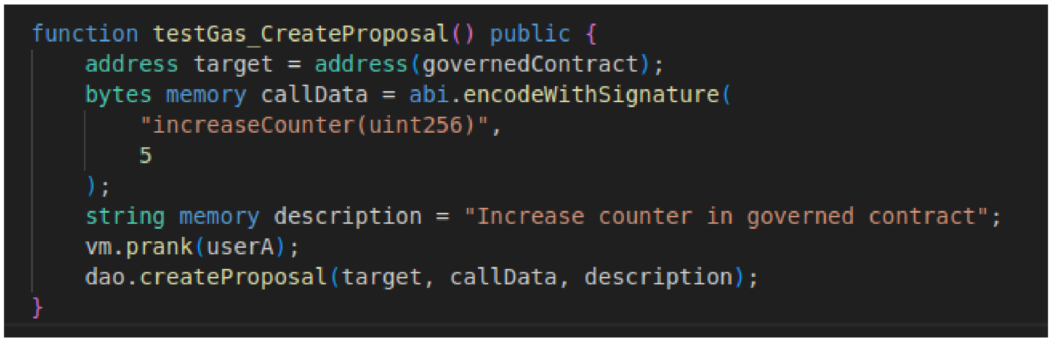
G-03 String and byte chains

- **Location:** Proposal struct, function parameters and event parameters.
- **Problem:** String and byte chains could be very expensive if they are not used properly. If you are not aware about this, you could add a high gas cost overload to your contract functions. That is what is happening here. Specially in the createProposal function
- **Impact:** Big string or byte chains can require a lot of storage slots, making reading and storing operations very expensive.
- **Solution:** There are many solutions: using bytes32 hashes, reducing the bytes or string chains, fixing a max length, etc… We decided to use the bytes32 solution.

The createProposal function is by far the most expensive function in our system because it executes a lot of write operations. This write operation at low level is SSTORE which is very expensive specially in cold writings. That is what is happening in the proposal creation.

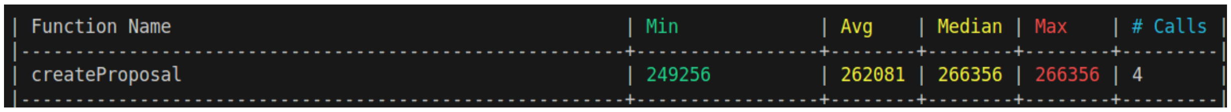
Checking the createProposal function we identified some key aspects that can reduce the costs of creating a new proposal. Lets see what is happening in V1 and the changes made from it to make the V2, which is the optimized one.

v1



```
function testGas_CreateProposal() public {
    address target = address(governedContract);
    bytes memory callData = abi.encodeWithSignature(
        "increaseCounter(uint256)",
        5
    );
    string memory description = "Increase counter in governed contract";
    vm.prank(userA);
    dao.createProposal(target, callData, description);
}
```

With this test function for the createProposal function we obtain the following results:



```
| Function Name                                            | Min             | Avg    | Median | Max    | # Calls |
|----------------------------------------------------------+-----------------+--------+--------+--------+---------|
| createProposal                                           | 249256          | 262081 | 266356 | 266356 | 4       |
|----------------------------------------------------------+-----------------+--------+--------+--------+---------|
```

As we can see calling this test function costs 262081 gas, which is considered a high cost. The reason this function costs a lot is because we are storing the proposal data to the proposals array. We are writing a struct of 8 fields to the storage. The write operation SSTORE is one of the most expensive operations. They cost 20000 gas if we are writing a new value and around 5000 gas if we are updating it. However, we can still implement some design features to reduce the cost.

Write operations in storage have a significant cost, especially if we are writing constantly. In this case, a proposal has 8 fields with different types: address, uint256, bytes, bool,… We can not reduce the number of fields (because all of them are needed for the proper contract behavior) but we certainly can reduce the number of SSTORE operations performed.

To start this optimization, we first focus on the use of bytes and strings. These types are dynamic memory and should be handled properly, specially if we are interested in reducing the gas costs. We are going to start from this point.

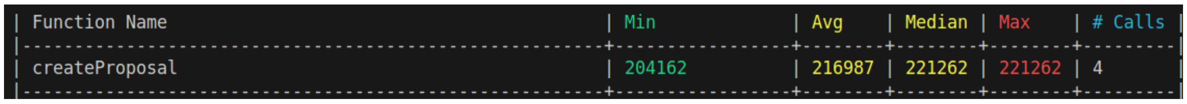
The following changes constitute the createProposal V2 version.

V2

The main goal in this function is to reduce the number of SSTORE operations. Each SSTORE and SLOAD operation works with chunks of 32 bytes of information that is called a slot. Each slot has 32 bytes of information. The problem with strings and bytes is that they can occupy more than one slot depending on their length. Maybe we are not interested in storing large strings. A large string chain could be stored in dozens or hundreds of slots. The more slots required, the greater the number of write and read operations and the higher the gas costs.

In the first gas cost test we created the proposal with the following description: “Increase counter in governed contract” which is 38 bytes, more than one slot. But the reality is that this is not too much.

Let’s see what happens when we reduce the description length and change it by the following: “Inc counter” (11 bytes) which can be filled in one slot.


```
| Function Name                                         | Min             | Avg    | Median | Max    | # Calls |
|-------------------------------------------------------+-----------------+--------+--------+--------+---------|
| createProposal                                        | 204162          | 216987 | 221262 | 221262 | 4       |
|-------------------------------------------------------+-----------------+--------+--------+--------+---------|
```

See that we reduced around 46k units of gas, only reducing the string length description! That’s not too bad!

This means that we can reduce or increase the proposal creation gas cost by varying the proposal description length. The question is: ¿Do we really need a string description in the proposal? Well, this will depend on the smart contract design. If we want to have a string description, then do not worry we can maintain it but be sure it is not too long. Also it is recommended to fix a max string length.

Nevertheless, there is a trick to reducing the cost a little bit more. If we don’t want to have a string description inside of our proposal, we could only save the hash description of this proposal inside the contract and maintain the raw description off-chain indexed by its own hash to retrieve it in the future.

To do this we will replace in our contract the string type to bytes32 in all needed fields, these are: the proposal string description, the ProposalCreated event string description param and the string description param in createProposal function. We do this because all the hashes created by keccak256 have 32 bytes of length. So, whatever the length description is, the hash will always be 32 bytes. This is very powerful, especially if we need to include long descriptions.

We could stay using string, but if we want to have a clear description of the proposal, the best thing is hashing it and storing its bytes hash in a bytes32. This is obviously a design decision that impacts the gas cost. But to be clear and organized I decided to implement this bytes32-hash description solution.

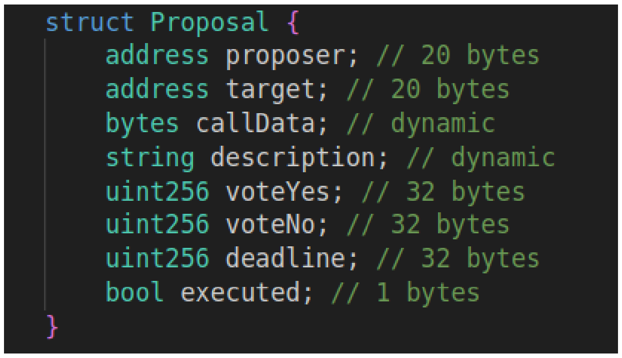
So the first change here in the V2 version is changing string to bytes32 in: the description variable in the Proposal struct, the description parameter in the ProposalCreated event and the description parameter in the createProposal function. That’s all for string and bytes types.

G-01 Variable length optimization

- **Location:** Proposal struct.
- **Problem:** Variable length optimization allows us to reduce the length of the variables inside a proposal and pack them inside a single slot. The problem we have in the V1 version is that no variable is optimized, then the amount of slots used in the Proposal struct is not minimized.
- **Impact:** This gas inefficiency affects the createProposal function where a new proposal is created and all of its variables are stored in the storage. So, if there is not a variable optimization, the createProposal function will be expensive. This will also affect other parts of our contract.
- **Solution:** Identify variables whose length can be reduced.

In the structs, the variables are packaged in slots consecutively. Meaning that the compiler reads the variables “up to down” and picks each variable, reads its length and stores it in a slot. If the variable type is less than 32 bytes then it will be stored in an entire slot. If the following variable can be stored in this previous slot it will be stored, but if not, it will take a new slot. For example: If the variable A is 16 bytes, it will be stored in SLOT0, and the remaining 16 bytes will be 0 (nothing). If the following variable B is 32 bytes, it will be stored in a second slot SLOT1, because the 32 bytes B variable does not fit in the remaining 16 bytes of SLOT0. However, if we can convert variable B to 16 bytes, then it will fit perfectly in the SLOT0 because: 16 bytes (A) + 16 bytes (B) = 32 bytes = slot. This means that we are using only a single slot for 2 variables. We are saving one SSLOAD operation which is expensive.

This behavior is exactly what is happening in our proposal struct which is not optimized for packaging. Initially we had this proposal struct, this is de V1 version:



```
struct Proposal {
    address proposer; // 20 bytes
    address target; // 20 bytes
    bytes callData; // dynamic
    string description; // dynamic
    uint256 voteYes; // 32 bytes
    uint256 voteNo; // 32 bytes
    uint256 deadline; // 32 bytes
    bool executed; // 1 bytes
}
```

See what is happening:

SLOT0: address (20 bytes).

SLOT1: address (20 bytes), it doesn't fit in SLOT0.

SLOT2: bytes is dynamic, SLOT2 and probably more.

SLOT3: string is dynamic, SLOT3 and probably more.

SLOT4: uint256 (32 bytes).

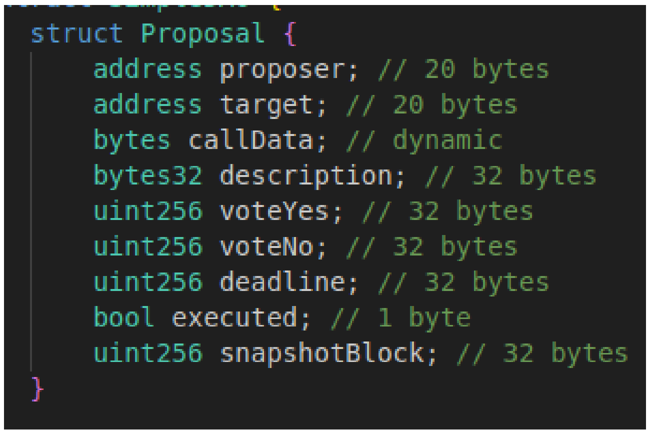
SLOT5: uint256 (32 bytes).

SLOT6: uint256 (32 bytes).

SLOT7: bool (1 byte), SLOT7 it does not fit in the previous one.

We are using 8 SLOTS in total. But we can reduce the number of slots.

In the V2 version we added an extra variable snapshotBlock. This variable is useful to check voting power transfer manipulation and avoid this kind of vulnerability that can produce unexpected behaviour. So it must be added in order to solve this problem. So the Proposal struct looks like this:



```
struct Proposal {
    address proposer; // 20 bytes
    address target; // 20 bytes
    bytes callData; // dynamic
    bytes32 description; // 32 bytes
    uint256 voteYes; // 32 bytes
    uint256 voteNo; // 32 bytes
    uint256 deadline; // 32 bytes
    bool executed; // 1 byte
    uint256 snapshotBlock; // 32 bytes
}
```

The difference is that we added the uint256 snapshotBlock variable at the end. Now we are going to optimize this thing:

See that we are using two variables that can be reduced. This is the case of snapshotBlock which stores the number of a block and the deadline which stores a timestamp in seconds since 1 January 1970.

If we think about it, we don’t really need 32 bytes to store the block number and the deadline. That’s massive!

32 bytes = 256 bits.

2^256 bits =
115792089237316195423570985008687907853269984665640564039457584007
913129639936.

That means we can store **115792089237316195423570985008687907853269984665640564039457584007 913129639936 blocks** and Ethereum blockchain has currently around 20 million blocks mined (20000000 blocks). This number of bits has literally no sense here to store this information.

The same is happening with the deadline, we probably won’t be using this smart contract in:

2^256 bits =
115792089237316195423570985008687907853269984665640564039457584007
913129639936 seconds =
**367046728064246890649896189207176302584236222016360232826880923798
0543488 years.**

Again, this does not make sense.

So, we can freely reduce the length of these variables from 256 bits (32 bytes) to 64 bits (8 bytes). With 64 bits is more than enough (it’s also a lot), we can also reduce the length of snapshotBlock variable to 32 bits (4 bytes). But if we do that, we won’t save more slots and the goal with this technique is to reduce slots. So, we stay with 64 bits. Now we have:

2^64 = 18446744073709551616.

This means that we can store **18446744073709551616 blocks** in the snapshotBlock variable. It’s even unnecessary but as we said, reducing the length will not save more slots. Moreover, supposing that each Ethereum block takes 12 seconds to mine, this means that mining 18446744073709551616 blocks will take:

Time of mining = 18446744073709551616 blocks * 12 seconds/block = 221360928884514619392 seconds = 7012759024870272 years.

With respect to the deadline:

Deadline capacity = 18446744073709551616 – 1 = 18446744073709551615 seconds = 584942417355.0941 years.

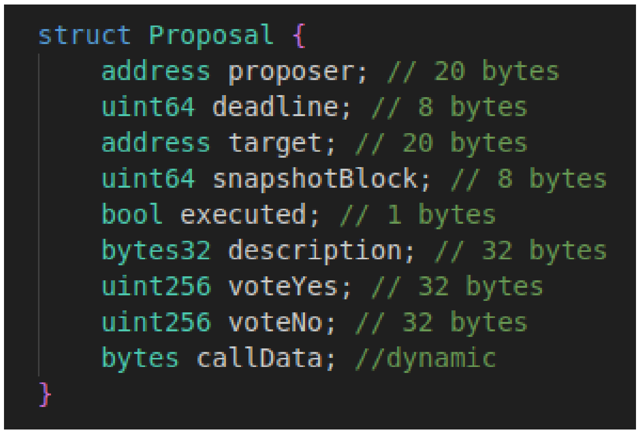
So, we can finally say that reducing these variables from 256 bits to 64 bits is safer.

G-02 Variable packaging

- **Location:** Proposal struct.
- **Problem:** This problem is related to the G-01 (Variable length optimization). Not only do we need to take care about dynamic memory and static memory length, but also the way these variables are packaged in slots. In our V1 version, there is not a variable packing optimization. So there will be additional gas costs if we want to read or write in storage. It is necessary to pack variables in a way they use the fewest amount of slots as possible.
- **Impact:** If we don't know how these variables are stored in slots, then we may probably trigger some extra SLOAD or SSTORE operations that may lead to high gas costs.
- **Solution:** Reorder the variables inside the Proposal struct to align them in the fewest slots as possible.

Now that we have reduced the length of these two variables, we need to reorganize the order of the struct variables in order to package them using the minimum number of slots possible.

The result is the following:



```
struct Proposal {
    address proposer; // 20 bytes
    uint64 deadline; // 8 bytes
    address target; // 20 bytes
    uint64 snapshotBlock; // 8 bytes
    bool executed; // 1 bytes
    bytes32 description; // 32 bytes
    uint256 voteYes; // 32 bytes
    uint256 voteNo; // 32 bytes
    bytes callData; //dynamic
}
```

This reorganization it will be more efficient:

SLOT0: 20 bytes (address) + 8 bytes (uint64) = 28 bytes

SLOT1: 20 bytes (address) + 8 bytes (uint64) + 1 byte (bool) = 29 bytes

SLOT2: 32 bytes (bytes32)

SLOT3: 32 bytes (uint256)

SLOT4: 32 bytes (uint246)

SLOT5: 32 bytes (bytes), but dynamic. Could be more bytes and more slots. One slot as minimum.

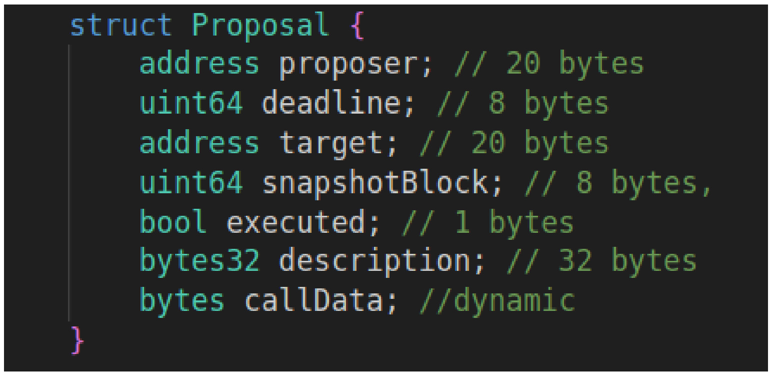
Now we have 6 slots, we reduced in 2 the slot amount from the previous one. Even after adding an additional variable uint256 snapshotBlock.

This is the reorganization inside the same contract. But as we are also making a security audit, we detected that a quorum policy must be implemented in order to make proposal executions safer. So in the V2 version we decided to make a refactor to implement this new behavior and policy in an efficient way.

The V2 has a completely new modular architecture. In order to make security audits more readable, efficient and scalable, we migrated the votes counting data and logic to an abstract contract named VoteManager. So the votes variables in the Proposal struct: voteYes and voteNo are now in a new struct placed in the VoteManager contract. Additionally, to add a quorum logic policy we need to store the token supply at the time the proposal is created. So we need to store a new variable uint256 to register this new data in the proposal data. In this new refactored version the proposal data is splitted in both: **GovernoBase Proposal struct and VoteManager Vote struct.**

So the final proposal data looks like:

GovernorBase:



```
struct Proposal {
    address proposer; // 20 bytes
    uint64 deadline; // 8 bytes
    address target; // 20 bytes
    uint64 snapshotBlock; // 8 bytes,
    bool executed; // 1 bytes
    bytes32 description; // 32 bytes
    bytes callData; //dynamic
}
```

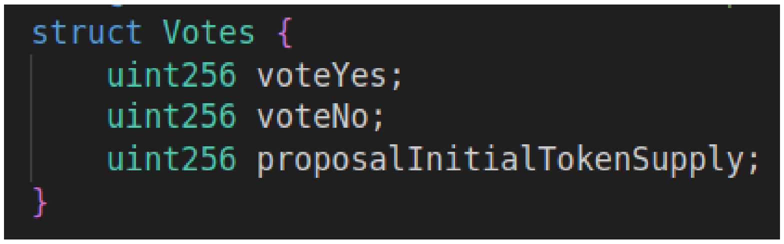
SLOT0: 20 bytes (address) + 8 bytes (uint64) = 28 bytes

SLOT1: 20 bytes (address) + 8 bytes (uint64) + 1 byte (bool) = 29 bytes

SLOT2: 32 bytes (bytes32)

SLOT3: 32 bytes (bytes), but dynamic. Could be more bytes and more slots. One slot as minimum.

VoteManager:



```
struct Votes {
    uint256 voteYes;
    uint256 voteNo;
    uint256 proposalInitialTokenSupply;
}
```

SLOT0: 32 bytes (uint256)

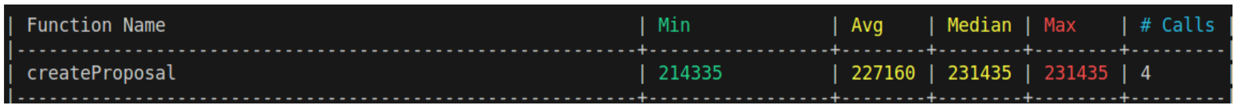
SLOT1: 32 bytes (uint256)

SLOT2: 32 bytes (uint256)

In total, we have now 7 slots storing proposal data in this new V2 refactoring version. Remember the addition of proposalInitialTokenSupply uint256 and snapshotBlock uint64 (optimized) which adds additional SSTORE instruction to be performed. This means the cost will go up, especially because of the uint256 proposalInitialTokenSupply variable. But remember that these two additions are because of security reasons!

Additionally, take in mind that logically, the V2 version makes an additional call to the token to get the past total supply at the specified block number. We need to call this to get the token supply information at the proposal creation time. Also, the V2 version calls the _beforePropose hook, adding a whitelist policy to check if the target in the proposal is allowed. So both, token supply call and hook checks will add an extra gas consumption.

But when we run the test we get the following results:



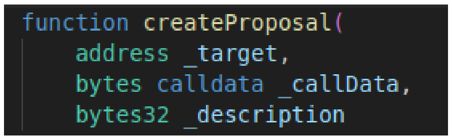
```
| Function Name                                            | Min             | Avg    | Median | Max    | # Calls |
|----------------------------------------------------------+-----------------+--------+--------+--------+---------
| createProposal                                           | 214335          | 227160 | 231435 | 231435 | 4       |
|----------------------------------------------------------+-----------------+--------+--------+--------+---------
```

We get 227160 gas, which is less than the original V1 version. Even after adding two additional variables and an extra logic, we receive less gas consumption. What it really worked here is the variable packaging. The token getPastTotalSupply call function adds an inevitable consume. Adding new variables to the proposal can increase the costs a lot, but if you know how to optimize the slots, you can reduce the gas costs even if you are adding more variables to the struct! And that is what is happening here.

So we increased the contract security and reduced the gas cost in this function!

Well, we reduced the gas units from 262081 gas to 227160 gas. This means we saved 34921 gas!

Another less improvement is changing memory to calldata in the _callData createProposal function parameter:



```
function createProposal(
    address _target,
    bytes calldata _callData,
    bytes32 _description
```

The difference is that we read the _callData information directly from the function ABI and not from the memory of the EVM. The _callData does not need to be copied in memory. This saved us 273 gas.

How much money is this?

Total gas saved = 34921 gas + 273 gas = 35194 gas

Supposing the following information about the Ethereum network:

ETH Price = 3000 USD/ETH

Gwei/Gas = 0.60 gwei

Gas amount = 35194 gas

ETH Cost = 35194 gas * 0.60 gwei/gas = 21116,4 gwei = 0.000021116 ETH

USD Cost = 3000 USD/ETH * 0.000021116 ETH = 0.06335 USD

We save **0.06335 USD** each time we create a new proposal

Percentage reduction = 35K / 262K = 13.4 %

We reduced the cost by 13.4% and the extra security additions did not negatively impact the cost.

5. Functions costs differences

createProposal()

We talk a lot about createProposal during the gas identification and optimization process in the previous chapter. All gas optimizations are resolved by optimizing the Proposal struct and its variables. So the createProposal function has had a huge weight in the gas cost reduction. As you will see before, other contract functions have been automatically optimized because of this Proposal struct packaging.

The differences between V1 and V2 version in the createProposal() function are:

Function name and version	Gas cost avg
createProposal V1	262081 gas
createProposal V2	227160 gas

This is an improvement of **13.4%.**

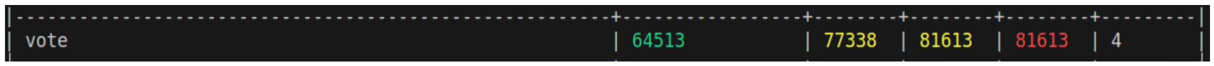
vote()

The vote function cannot be optimized because its behavior is just what we need to vote for a proposal. We are not wasting reading or writing operations. We access the proposal to check its length (necessary), its deadline (necessary) and the votes: yes or no, which is also necessary. Then we also need to read and write the flag hasVoted to check if a user has already voted for the proposal, also necessary. And if the voter has votes to vote.

But in the V2 version, due to security improvements, we changed token.balanceOf() from V1 to token.gePastVotes() in V2. The getPastVotes() function is more expensive than balanceOf(). Moreover, in V2 we need to perform an additional SLOAD reading proposal.snapshotBlock. So, the V2 cost is expected to be higher. But this is for security reasons!

V1

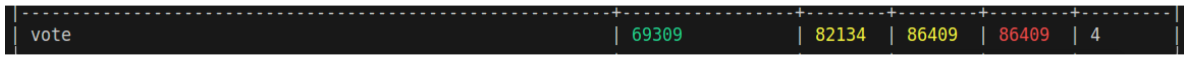
In V1 the vote cost is 77338 gas.



```
|------------------------------------------------------------+-----------------+--------+--------+--------+---------|
| vote                                                       | 64513           | 77338  | 81613  | 81613  | 4       |
```

V2

In V2 the vote is 82134 gas.



```
|-----------------------------------------------------------+-----------------+--------+--------+--------+---------|
| vote                                                      | 69309           | 82134  | 86409  | 86409  | 4       |
```

An increment of 4.8k gas.

Function name and version	Gas cost avg
vote V1	77338 gas
vote V2	82134 gas

In this case there is not a gas optimization. The cost has growed, but because we secured the function.

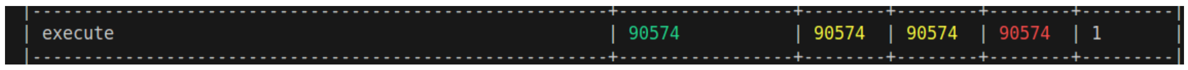
execute()

In the execute function something happens like the last one. We perform 6 read operations SLOAD all of them necessary (7 * ~2100gas = ~14700 gas), 1 write operation SSTORE ~20000 gas, 1 call ~20000-30000 it depends on what the call function does. 1 emit ~375 gas + data cost, it depends on the data emited. And finally, 5 requieres (~700-1000 gas, 5 * 1000 gas = 5000 gas)

Theoretically this function costs approximately ~ 14700 + 20000 + 30000 + 375 + 5000 gas = ~70075 gas. But this theoretical cost is not exactly because we are not measuring the call function costs.

When we execute the gas report, we obtain:

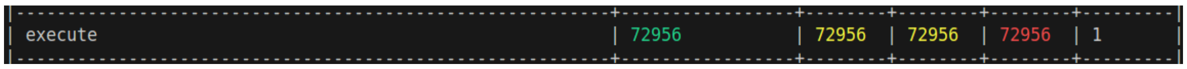
V1



```
|-----------------------------------------------------------+-----------------+--------+--------+--------+---------|
| execute                                                   | 90574           | 90574  | 90574  | 90574  | 1       |
|-----------------------------------------------------------+-----------------+--------+--------+--------+---------|
```

We get 90574 gas. In the V1 version we are calling as a target the GovernedContract which contains a counter that is being updated using the increaseCounter function. The GovernedContract is just a simple contract we made to use it as a target in the proposals. We are using this target and function in the V1 proposal to measure the proposal execution cost.

V2



```
|-------------------------------------------------------------+------------------+--------+--------+--------+---------|
| execute                                                     | 72956            | 72956  | 72956  | 72956  | 1       |
|-------------------------------------------------------------+------------------+--------+--------+--------+---------|
```

In the V2 version we get a lower cost! 72956 gas. That is probably because of two things: One of them is the variable packaging explained before. Variable packaging allows to align variables and pack them in slots in an efficient way. So reading operations can be cheaper. Finally the target call function could have a significant impact. In the V2 version we are using the governor's allowTarget function as the proposal target call function. This function only writes a value in storage. On the other hand, the V1 version uses the counter contract as a target, which reads and writes a value from/to storage. So here, in V1 there are more operations (read from storage and write to storage), thus the cost is higher. That is because we are seeing that difference between them.

In general, we are not expecting an improvement in this function because in the V2 version we added a hook to run the quorum. So the cost in V2 should be a little bit higher for security reasons. But that's ok, as we said before, there is nothing considerable to optimize in this function because what is doing is what we need to accomplish the purpose of a proposal execution.

A quick thing that we can do to prove this is commenting the call lines in the execute function of both versions and check what happens. The results are:

execute V1 = 53867 gas

execute V2 = 42801 gas

Even doing this, the V2 function is cheaper, so we can say that our new V2 refactor and variable packaging has had an influence lowering the gas consumption in the execute proposal function.

But it is important to know that the final cost of the execution function will depend on the target function being called and not only the execute function by itself.

So the real cost will be: execute function gas + target function gas + call instruction gas.

Function name and version	Gas cost avg
execute V1	53867 gas + additional call gas costs
execute V2	42801 gas + additional call gas costs

This represents a base improvement of: **20.5%**

6. Minor changes

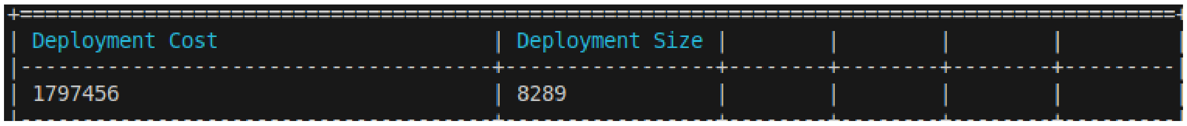
We changed all the function modifiers from public to external. We don't really need to call any contract function from the contract (except the internal ones obviously). All public calls will be external. This will save a little bit of gas.

7. Deployment costs

The deployment costs are related with the amount of bytes the contract occupies. This byte is what it is called the bytecode and represents the contract code (low level). The bytecode is what is stored in the blockchain. So writing bytes in the blockchain costs a lot of gas. The deployment costs will increase depending on the code length, requiere strings, strings, variable initializations, constants, etc…

V1

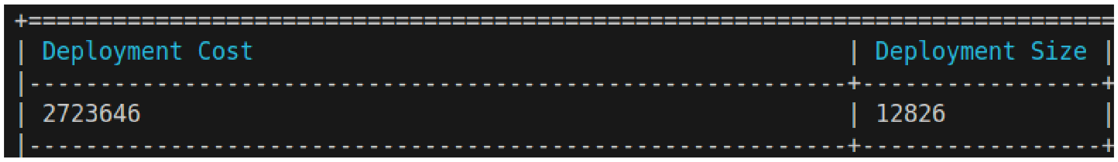
In this case our V1 contract is 8289 bytes and the total gas cost writing these bytes in the blockchain is 1797456 gas.



```
+=====================================================================================
| Deployment Cost                     | Deployment Size |        |        |        |
|-------------------------------------+-----------------+--------+--------+--------+---------
| 1797456                             | 8289            |        |        |        |
```

V2

In the V2 version the deployment costs will definitely go up because this version has more code than V1. It is inevitable that the deployment cost will increase.



```
+=============================================================================
| Deployment Cost                                          | Deployment Size |
|----------------------------------------------------------+-----------------+
| 2723646                                                  | 12826           |
|----------------------------------------------------------+-----------------+
```

In the V2 version the size increased by 4,5k bytes and the cost by 926190 gas. That's an obvious gas cost increase, but we added security to the contract (quorum, whitelist and vote manager) and modularized it to have a better contract architecture. This is better for security audits and future development as we could add new features or functionality such as timelocks without touching too much the contract base code.

But it is important to point out that deployment costs can also be optimized if we reduce the require string errors. The strings are chains of bytes and writing or reading bytes to the blockchain is expensive. To reduce this string's length, we could replace them with certain customized codes that take up less space. Another option is using custom error reverts and replacing all the requires in our code by this new revert method. This could significantly save a lot of deployment gas and also error reverts gas during execution time.

But we are not going to do this now. Maybe in future updates. Only bear in mind that it is possible.

Also using interfaces instead of importing other contract codes inside of our contract is another way to reduce the bytecode length. In our case we are using the IGovernanceToken interface to interact with the ERC20Votes token. We only want the getPastTotalSupply and getPastVotes functions. The rest of the token code inside our contract is unnecessary. Then using an interface is more efficient.

8. Final gas comparison

Function	Original	Optimized	Reduction
createProposal()	262081 gas	227160 gas	-13.4%
execute()	53867 gas	42801 gas	-20.5%

9. Limitations

It is important to know that gas consumptions may vary depending on network conditions and context. Also, different compiler versions could measure different values. Therefore, the results we studied along this report are approximate and not fully accurate.