Gas-wasting Code Smells Demonstrations

December 2023

Jinan Jiang, Zihao Li, Haoran Qin, Muhui Jiang, Xiapu Luo, Xiaoming Wu, Haoyu Wang, Yutian Tang, Chenxiong Qian, Ting Chen

Contents

0.1 Supplementary Materials for the main paper.

In this section, we provide some supplementary discussion for the gas-wasting code smells presented in our main paper. As a summary, we identified 26 gas-wasting code smells, out of which 13 are new to the set of existing works [2, 3, 4, 5, 6, 7, 8, 9, 10, 12] that we compared to. If we exclude the code smells that are tangentially related to existing works, 9 novel code smells remain. These ten existing works were identified following a paper searching methodology similar to that of Systematic Literature Review (i.e., SLR).

0.1.1 Impact Analysis

One limitation associated with our identified code smells is that they might require changes to the orignal codes' functionality. In this section, we examine the impact of our identified code smells on the targeted codes' functionality.

Affects Functionality

3: Other parts of the code that interact with these variables may need to be updated to handle the new types, ensuring compatibility and preventing potential overflows or underflows.
8: Changing the data structure from an array to a map alters how data is stored and accessed.
16: Merging events or changing the amount of data emitted requires changes to the contract's interface.
17: Adding the constant modifier to a state variable prevents it from being modified and should only be done under the premise that the data will not change.
18: Changing a function's visibility from public to external affects how it can be called.
20: Changes how errors are reported.
21: Changed the layout and length of information to be stored.
23: Reordering variables within a struct changes its storage layout.
26: Changing a variable's type from string to bytes32 affects how the variable is used throughout the code, and requires others to adapt accordingly.

Does Not Affect Functionality

1: Only requires adding an intermediate variable.
2: Only wraps the internal repeated code chunks; not visible to how it is used. Does not affect how it is called or used elsewhere.
4: Removing redundant operations does not affect functionality.
5: Pre-computing a fixed-result operation is fully equivalent to original codes.
6: Since the condition doesn't depend on runtime values, simplifying it doesn't impact functionality.
7: Since these are equivalent replacements.
9: Reducing the "require" redundancy is an equivalent change.
10: Eliminating unnecessary variables is fully equivalent.
11: As long as the computation where "unchecked" is to be added is inferred to not overflow/underflow, this is an equivalent operation.
12: This optimization doesn't affect how the array is used elsewhere in the code.
13: Moving require statements in an order-independent manner does not affect functionality.
14: Eliminating assignments where a variable is set to its current value (no-op writes) can be done safely.
15: The order of execution is kept equivalent.
19: Removing dead codes is always equivalent.
22: A local, equivalent operation.

24: Moving code that doesn't change between iterations (loop invariants) outside of the loop optimizes performance without altering the program's behavior.
25: Moving expensive operations within the loop to outside of it, in an equivalent manner, does not affect functionality, and maintains the equivalence at the syntactic level of the loop block.

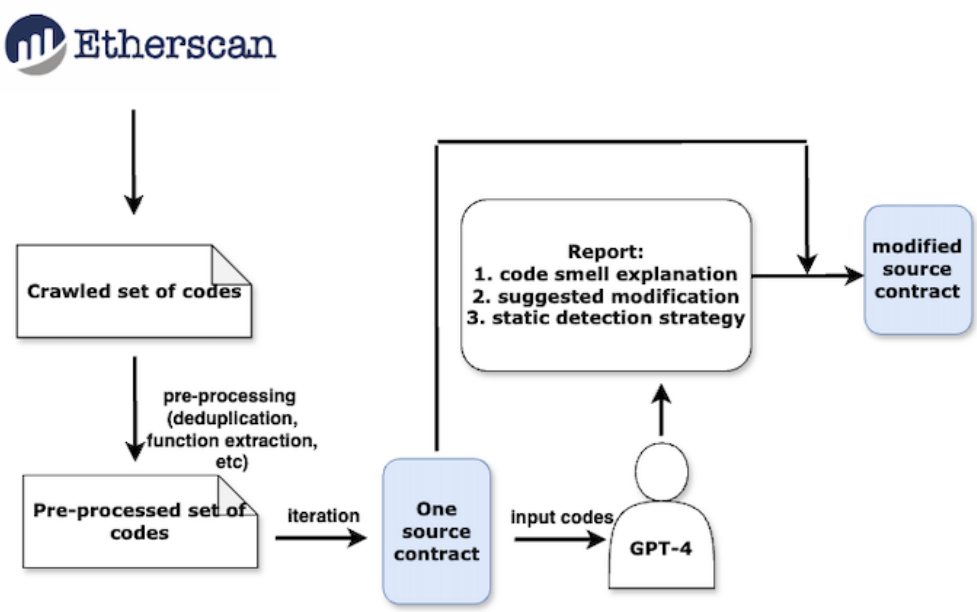

Figure 1: The pattern reporting stage.

0.1.2 Future plan

In this section, we outline our plan for further expanding the gas-wasting code smell detection in the future. We will further automate our current framework presented in the paper to extract gas-wasting patterns from smart contracts at the pattern level, at our supplementary repository: `https://github.com/jinan789/gas_patterns_supplementary`

Figure 1 and Figure 2 illustrate our automated pipeline designed to optimize gas consumption in Solidity smart contracts using a Large Language Model (LLM). The pipeline operates following 2 stages, which we detail below.

Stage 1: The pattern reporting stage.

Figure 1 illustrates our automated pipeline designed to find gas-wasting patterns in Solidity smart contracts using a Large Language Model (LLM). The pipeline operates as follows:

1. **Input Solidity code**: The process begins with the original Solidity source code obtained from Etherscan (or supplied by users), which may contain gas-wasting patterns.
2. **LLM-based detection and suggestion**: The Solidity code is input into a selected LLM (e.g., GPT-4, Claude-3), which is prompted to:
   - Detect and explain *general* gas-wasting patterns applicable across a wide range of smart contracts, such that the suggestions are not tailored to a specific contract.
   - Provide code modification suggestions to eliminate the identified gas-wasting patterns.
   - Provide explanations that how this suggested pattern could be detected via automated strategies using static analysis methods.

   To facilitate the transition from LLM outputs into the modification of source codes, we

design a parseable interface that specifies the output format of the LLM, ensuring that the suggestions can be automatically applied. The LLM's suggestions of modification must adhere to a set of allowed operations:

- **Insertion**
  - Insert [`codes_text`] at line [`line_number`] character [`character_position`]
- **Deletion**
  - Delete from line [`start_line`] character [`start_character`] to line [`end_line`] character [`end_character`]
- **Modification**
  - Replace from line [`start_line`] character [`start_character`] to line [`end_line`] character [`end_character`] with [`new_codes_text`]

3. **Automated code modification**: The suggested changes are parsed and automatically integrated into the original code, producing a modified version intended to be more gas-efficient.
4. **Iterative refinement**: If the modified code fails to compile or execute, the error messages are fed back to the LLM for another round of suggestions. This process repeats up to a maximum number of iterations N (e.g., $N = 3$) to prevent infinite loops.

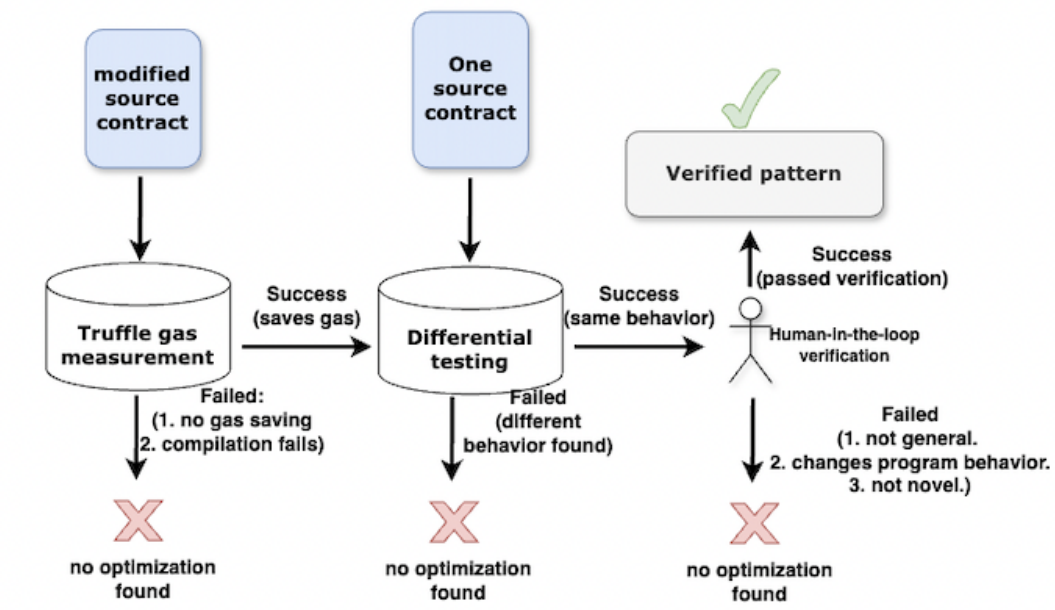

Figure 2: The pattern verification stage.

Stage 2: the pattern verification stage.

To automatically check the proposed gas-wasting patterns proposed by the LLM, we employ a testing framework as shown in Figure 2:

1. **Gas reduction verification**:
   - We execute both the original and modified contracts with a set of representative inputs.
   - We measure and compare their gas consumption to verify that the modifications indeed reduce gas usage.
2. **Functional equivalence testing**:
   - We perform differential testing by fuzzing both versions with the same inputs.
   - We check whether the outputs and behaviors are identical, ensuring that the modifications have not altered the original functionality.

3. **Human analysis**:
    - If both automated stages pass, we conduct an additional round of human review to verify correctness, and identify any subtle issues the automated tests might have missed.

This framework addresses two significant challenges:

- **Challenge 1**: How to automatically check if the pattern reduces gas?
    - **Solution**: By comparing gas usage between the original and modified contracts during testing via the Truffle testing framework, we can confirm the effectiveness of the modifications.
- **Challenge 2**: how to check if the modified codes are functionally identical to the original one?
    - **Solution**: We use differential testing, applying the same inputs to both versions and comparing the outputs. Any discrepancies suggest functional difference.

0.1.3 Descriptions of the supplementary repo scripts

Below, we provide the list of codes and instructions for reproducing our results. The codes that we describe are hosted at `https://github.com/jinan789/gas_patterns_supplementary`.

1. **average_gas.ipynb**: this is the codes we used to obtain table IV of the paper
2. **get_vyper_source.ipynb**: this is the codes we used to
3. **inputs_generation.ipynb**: this contains the codes that we used to generate our inputs to LLM from crawled codes. This notebook also shows the codes that we used to count all of the intermediate contract/function counts that are presented in our methodology section. It also contains codes that we used to extract and count salient functions, which is discussed in our methodology section.
4. **interacting with LLM.ipynb**: this provides the instruction on how to interact with our tested LLMs.
5. **making input for different EVM version.ipynb**: this contains the codes we used to generate the LLM inputs to RQ6-2) Extensibility to new EVM features.
6. **making tables.ipynb**: contains the codes we use to make many tables shown in our paper. Specifically, it accounts for TABLE II: Statistics of each round of GPT-4-facilitated gas-finding, TABLE V: Ablation results, and TABLE VI: Statistics of each round of Claude-3-facilitated gas-finding. it also provides codes for computing the estimated human efforts and the codes for reporting experimental statistics for the Vyper experiment at RQ6-1) Extensibility to other programming languages.
7. **making_vyper_input.ipynb**: this contains the codes we used to replicate our input code extraction process on the main Solidity contracts to the Vyper language.
8. **meta-category of patterns.ipynb**: this provides codes for how we generated Fig. 4: Three types of meta-categorization.
9. **misc.ipynb**: this provides codes for some miscellaneous statistics reported in the paper, including the compiler version counts discussed in Section III-A, and the toy contracts discussion in Section VI-C.
10. **Recording gas cost.ipynb**: this provides the instructions on how to conduct gas consumption measurement on Remix.
11. **config.yaml**: this is where a valid etherscan API key is to be stored at.

We will supplement these scripts with our newest development on further automating our framework to extract gas-wasting patterns from smart contracts at the pattern level, at our supplementary repository: `https://github.com/jinan789/gas_patterns_supplementary`

0.2 Code Smell 1. Repeated computation of the same expression.

Table 1: Gas consumption table for code smell 1.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	231938	220290	**5.0220%**	27154	26391	**2.8098%**
Execution cost	166408	155602	**6.4936%**	6090	5327	**12.528%**

Listing 1: Unoptimized example of Code Smell 1

```
1
2  pragma solidity ^0.8.17;
3
4  contract OperaBaseTokenTaxed {
5      uint256 public _liquidityBuyTax;
6      uint256 public _liquiditySellTax;
7
8      function getTotalTax() internal view returns (uint) {
9          return 300;
10     }
11
12     function tokenSwap() public {
13         uint256 amount = 200;
14
15         uint256 amountToLiquify = (_liquidityBuyTax + _liquiditySellTax > 0)
16             ? amount * (_liquidityBuyTax + _liquiditySellTax) / getTotalTax() / 2
17             : 0;
18         uint256 totalETHFee = (_liquidityBuyTax + _liquiditySellTax > 0)
19             ? getTotalTax() - (_liquidityBuyTax + _liquiditySellTax) / 2
20             : getTotalTax();
21
22         uint256 amountETH = 100;
23         uint256 amountETHLiquidity = amountETH * (_liquidityBuyTax + _liquiditySellTax) / totalETHFee / 2;
24     }
25 }
```

Listing 2: Optimized example of Code Smell 1

```
1
2  pragma solidity ^0.8.17;
3
4  contract OperaBaseTokenTaxed {
5      uint256 public _liquidityBuyTax;
6      uint256 public _liquiditySellTax;
7
8      function getTotalTax() internal view returns (uint) {
9          return 300;
10     }
11
12     function tokenSwap() public {
13         uint256 amount = 200;
14
15         uint256 taxSum = _liquidityBuyTax + _liquiditySellTax;
16         uint256 amountToLiquify = (taxSum > 0)
17             ? amount * (taxSum) / getTotalTax() / 2
18             : 0;
19         uint256 totalETHFee = (taxSum > 0)
20             ? getTotalTax() - (taxSum) / 2
21             : getTotalTax();
22
23         uint256 amountETH = 100;
24         uint256 amountETHLiquidity = amountETH * (taxSum) / totalETHFee / 2;
25     }
26 }
```

1. Code smell explanation. This code smell occurs when there are multiple repetitions of the same costly expression (e.g. mathematical or logical expressions, calls to the same external functions that would produce the same effect, reading from the same storage variable (note that reading from storage variables is a very gas-expensive operation in Solidity), etc). This code smell wastes gas because it performs the same evaluations multiple times. To address it, we could evaluate the repeated costly expression by just once, store its result in an intermediate memory variable, and then replace other repetitions of the same expression by the cached value.

2. Example. To further illustrate this code smell, we take as an example the function *tokenSwap* of the contract *OperaBaseTokenTaxed*, which is deployed at *0x0586638503CCaA365cD8a1338f3b84C54BAe65B3*. The unoptimized codes are shown in Listing 1 and the optimized version is in Listing 2. In addition,

the gas consumption result is listed in Table 1 [1]. It can be observed that the code segment *_liquidityBuyTax + _liquiditySellTax* is repeated 5 times in the presented codes, where 4 such additions could be saved by caching the result of *_liquidityBuyTax + _liquiditySellTax* in an intermediate variable. The gas saving comes from turn accesses to storage variables into memory variables, which is much cheaper.

[1] Note that in this report, $\mathscr{D}_i$ stands for deployment costs, $\mathscr{R}_i$ stands for message call costs, where $i \in \{u, o\}$ with u standing for the unoptimized version and o for the optimized one.

0.3 Code Smell 2. Extractable code chunks.

Table 2: Gas consumption table for code smell 2.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	590280	518438	**12.170%**	30306	30369	**-0.207%**
Execution cost	500672	433810	**13.354%**	7958	8021	**-0.791%**

Listing 3: Unoptimized example of Code Smell 2

```
1
2  pragma solidity ^0.8.13;
3  error OperatorNotAllowed(address a);
4
5  contract IOperatorFilterRegistry {
6      function isOperatorAllowed(address a, address b) public returns (bool) {
7          return true;
8      }
9  }
10
11 contract HolographDropERC721 {
12     IOperatorFilterRegistry public openseaOperatorFilterRegistry =
13         new IOperatorFilterRegistry();
14
15     function msgSender() public returns (address) {
16         return address(this);
17     }
18
19     function beforeSafeTransfer(
20         address _from,
21         address, /* _to*/
22         uint256, /* _tokenId*/
23         bytes calldata /* _data*/
24     ) external returns (bool) {
25         if (
26             _from != address(0) && // skip on mints
27             _from != msgSender() // skip on transfers from sender
28         ) {
29             bool osRegistryEnabled;
30             assembly {
31                 osRegistryEnabled := sload(osRegistryEnabled)
32             }
33             if (osRegistryEnabled) {
34                 try
35                     openseaOperatorFilterRegistry.isOperatorAllowed(
36                         address(this),
37                         msgSender()
38                     )
39                 returns (bool allowed) {
40                     return allowed;
41                 } catch {
42                     revert OperatorNotAllowed(msgSender());
43                 }
44             }
45         }
46         return true;
47     }
48
49     function beforeTransfer(
50         address _from,
51         address, /* _to*/
52         uint256, /* _tokenId*/
53         bytes calldata /* _data*/
54     ) external returns (bool) {
55         if (
56             _from != address(0) && // skip on mints
57             _from != msgSender() // skip on transfers from sender
58         ) {
59             bool osRegistryEnabled;
60             assembly {
61                 osRegistryEnabled := sload(osRegistryEnabled)
62             }
63             if (osRegistryEnabled) {
64                 try
65                     openseaOperatorFilterRegistry.isOperatorAllowed(
66                         address(this),
67                         msgSender()
68                     )
69                 returns (bool allowed) {
70                     return allowed;
71                 } catch {
72                     revert OperatorNotAllowed(msgSender());
73                 }
74             }
75         }
76         return true;
77     }
78 }
```

Listing 4: Optimized example of Code Smell 2

```
pragma solidity ^0.8.13;
error OperatorNotAllowed(address a);

contract IOperatorFilterRegistry {
    function isOperatorAllowed(address a, address b) public returns (bool) {
        return true;
    }
}

contract HolographDropERC721 {
    IOperatorFilterRegistry public openseaOperatorFilterRegistry =
        new IOperatorFilterRegistry();

    function beforeTransferNew(
        address _from,
        address, /* _to*/
        uint256, /* _tokenId*/
        bytes calldata /* _data*/
    ) internal returns (bool) {
        if (
            _from != address(0) && // skip on mints
            _from != msgSender() // skip on transfers from sender
        ) {
            bool osRegistryEnabled;
            assembly {
                osRegistryEnabled := sload(osRegistryEnabled)
            }
            if (osRegistryEnabled) {
                try
                    openseaOperatorFilterRegistry.isOperatorAllowed(
                        address(this),
                        msgSender()
                    )
                returns (bool allowed) {
                    return allowed;
                } catch {
                    revert OperatorNotAllowed(msgSender());
                }
            }
        }
        return true;
    }

    function msgSender() public returns (address) {
        return address(this);
    }

    function beforeSafeTransfer(
        address _from,
        address _to,
        uint256 _tokenId,
        bytes calldata _data
    ) external returns (bool) {
        return beforeTransferNew(_from, _to, _tokenId, _data);
    }

    function beforeTransfer(
        address _from,
        address _to,
        uint256 _tokenId,
        bytes calldata _data
    ) external returns (bool) {
        return beforeTransferNew(_from, _to, _tokenId, _data);
    }
}
```

1. Code smell explanation. This code smell occurs when there are multiple repetitions of the same chunk of codes across multiple functions or within one single function. This code smell wastes gas because the high repitition of codes would increase the cost to deploy the contract. To address this code smell, we could extract the repeated chunks of codes into a separate function, and replace the original chunks of codes by a call to the function. This largely reduced the amount of codes to be deployed, and thus saves deployment gas costs.

2. Example. To further illustrate this code smell, we take as an example the function *beforeSafeTransfer* of the contract *IOperatorFilterRegistry*, which is deployed at *0x257dab74AB23BBF2018C088A29991714ee124* The unoptimized codes are shown in Listing 3 and the optimized version is in Listing 4. In addition, the gas consumption result is listed in Table 2. It can be observed that the functions *beforeSafeTransfer* and *beforeTransfer* have identical code segments, and it would be more maintainbale, readable, as well as gas-efficient if we extract their codes into a new function *beforeTransferNew* and instead call the new function directly. The benefit of maintainability comes from the fact that

future amendments to the codes only need to be performed on one function (i.e. the new function *beforeTransferNew*), instead of to both *beforeSafeTransfer* and *beforeTransfer*. The benefits of gas efficiency comes from saving the amount of bytecodes that need to be stored on-chain as well as from a lower cost of the CODECOPY instruction during deployment.

0.4 Code Smell 3. State Variable Refactoring.

Table 3: Gas consumption table for code smell 3.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	207710	197484	**4.9232%**	34046	24173	**28.999%**
Execution cost	146106	130506	**10.677%**	12982	3109	**76.051%**

Listing 5: Unoptimized example of Code Smell 3

```
1
2  pragma solidity ^0.8.0;
3
4  contract PumpNft {
5      uint256 public gStartsAt;
6      uint256 public gEndsAt;
7      uint256 public fcfsStartsAt;
8      uint256 public fcfsEndsAt;
9      uint256 public publicStartsAt;
10     uint256 public publicEndsAt = type(uint256).max;
11
12     constructor(
13         uint256 _gStartsAt,
14         uint256 _gEndsAt,
15         uint256 _fcfsStartsAt,
16         uint256 _fcfsEndsAt,
17         uint256 _publicStartsAt
18     ) {
19         gStartsAt = _gStartsAt;
20         gEndsAt = _gEndsAt;
21         fcfsStartsAt = _fcfsStartsAt;
22         fcfsEndsAt = _fcfsEndsAt;
23         publicStartsAt = _publicStartsAt;
24     }
25
26     function currentStage() public returns (uint256 stage) {
27         if (block.timestamp >= gStartsAt && block.timestamp <= gEndsAt) {
28             return 1;
29         }
30
31         if (block.timestamp >= fcfsStartsAt && block.timestamp <= fcfsEndsAt) {
32             return 2;
33         }
34
35         if (
36             block.timestamp >= publicStartsAt && block.timestamp <= publicEndsAt
37         ) {
38             return 3;
39         }
40
41         return 0;
42     }
43 }
```

Listing 6: Optimized example of Code Smell 3

```
1
2  pragma solidity ^0.8.0;
3
4  contract PumpNfto {
5      uint32 public gStartsAt;
6      uint32 public gEndsAt;
7      uint32 public fcfsStartsAt;
8      uint32 public fcfsEndsAt;
9      uint32 public publicStartsAt;
10     uint32 public publicEndsAt = type(uint32).max;
11
12     constructor(
13         uint32 _gStartsAt,
14         uint32 _gEndsAt,
15         uint32 _fcfsStartsAt,
16         uint32 _fcfsEndsAt,
17         uint32 _publicStartsAt
18     ) {
19         gStartsAt = _gStartsAt;
20         gEndsAt = _gEndsAt;
21         fcfsStartsAt = _fcfsStartsAt;
22         fcfsEndsAt = _fcfsEndsAt;
23         publicStartsAt = _publicStartsAt;
24     }
25
26     function currentStage() public returns (uint256 stage) {
27         if (block.timestamp >= gStartsAt && block.timestamp <= gEndsAt) {
28             return 1;
29         }
```

```
30
31          if (block.timestamp >= fcfsStartsAt && block.timestamp <= fcfsEndsAt) {
32              return 2;
33          }
34
35          if (block.timestamp >= publicStartsAt && block.timestamp <= publicEndsAt) {
36              return 3;
37          }
38
39          return 0;
40      }
41  }
```

1. Code smell explanation. This code smell seeks to specifically rearrange the layout of the state variables in storage to a more compact form by changing the type of variables (e.g. uint256 to uint32), while ensuring that the changed variable type is still compatible with the task at hand.
2. Example. To further illustrate this code smell, we take as an example the function *currentStage* of the contract *PumpNft*, which is deployed at *0xE3C7b06e06EAc93C9E3B11ea315C838A90CFB4ab*. The unoptimized codes are shown in Listing 5 and the optimized version is in Listing 6. In addition, the gas consumption result is listed in Table 3.

It can be observed that the variables *gStartsAt*, *gEndsAt*, *fcfsStartsAt*, *fcfsEndsAt*, *publicStartsAt*, *publicEndsAt* are all state variables of the PumpNft contract, and are all declared as **unit256**. However, since these variables are used as Unix timestamps to be compared to block timestamps, 256 bits would be too large for this purpose. As a reference, existing mature systems like MariaDB [11] and MongoDB [1] use 4 bytes to store timestamps. Therefore, it is not only safe but also enough to reduce the uint256 data type of the afroementioned state variables to uint32 in this example. Our experiment shows that this contributes to a significant saving of gas (i.e. a saving of 10.677% deployment gas and 76.051% message call gas). In particular, the gas saving comes from repeated warm accesses to the same address in the optimized codes. To be more specific, in the *currentStage* function of the unoptimized contract, since the state variables are declared as uint256, each would occupy a different storage slot. Because of this, during execution, the six accesses to each of the state variables are cold and would cost $12600 = 2100 \times 6$ gas in total. On the other hand, in the optimized version, the six state variables are packed into the same storage slot since each is of a small size of 4 bytes and they altogether could fit in one slot. This means that only the first access to one of the state variables is cold (i.e. access to *gStartsAt*), and all subsequent ones are warm. This way, the cost gets reduced to $2600 = 2100 + 100 \times 5$ gas, which is much lower than the unoptimized version.

It is worth noting that, if uint32 were to be used in the place of uint256, 3 limitations may arise. First, a 32-bit unsigned int can only represent time up to the year of 2106, which means that it will not be valid beyond that point. However, gas could still be saved if a 64-bit unsigned int were used, which can still save gas via a more compact layout and won't expire until 584.9 billion years in the future, which is an astronomical number. Second, this optimization is only positive if consecutive uint256 ints are in the place of storage variables, which won't work for those that are single uint256 variables. Third, it might raise compatibility issues with other parts of the codes that use these variables.

0.5 Code Smell 4. Redundant operations with same effects.

Table 4: Gas consumption table for code smell 4.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	275399	275399	**0.0%**	23647	23631	**0.0676%**
Execution cost	206851	206851	**0.0%**	1847	1831	**0.8662%**

Listing 7: Unoptimized example of Code Smell 4

```
1
2  pragma solidity ^0.8.18;
3
4  contract InitialFairOffering {
5      int24 public constant TICK_SPACING = 60; // Tick space is 60
6
7      function getInitialRate (uint, uint, uint, uint) internal returns (uint) {
8          return 1;
9      }
10
11     function priceToSqrtPriceX96(int, int24) internal returns (uint160) {
12         return 1;
13     }
14
15     function _initializePool(
16         address _weth,
17         address _token
18     )
19         public
20         returns (
21             address _token0,
22             address _token1,
23             uint _uintRate,
24             uint160 _sqrtPriceX96,
25             address _pool
26         )
27     {
28         _token0 = _token;
29         _token1 = _weth;
30
31         _uintRate = getInitialRate(
32             100,
33             200,
34             300,
35             400
36         ); // weth quantity per token
37         require(_uintRate > 0, "uint rate zero");
38
39         if (_token < _weth) {
40             _sqrtPriceX96 = priceToSqrtPriceX96(
41                 int(_uintRate),
42                 TICK_SPACING
43             );
44         } else {
45             _token0 = _weth;
46             _token1 = _token;
47             _uintRate = 10 ** 36 / _uintRate; // token quantity per weth
48             _sqrtPriceX96 = priceToSqrtPriceX96(
49                 int(_uintRate),
50                 TICK_SPACING
51             );
52         }
53     }
54
55 }
```

Listing 8: Optimized example of Code Smell 4

```
1
2  pragma solidity ^0.8.18;
3
4  contract InitialFairOffering {
5      int24 public constant TICK_SPACING = 60; // Tick space is 60
6
7      function getInitialRate (uint, uint, uint, uint) internal returns (uint) {
8          return 1;
9      }
10
11     function priceToSqrtPriceX96(int, int24) internal returns (uint160) {
12         return 1;
13     }
14
15     function _initializePool(
16         address _weth,
17         address _token
```

```
18      )
19          public
20          returns (
21              address _token0,
22              address _token1,
23              uint _uintRate,
24              uint160 _sqrtPriceX96,
25              address _pool
26          )
27      {
28          _uintRate = getInitialRate(
29              100,
30              200,
31              300,
32              400
33          ); // weth quantity per token
34          require(_uintRate > 0, "uint rate zero");
35
36          if (_token < _weth) {
37              _token0 = _token;
38              _token1 = _weth;
39              _sqrtPriceX96 = priceToSqrtPriceX96(
40                  int(_uintRate),
41                  TICK_SPACING
42              );
43          } else {
44              _token0 = _weth;
45              _token1 = _token;
46              _uintRate = 10 ** 36 / _uintRate; // token quantity per weth
47              _sqrtPriceX96 = priceToSqrtPriceX96(
48                  int(_uintRate),
49                  TICK_SPACING
50              );
51          }
52      }
53
54  }
```

1. Code smell explanation. This code smell occurs when there are operations that overwrite the effect of the previous ones, where the previous effect was not utilized in anywhere, rendering the previous one useless. This code smell wastes gas because the repeated computations could be removed without affecting the functionality of the codes.

2. Example. To further illustrate this code smell, we take as an example the function *_initializePool* of the contract *InitialFairOffering*, which is deployed at *0x62700eA68B3DF1Bff05c596734f976f0AD901A4E*. The unoptimized codes are shown in Listing 7 and the optimized version is in Listing 8. In addition, the gas consumption result is listed in Table 4. It can be observed that in the _initializePool function, _token0 and _token1 are assigned values twice under certain conditions (i.e. if the "else" branch is taken). This redundant assignment wastes gas because it overwrites the initial assignments unnecessarily. Instead of assigning _token0 and _token1 at the beginning of the function, we can use a conditional assignment to assign them only once. This way, we could save the gas cost of an additional assignment. In particular, this is done by moving the lines *_token0 = _token;* and *_token1 = _weth;* into the beginning of the if branch.

0.6 Code Smell 5. Pre-computable operations on constants.

Table 5: Gas consumption table for code smell 5.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	305335	293031	**4.0296%**	23584	23288	**1.2550%**
Execution cost	234675	223263	**4.8628%**	2028	1732	**14.595%**

Listing 9: Unoptimized example of Code Smell 5

```
1
2  pragma solidity ^0.8.0;
3
4  contract UNCX_ProofOfReservesV2_UniV3 {
5      event onRemoveFee(bytes32);
6      function contains (bytes32) internal returns (bool) {
7          return true;
8      }
9
10     function remove (bytes32) internal returns (bool) {
11         return true;
12     }
13
14     function removeFee (string memory _name) external {
15         bytes32 nameHash = keccak256(abi.encodePacked(_name));
16         require(nameHash != keccak256(abi.encodePacked("DEFAULT")), "DEFAULT");
17         require(contains(nameHash));
18         remove(nameHash);
19         emit onRemoveFee(nameHash);
20     }
21 }
```

Listing 10: Optimized example of Code Smell 5

```
1
2  pragma solidity ^0.8.0;
3
4  contract UNCX_ProofOfReservesV2_UniV3 {
5      event onRemoveFee(bytes32);
6      function contains (bytes32) internal returns (bool) {
7          return true;
8      }
9
10     function remove (bytes32) internal returns (bool) {
11         return true;
12     }
13
14     function removeFee (string memory _name) external {
15         bytes32 nameHash = keccak256(abi.encodePacked(_name));
16         require(nameHash != 0x9F28225C7D0ACE67FA2516BD7725F3949E9A591DE0EAE9DB822B2CB79F38A6B0, "DEFAULT");
17         require(contains(nameHash));
18         remove(nameHash);
19         emit onRemoveFee(nameHash);
20     }
21 }
```

1. Code smell explanation. This code smell occurs when there are operations (e.g. logical comparisons, mathematical operations, a keccak256 hash operation on constants, etc) that are performed on constants whose values could be inferred without being compiled. This code smell wastes gas because such computations could be carried out before deploying the contract, which saves the gas consumption during runtime.

2. Example. To further illustrate this code smell, we take as an example the function *removeFee* of the contract *UNCX_ProofOfReservesV2_UniV3*, which is deployed at *0x7f5C649856F900d15C83741f45AE46f5C68582*. The unoptimized codes are shown in Listing 9 and the optimized version is in Listing 10. In addition, the gas consumption result is listed in Table 5. It can be observed that there is a check of whether the variable *nameHash* (i.e. the hash of the input string *_name*) is the same as the hash of the string "DEFAULT". In other words, it is checking if the input variable *_name* has the same hashcode as the string "DEFAULT". Note that each time the comparison is done, the string "DEFAULT" is first encoded and then hashed; this is highly repetitive since the hashcode could be computed offline and placed in the codes directly. To save gas, we directly replace the codes *keccak256(abi.encodePacked("DEFAULT"))* by the corresponding hashcode of "DEFAULT" (i.e.

0x9F28225C7D0ACE67FA2516BD7725F3949E9A591DE0EAE9DB822B2CB79F38A6B0). One might argue that this constitutes a compromise to code readability, but a line of comment could be readily added to elaborate the purpose of this line of code, which still maximally retains the clarity without compromising gas costs.

0.7 Code Smell 6. Deterministic conditional checks.

Table 6: Gas consumption table for code smell 6.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	243762	240078	**1.5113%**	23691	23651	**0.1688%**
Execution cost	177820	174420	**1.9120%**	2487	2447	**1.6083%**

Listing 11: Unoptimized example of Code Smell 6

```
1
2  pragma solidity ^0.8.18;
3
4  contract InitialFairOffering {
5      function addLiquidity(uint16 slippage) public {
6          uint256 balanceOfWeth = 100;
7          uint256 liquidityEtherPercent = 200;
8          uint256 maxRollups = 300;
9          uint256 fundingCommission = 400;
10         uint256 crowdFundingRate = 500;
11
12
13         require(slippage >= 0 && slippage <= 10000, "slippage error");
14
15         // Send ether back to deployer, the eth liquidity is based on the balance of this contract. So,
               anyone can send eth to this contract
16         uint256 backToDeployAmount = (balanceOfWeth *
17             (10000 - liquidityEtherPercent)) / 10000;
18         uint256 maxBackToDeployAmount = (maxRollups *
19             (10000 - fundingCommission) *
20             crowdFundingRate *
21             (10000 - liquidityEtherPercent)) / 100000000;
22
23     }
24 }
```

Listing 12: Optimized example of Code Smell 6

```
1
2  pragma solidity ^0.8.18;
3
4  contract InitialFairOffering {
5      function addLiquidity(uint16 slippage) public {
6          uint256 balanceOfWeth = 100;
7          uint256 liquidityEtherPercent = 200;
8          uint256 maxRollups = 300;
9          uint256 fundingCommission = 400;
10         uint256 crowdFundingRate = 500;
11
12
13         require(slippage <= 10000, "slippage error");
14
15         // Send ether back to deployer, the eth liquidity is based on the balance of this contract. So,
               anyone can send eth to this contract
16         uint256 backToDeployAmount = (balanceOfWeth *
17             (10000 - liquidityEtherPercent)) / 10000;
18         uint256 maxBackToDeployAmount = (maxRollups *
19             (10000 - fundingCommission) *
20             crowdFundingRate *
21             (10000 - liquidityEtherPercent)) / 100000000;
22
23     }
24 }
```

1. Code smell explanation. This code smell occurs when some of the conditional checks always evaluate to either true or false. In other words, the value of such conditional expressions could be logically inferred (e.g. from variable types) without actually running the codes, regardless of the input values. This code smell wastes gas because the conditional expressions could be just replaced by the corresponding true/false value, without having to go through the actual computation.
2. Example. To further illustrate this code smell, we take as an example the function *addLiquidity* of the contract *InitialFairOffering*, which is deployed at *0x62700eA68B3DF1Bff05c596734f976f0AD901A4E*. The unoptimized codes are shown in Listing 11 and the optimized version is in Listing 12. In addition, the gas consumption result is listed in Table 6.

It can be observed that In the addLiquidity function, there's a check for *slippage >= 0*. Since slippage is a uint16, it can never be less than zero. Note that if we force the argument (i.e. slippage)

to be negative (e.g. -1), we would get the error: "transact to InitialFairOffering.addLiquidity errored: Error encoding arguments: Error: value out-of-bounds (argument=null, value="-1", code=INVALID_ARGUMENT, version=abi/5.7.0)." As a result, we could just remove this conditional check to save computation.

0.8 Code Smell 7. Conditional statements with simpler equivalents.

Table 7: Gas consumption table for code smell 7.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	232355	228254	**1.7649%**	24142	24094	**0.1988%**
Execution cost	166957	163150	**2.2802%**	2798	2750	**1.7155%**

Listing 13: Unoptimized example of Code Smell 7

```
1
2  pragma solidity ^0.8.0;
3
4  contract presale {
5      address public presale_owner = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
6
7      function updateRatePresale(uint256 _rate, uint256 _rateStable) external {
8          bool isOwner = false;
9
10         if(msg.sender == presale_owner) {
11             isOwner = true;
12         }
13
14         require(isOwner == true, "Requires owner");
15
16         uint256 ratePresale; //listing price in wei
17         uint256 ratePresaleStable;
18
19         ratePresale = _rate;
20         ratePresaleStable = _rateStable;
21     }
22 }
```

Listing 14: Optimized example of Code Smell 7

```
1
2  pragma solidity ^0.8.0;
3
4  contract presale {
5      address public presale_owner = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
6
7      modifier owner {
8          require(msg.sender == presale_owner, "Requires owner");
9          _;
10     }
11
12     function updateRatePresale(uint256 _rate, uint256 _rateStable) external owner {
13         uint256 ratePresale; //listing price in wei
14         uint256 ratePresaleStable;
15
16         ratePresale = _rate;
17         ratePresaleStable = _rateStable;
18     }
19 }
```

1. Code smell explanation. This code smell occurs when the codes contain expressions that involve logical operations that could be simplified to an equivalent form with lower gas costs.
2. Example. To further illustrate this code smell, we take as an example the function *owner* of the contract *presale*, which is deployed at *0x846bB98EA9BD5e766d5FDB1a415E0cf0202D3801*. The unoptimized codes are shown in Listing 13 and the optimized version is in Listing 14. In addition, the gas consumption result is listed in Table 7. It can be observed that the if statement in the codes is redundant, where it first assigns *isOwner* to false, changes it to true under certain conditions, and finally places *isOwner* as the condition of a require statement. Instead, it would be both more readable and less gas-costly if all those operations could be condensed into one line as *require(msg.sender == presale_owner, "Requires owner");* This way, we get to save the logical comparisons.

0.9 Code Smell 8. Replacing item-by-item iterated arrays by a map.

Table 8: Gas consumption table for code smell 8.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	630622	475076	**24.665%**	96842	93068	**3.8970%**
Execution cost	536968	392430	**26.917%**	99428	94711	**4.7441%**

Listing 15: Unoptimized example of Code Smell 8

```
pragma solidity ^0.8.0;

contract DigiMonkzStaking {

    mapping(address => uint16[]) public gen1StakedArray;
    mapping(uint16 => bool) public isGen1Staked;

    function gen1IndividualStake(uint16 _tokenId) private {
        require(isGen1Staked[_tokenId] == false);
        gen1StakedArray[msg.sender].push(_tokenId);
        isGen1Staked[_tokenId] = true;
        // emit Stake(_tokenId);
    }


    function gen1Stake(uint16[] memory _tokenIds) private returns (bool) {
        uint256 tokenLen = _tokenIds.length;
        for (uint256 i = 0; i < tokenLen; i++) {
            gen1IndividualStake(_tokenIds[i]);
        }
        return true;
    }

    function gen1IndividualUnstake(uint16 _tokenId) private {

        uint256 len = gen1StakedArray[msg.sender].length;
        require(len != 0);

        uint256 idx = len;
        for (uint16 i = 0; i < len; i++) {
            if (gen1StakedArray[msg.sender][i] == _tokenId) {
                idx = i;
            }
        }
        require(idx != len);

        // uint256 stakedTime = gen1InfoPerStaker[msg.sender][idx].stakedAt;
        if (idx != len - 1) {
            gen1StakedArray[msg.sender][idx] = gen1StakedArray[msg.sender][
                len - 1
            ];
        }

        gen1StakedArray[msg.sender].pop();
        isGen1Staked[_tokenId] = false;

        // emit Unstake(_tokenId, stakedTime, block.timestamp);
    }

    function gen1Unstake(uint16[] memory _tokenIds) external returns (bool) {
        // for testing purposes, we first load the token ids
        gen1Stake(_tokenIds);
        uint256 tokenLen = _tokenIds.length;
        for (uint256 i = 0; i < tokenLen; i++) {
            gen1IndividualUnstake(_tokenIds[i]);
        }
        return true;
    }
}
```

Listing 16: Optimized example of Code Smell 8

```
pragma solidity ^0.8.0;

contract DigiMonkzStaking {

    mapping(address => mapping(uint16 => bool)) public gen1StakedArray;
    mapping(uint16 => bool) public isGen1Staked;

    function gen1IndividualStake(uint16 _tokenId) private {
        require(isGen1Staked[_tokenId] == false);
        gen1StakedArray[msg.sender][_tokenId] = true;
        isGen1Staked[_tokenId] = true;
```

```
13          // emit Stake(_tokenId);
14      }
15
16      function gen1Stake(uint16[] memory _tokenIds) private returns (bool) {
17          uint256 tokenLen = _tokenIds.length;
18          for (uint256 i = 0; i < tokenLen; i++) {
19              gen1IndividualStake(_tokenIds[i]);
20          }
21          return true;
22      }
23
24      function gen1IndividualUnstake(uint16 _tokenId) private {
25          require (gen1StakedArray[msg.sender][_tokenId] == true);
26
27          gen1StakedArray[msg.sender][_tokenId] = false;
28          isGen1Staked[_tokenId] = false;
29
30          // emit Unstake(_tokenId, stakedTime, block.timestamp);
31      }
32
33      function gen1Unstake(uint16[] memory _tokenIds) external returns (bool) {
34          // for testing purposes, we first load the token ids
35          gen1Stake(_tokenIds);
36          uint256 tokenLen = _tokenIds.length;
37          for (uint256 i = 0; i < tokenLen; i++) {
38              gen1IndividualUnstake(_tokenIds[i]);
39          }
40          return true;
41      }
42  }
```

1. Code smell explanation. This code smell occurs when there is an array that often gets iterated to extract particular elements (i.e. O(N) to find a result), while the same effect could be achieved by implementing the data structure instead as a map (i.e. O(1) to find a result). Arrays are great for storing ordered data but can be gas-inefficient when used for lookups and deletions.

2. Example. To further illustrate this code smell, we take as an example the function *gen1IndividualUnstake* of the contract *DigiMonkzStaking*, which is deployed at *0x077B1BB5Aa45A907866cd8338592b6B2080EF747*. The unoptimized codes are shown in Listing 15 and the optimized version is in Listing 16. In addition, the gas consumption result is listed in Table 8. In particular, in the optimized version, we get rid of the for loop and instead find the idx variable using a map, which is much more gas-friendly.

0.10 Code Smell 9. Repeated security checks across function calls.

Table 9: Gas consumption table for code smell 9.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	428902	381781	**10.986%**	26312	26242	**0.2660%**
Execution cost	350386	306143	**12.626%**	4384	4314	**1.5967%**

Listing 17: Unoptimized example of Code Smell 9

```
1
2  pragma solidity ^0.8.0;
3
4  contract OperaBaseTokenTaxed{
5      mapping(address => mapping(address => uint256)) _allowances;
6      uint256 public _totalSupply;
7      mapping(address => uint256) _balances;
8
9      function transferFrom(
10         address sender,
11         address recipient,
12         uint256 amount
13     ) external returns (bool) {
14         require(sender != address(0), "ERC20: transfer from the zero address");
15         require(recipient != address(0), "ERC20: transfer to the zero address");
16
17         return _transferFrom(sender, recipient, amount);
18     }
19
20     function _transferFrom(
21         address sender,
22         address recipient,
23         uint256 amount
24     ) internal returns (bool) {
25         require(sender != address(0), "ERC20: transfer from the zero address");
26         require(recipient != address(0), "ERC20: transfer to the zero address");
27
28         _balances[sender] = _balances[sender] - amount;
29
30         _balances[recipient] = _balances[recipient] + amount;
31
32         return true;
33     }
34 }
```

Listing 18: Optimized example of Code Smell 9

```
1
2  pragma solidity ^0.8.0;
3
4  contract OperaBaseTokenTaxed{
5      mapping(address => mapping(address => uint256)) _allowances;
6      uint256 public _totalSupply;
7      mapping(address => uint256) _balances;
8
9      function transferFrom(
10         address sender,
11         address recipient,
12         uint256 amount
13     ) external returns (bool) {
14         return _transferFrom(sender, recipient, amount);
15     }
16
17     function _transferFrom(
18         address sender,
19         address recipient,
20         uint256 amount
21     ) internal returns (bool) {
22         require(sender != address(0), "ERC20: transfer from the zero address");
23         require(recipient != address(0), "ERC20: transfer to the zero address");
24
25         _balances[sender] = _balances[sender] - amount;
26
27         _balances[recipient] = _balances[recipient] + amount;
28
29         return true;
30     }
31 }
```

1. Code smell explanation. This code smell occurs when in function calls, the caller first performs some kind of a security check and then immediately followed by a call to another function in which the same set of security checks are performed. This constitutes a repeated security check upon entry, and the caller's security check could be removed.

2. Example. To further illustrate this code smell, we take as an example the function *transferFrom* of the contract *OperaBaseTokenTaxed*, which is deployed at *0x0586638503CCaA365cD8a1338f3b84C54BAe65B3*. The unoptimized codes are shown in Listing 17 and the optimized version is in Listing 18. In addition, the gas consumption result is listed in Table 9.

In the unoptimized contract, the *transferFrom* function conducts two security checks, represented by "require" statements. These checks are then repeated in the immediately following invocation of the *_transferFrom* function, unnecessarily resulting in redundancy. To optimize this, we eliminated the checks within the transferFrom function.

It's important to note that we didn't remove the checks from the *_transferFrom* function, as we can't be certain all callers of this function will consistently perform the necessary security checks. However, we have confidence in removing checks from the *transferFrom* function given that it consistently calls *_transferFrom*.

0.11 Code Smell 10. Unnecessarily introducing variables.

Table 10: Gas consumption table for code smell 10.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	287979	276947	**3.8308%**	24966	24852	**0.4566%**
Execution cost	218263	208051	**4.6787%**	1790	1676	**6.3687%**

Listing 19: Unoptimized example of Code Smell 10

```
1
2  pragma solidity ^0.8.0;
3
4  contract Donate3  {
5      function _transferToken(
6          address token,
7          address from,
8          uint256 amountInDesired,
9          address rAddress,
10         bytes calldata merkleProof
11     ) internal returns (uint256 amountOut) {
12         return 1;
13     }
14
15     function donateERC20(
16         address _token,
17         string calldata _tokenSymbol,
18         uint256 _amountInDesired,
19         address _to,
20         bytes calldata _message,
21         bytes calldata _merkleProof
22     ) external {
23         address from = msg.sender;
24         string calldata symbol = _tokenSymbol;
25         bytes calldata message = _message;
26         address token = _token;
27         bytes calldata merkleProof = _merkleProof;
28         uint256 amountInDesired = _amountInDesired;
29
30         address to = _to;
31         require(from != to, "The donor address is equal to receive");
32
33         uint256 amountOut = _transferToken(
34             token,
35             from,
36             amountInDesired,
37             to,
38             merkleProof
39         );
40     }
41 }
```

Listing 20: Optimized example of Code Smell 10

```
1
2  pragma solidity ^0.8.0;
3
4  contract Donate3  {
5      function _transferToken(
6          address token,
7          address from,
8          uint256 amountInDesired,
9          address rAddress,
10         bytes calldata merkleProof
11     ) internal returns (uint256 amountOut) {
12         return 1;
13     }
14
15     function donateERC20(
16         address _token,
17         string calldata _tokenSymbol,
18         uint256 _amountInDesired,
19         address _to,
20         bytes calldata _message,
21         bytes calldata _merkleProof
22     ) external {
23         address from = msg.sender;
24         require(from != _to, "The donor address is equal to receive");
25
26         uint256 amountOut = _transferToken(
27             _token,
28             from,
29             _amountInDesired,
30             _to,
31             _merkleProof
```

```
32            );
33        }
34    }
```

1. Code smell explanation. This code smell occurs when codes introduce new but unnecessarily derived variables, where just using the original one would have the same effect. Such new variables could be removed to save gas.

2. Example. To further illustrate this code smell, we take as an example the function *donateERC20* of the contract *Donate3*, which is deployed at *0x3a42ddc676f68547301517 50f3dbd0ebfe3c6cd3*. The unoptimized codes are shown in Listing 19 and the optimized version is in Listing 20. In addition, the gas consumption result is listed in Table 10. In particular, in the unoptimized contract, it can be observed that multiple variables like symbol, message, token, merkleProof, and amountInDesired are just straightforward duplicates of their corresponding input variables. This extra step is not necessary as defining new variables consumes gas, and we remove this part in the optimized codes.

0.12 Code Smell 11. Unnecessary overflow/underflow validation since Solidity 0.8.0

Table 11: Gas consumption table for code smell 11.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	273423	246758	**9.7522%**	37597	36997	**1.5958%**
Execution cost	204845	179826	**12.213%**	13553	12953	**4.4270%**

Listing 21: Unoptimized example of Code Smell 11

```
pragma solidity ^0.8.0;

contract KSElasticLMV2 {
    event WithdrawUnusedRewards(
        address token,
        uint256 amount,
        address receiver
    );

    function withdrawUnusedRewards(
        address[] calldata tokens,
        uint256[] calldata amounts
    ) external {
        uint256 rewardTokenLength = tokens.length;
        for (uint256 i; i < rewardTokenLength; ) {
            emit WithdrawUnusedRewards(tokens[i], amounts[i], msg.sender);
            ++i;
        }
    }
}
```

Listing 22: Optimized example of Code Smell 11

```
pragma solidity ^0.8.0;

contract KSElasticLMV2 {
    event WithdrawUnusedRewards(
        address token,
        uint256 amount,
        address receiver
    );

    function withdrawUnusedRewards(
        address[] calldata tokens,
        uint256[] calldata amounts
    ) external {
        uint256 rewardTokenLength = tokens.length;
        for (uint256 i; i < rewardTokenLength; ) {
            emit WithdrawUnusedRewards(tokens[i], amounts[i], msg.sender);

            unchecked {
                ++i;
            }
        }
    }
}
```

1. Code smell explanation. Since Solidity 0.8.0, over- and unver-flow checks are inherently integrated and using safemath or manual validation becomes unnecessary. This means that if we are confident that our usage of the variable will not cause over- or under-flow for certain code lines, then we should just use the "unchecked" keyword on the corresponding lines to save the inherently integrated checks.

2. Example. To further illustrate this code smell, we take as an example the function *withdrawUnusedRewards* of the contract *KSElasticLMV2*, which is deployed at *0x3D6AfE2fB73fFEd2E3dD00c501A174554*
The unoptimized codes are shown in Listing 21 and the optimized version is in Listing 22. In addition, the gas consumption result is listed in Table 11. Note that in this example, we use the original contract as the optimized codes and a modified contract (i.e. with the "unchecked" keyword removed) as the unoptimized codes. In particular, The gas is saved because in the optimized version, there are no longer repeated checks for overflow and underflow upon each iteration of the loop.

0.13 Code Smell 12. Redundant memory array initialization.

Table 12: Gas consumption table for code smell 12.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	153465	100093	**34.777%**	21755	21463	**1.3422%**
Execution cost	93541	43293	**53.717%**	691	399	**42.257%**

Listing 23: Unoptimized example of Code Smell 12

```
1
2 pragma solidity ^0.8.17;
3
4 contract AaveV2Strategy {
5     function assetRatio() public returns (uint256) {
6         uint256[] memory _assetRatio = new uint256[](3);
7         _assetRatio[0] = 1;
8         _assetRatio[1] = 2;
9         _assetRatio[2] = 3;
10        return 1;
11    }
12 }
```

Listing 24: Optimized example of Code Smell 12

```
1
2 pragma solidity ^0.8.17;
3
4 contract AaveV2Strategy {
5     function assetRatio() public returns (uint256) {
6         uint256[3] memory _assetRatio = [uint256(1),2,3];
7         return 1;
8     }
9 }
```

1. Code smell explanation. This code smell occurs when new memory arrays are initialized with a fixed size and then manually populated with values. A more gas-efficient approach is to initialize the array with populated values in one step (e.g. direct initialization like [1,2]).
2. Example. To further illustrate this code smell, we take as an example the function *assetRatio* of the contract *AaveV2Strategy*, which is deployed at *0x331c27d9daf6d8f6a2dbf3c16b5c5733da1b4431*. The unoptimized codes are shown in Listing 23 and the optimized version is in Listing 24. In addition, the gas consumption result is listed in Table 12. It can be observed that the optimized version initializes and populates the fixed-size array in just one step.

0.14 Code Smell 13. Placement of require statements.

Table 13: Gas consumption table for code smell 13.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	431521	431533	**-0.002%**	31680	30834	**2.6704%**
Execution cost	352233	352233	**0.0%**	4444	3598	**19.036%**

Listing 25: Unoptimized example of Code Smell 13

```
1
2  pragma solidity ^0.8.0;
3
4  contract DoughImplementationM1 {
5      mapping (address => bool) internal _auth;
6      address internal immutable doughIndex = address(0x0);
7
8      function isConnectors(string[] calldata) internal returns (bool) {
9          return false;
10     }
11
12     function cast(
13         string[] calldata _targetNames,
14         bytes[] calldata _datas,
15         address _origin
16     )
17     public
18     payable
19     returns (bytes32) // Dummy return to fix doughIndex buildWithCast function
20     {
21         uint256 _length = _targetNames.length;
22         require(_auth[msg.sender] || msg.sender != doughIndex, "1: permission-denied");
23         require(_length != 0, "1: length-invalid");
24         require(_length == _datas.length , "1: array-length-invalid");
25
26         string[] memory eventNames = new string[](_length);
27         bytes[] memory eventParams = new bytes[](_length);
28
29         bool isOk = isConnectors(_targetNames);
30
31         require(isOk, "1: not-connector");
32
33     }
34 }
```

Listing 26: Optimized example of Code Smell 13

```
1
2  pragma solidity ^0.8.0;
3
4  contract DoughImplementationM1 {
5      mapping (address => bool) internal _auth;
6      address internal immutable doughIndex = address(0x0);
7
8      function isConnectors(string[] calldata) internal returns (bool) {
9          return false;
10     }
11
12     function cast(
13         string[] calldata _targetNames,
14         bytes[] calldata _datas,
15         address _origin
16     )
17     public
18     payable
19     returns (bytes32) // Dummy return to fix doughIndex buildWithCast function
20     {
21         uint256 _length = _targetNames.length;
22         require(_auth[msg.sender] || msg.sender != doughIndex, "1: permission-denied");
23         require(_length != 0, "1: length-invalid");
24         require(_length == _datas.length , "1: array-length-invalid");
25
26         bool isOk = isConnectors(_targetNames);
27         require(isOk, "1: not-connector");
28
29         string[] memory eventNames = new string[](_length);
30         bytes[] memory eventParams = new bytes[](_length);
31     }
32 }
```

1. Code smell explanation. If no dependency is required, we should put the require statements as early as possible, such that upon errors, we do not make unnecessary executions of unrelated lines.

This is because the execution will revert anyways upon executing the failed require statement, then there is no need to execute unrelated lines.

2. Example. To further illustrate this code smell, we take as an example the function *cast* of the contract *DoughImplementationM1*, which is deployed at *0x17ccfEFBAa25F5C06344D133B4ce5D5F47D72b9c*. The unoptimized codes are shown in Listing 25 and the optimized version is in Listing 26. In addition, the gas consumption result is listed in Table 13. Note that in the unoptimized version, an extra set of initializations of 2 arrays are defined at lines 26 and 27. Then if the next require statement at line 31 fails, the gas paid for the initialization of the arrays will be wasted.

0.15 Code Smell 14. Avoid no-op writes to state variables.

Table 14: Gas consumption table for code smell 14.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	152253	153969	**-1.127%**	24065	23974	**0.3781%**
Execution cost	92541	94141	**-1.728%**	3001	2910	**3.0323%**

Listing 27: Unoptimized example of Code Smell 14

```
1
2  pragma solidity ^0.8.0;
3
4  contract IdleStrategy  {
5      uint256 private _lastIdleTokenPrice;
6
7      function _calculateYieldPercentage(uint256 inVal) internal returns (uint256) {
8          return inVal % 2 + inVal;
9      }
10
11     function tokenPriceWithFee(address inVal) internal returns (uint256) {
12         return 0;
13     }
14
15     function _getYieldPercentage() public returns (uint256 baseYieldPercentage) {
16         uint256 currentIdleTokenPrice = tokenPriceWithFee(address(this));
17         uint256 cachedVal = _lastIdleTokenPrice;
18
19         baseYieldPercentage = _calculateYieldPercentage(cachedVal);
20
21         _lastIdleTokenPrice = currentIdleTokenPrice;
22     }
23 }
```

Listing 28: Optimized example of Code Smell 14

```
1
2  pragma solidity ^0.8.0;
3
4  contract IdleStrategy  {
5      uint256 private _lastIdleTokenPrice;
6
7      function _calculateYieldPercentage(uint256 inVal) internal returns (uint256) {
8          return inVal % 2 + inVal;
9      }
10
11     function tokenPriceWithFee(address inVal) internal returns (uint256) {
12         return 0;
13     }
14
15     function _getYieldPercentage() public returns (uint256 baseYieldPercentage) {
16         uint256 currentIdleTokenPrice = tokenPriceWithFee(address(this));
17         uint256 cachedVal = _lastIdleTokenPrice;
18
19         baseYieldPercentage = _calculateYieldPercentage(cachedVal);
20
21         if (currentIdleTokenPrice != cachedVal) {
22             _lastIdleTokenPrice = currentIdleTokenPrice;
23         }
24     }
25 }
```

1. Code smell explanation. This code smell occurs when a variable gets rewritten by a value that is the same as the existing value. This is a no-op and is especially expensive for storage variables.
2. Example. To further illustrate this code smell, we take as an example the function *_getYieldPercentage* of the contract *IdleStrategy*, which is deployed at *0x1892038be4bd3968f4a8574593032d61c88dcacb*. The unoptimized codes are shown in Listing 27 and the optimized version is in Listing 28. In addition, the gas consumption result is listed in Table 14. In particular, the unoptimized contract unconditionally writes to a state variable, where if the new value is the same as the old one, gas would be wasted.

0.16 Code Smell 15. Reordering conditional checks for short-circuiting.

Table 15: Gas consumption table for code smell 15.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	353980	353980	**0.0%**	23078	22947	**0.5676%**
Execution cost	280518	280518	**0.0%**	1150	1019	**11.391%**

Listing 29: Unoptimized example of Code Smell 15

```
pragma solidity ^0.8.0;

contract DEVGPT {

    function owner() internal returns (address) {
        return address(0x0);
    }

    function balanceOf(address) internal returns (uint256) {
        return 100;
    }

    function _transfer(
        address from,
        address to,
        uint256 amount
    ) public {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");

        if (from != owner() && to != owner() && from != address(this) && to != 0
            x2f8fD77D037C0778E98fF160168995CD14634eaE) {
            uint256 contractTokenBalance = balanceOf(address(this));
        }
    }
}
```

Listing 30: Optimized example of Code Smell 15

```
pragma solidity ^0.8.0;

contract DEVGPT {

    function owner() internal returns (address) {
        return address(0x0);
    }

    function balanceOf(address) internal returns (uint256) {
        return 100;
    }

    function _transfer(
        address from,
        address to,
        uint256 amount
    ) public {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");

        if (to != 0x2f8fD77D037C0778E98fF160168995CD14634eaE && from != owner() && to != owner() && from !=
            address(this) ) {
            uint256 contractTokenBalance = balanceOf(address(this));
        }
    }
}
```

1. Code smell explanation. This code smell aims at utilizing the short-circuiting rule for conditional statements that are connected by "and" or "or". In particular, it suggests to restructure the conditions in a way that the scenario that have a higher chance of triggering the short-circuiting is checked first, thus avoiding unnecessary checks. Note that this code smell is based on the assumption that reordering conditions to optimize for short-circuit evaluation does not change the order in which expressions are evaluated. In other words, if any of the conditions have side effects or depend on the evaluation order, then reordering should not be carried out.

2. Example. To further illustrate this code smell, we take as an example the function *_transfer* of the contract *DEVGPT*, which is deployed at *0x2f8fD77D037C0778E98fF160168995CD14634eaE*. The unoptimized codes are shown in Listing 29 and the optimized version is in Listing 30. In addition, the gas consumption result is listed in Table 15. In this example, we make the hypothetical assumption that the "to" address is more likely to be 0x2f8fD77D037C0778E98fF160168995CD14634eaE, and thus the condition "to != 0 x2f8fD77D037C0778E98fF160168995CD14634eaE" should be placed first, since upon its failure, the other conditional statements will not be executed.

0.17 Code Smell 16. Combinable events.

Table 16: Gas consumption table for code smell 16.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	206618	178025	**13.838%**	26732	25092	**6.1349%**
Execution cost	142990	116565	**18.480%**	5668	4028	**28.934%**

Listing 31: Unoptimized example of Code Smell 16

```
1
2  pragma solidity >=0.7.6;
3
4  contract LEGO {
5      event logOnChain(string info);
6
7      function loggingOnChain() public {
8          emit logOnChain("pKeuLv");
9          emit logOnChain("IOpBBS");
10         emit logOnChain("pOmZqnWO");
11         uint256 e = 87;
12         uint256 p = (56 % 73) % 56;
13         uint256 rzzlcjlfc = 76 % 94;
14         uint256 joblgy = (77 % 86) % 41;
15         uint256 tfsleihel = 20 % 17;
16         if (
17             e == 56 * 71 &&
18             p == ((36 * 4) % 39) % 57 &&
19             rzzlcjlfc == 95 + 56 + 85 * 80 &&
20             joblgy == 20 % 27 &&
21             tfsleihel == (72 % 3) * 91
22         ) return;
23     }
24 }
```

Listing 32: Optimized example of Code Smell 16

```
1
2  pragma solidity >=0.7.6;
3
4  contract LEGO {
5      event logOnChain(string, string, string);
6
7      function loggingOnChain() public {
8          emit logOnChain("pKeuLv", "IOpBBS", "pOmZqnWO");
9          uint256 e = 87;
10         uint256 p = (56 % 73) % 56;
11         uint256 rzzlcjlfc = 76 % 94;
12         uint256 joblgy = (77 % 86) % 41;
13         uint256 tfsleihel = 20 % 17;
14         if (
15             e == 56 * 71 &&
16             p == ((36 * 4) % 39) % 57 &&
17             rzzlcjlfc == 95 + 56 + 85 * 80 &&
18             joblgy == 20 % 27 &&
19             tfsleihel == (72 % 3) * 91
20         ) return;
21     }
22 }
```

1. Code smell explanation. Consider whether all events are absolutely necessary. If they are, consider whether they can be consolidated or whether the amount of data included can be reduced.
2. Example. To further illustrate this code smell, we take as an example the function *loggingOnChain* of the contract *LEGO*, which is deployed at *0x111ACf72AA4A1fdA8500ED9f1Ba3F2374c02a21e*. The unoptimized codes are shown in Listing 31 and the optimized version is in Listing 32. In addition, the gas consumption result is listed in Table 16. In the unoptimized example, 3 separate events are continuously emitted. To analyze their gas costs, we first note that the events at lines 7, 8, and 9 each costs 1,518 gas, totaling to 4554 gas. It is worth noting that each event has an individual charge of 375 gas for an LOG1 operation. In addition, since each event is not anonymous and no arguments are indexed (i.e. no "indexed" keywords specified), there will be only one topic, which is just the hashcode of the event signature, and thus there is a 375 gas cost for each event emission. In the optimized codes, by changing the event declaration at line 4 to accept 3 string, and combining lines 7, 8, and 9 into one event emission (i.e. *emit logOnChain("pKeuLv", "IOpBBS",*

"pOmZqnWO");), we get to reduce the gas cost down to 3,054 gas, which is a saving of 1,500 gas (32.94 %). This corresponds to saving the $750 = 2 \times 375$ gas of two LOG1 operations, as well as the $750 = 2 \times 375$ gas of two topics, since they are all combined into one line now.

0.18 Code Smell 17. add constant modifier for non-changing variables.

Table 17: Gas consumption table for code smell 17.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	823085	867805	**-5.433%**	56032	50528	**9.8229%**
Execution cost	724419	768627	**-6.102%**	34092	28588	**16.144%**

Listing 33: Unoptimized example of Code Smell 17

```
pragma solidity ^0.8.0;

contract BBB {
    uint256 private _initialBuyTax=10;
    uint256 private _initialSellTax=30;
    uint256 private _finalBuyTax=2;
    uint256 private _finalSellTax=2;
    uint256 private _reduceBuyTaxAt=10;
    uint256 private _reduceSellTaxAt=20;
    uint256 private _preventSwapBefore=20;
    uint256 private _buyCount=0;

    uint8 private constant _decimals = 10;
    uint256 private constant _tTotal = 696969696969696 * 10**_decimals;
    string private constant _name = unicode"Benevolent Brainpower Brigade";
    string private constant _symbol = unicode"BBB";
    uint256 public _maxTxAmount = 20909090909090 * 10**_decimals;
    uint256 public _maxWalletSize = 20909090909090 * 10**_decimals;
    uint256 public _taxSwapThreshold= 6969696969696 * 10**_decimals;
    uint256 public _maxTaxSwap= 6969696969696 * 10**_decimals;

    address private uniswapV2Pair;
    bool private tradingOpen;
    bool private inSwap = false;
    bool private swapEnabled = false;

    event Transfer(address indexed from, address indexed to, uint256 value);


    function owner() internal returns (address) {
        return address(0x0);
    }

    function balanceOf(address) internal returns (uint256) {
        return 10;
    }


    function _transfer(address from, address to, uint256 amount) public {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");
        uint256 taxAmount=0;
        if (from != owner() && to != owner()) {
            require(amount <= _maxTxAmount, "Exceeds the _maxTxAmount.");
            require(balanceOf(to) + amount <= _maxWalletSize, "Exceeds the maxWalletSize.");
            require(amount <= _maxTaxSwap, "Exceeds the _maxTaxSwap.");

            _buyCount++;

            uint256 contractTokenBalance = balanceOf(address(this));
            if (contractTokenBalance>_taxSwapThreshold) {
                uint256 contractETHBalance = address(this).balance;
            }
        }
        emit Transfer(from, to, taxAmount);
    }
}
```

Listing 34: Optimized example of Code Smell 17

```
pragma solidity ^0.8.0;

contract BBB {
    uint256 private _initialBuyTax=10;
    uint256 private _initialSellTax=30;
    uint256 private _finalBuyTax=2;
    uint256 private _finalSellTax=2;
    uint256 private _reduceBuyTaxAt=10;
    uint256 private _reduceSellTaxAt=20;
    uint256 private _preventSwapBefore=20;
    uint256 private _buyCount=0;

```

```
14      uint8 private constant _decimals = 10;
15      uint256 private constant _tTotal = 696969696969696 * 10**_decimals;
16      string private constant _name = unicode"Benevolent Brainpower Brigade";
17      string private constant _symbol = unicode"BBB";
18      uint256 public constant _maxTxAmount = 20909090909090 * 10**_decimals;
19      uint256 public constant _maxWalletSize = 20909090909090 * 10**_decimals;
20      uint256 public constant _taxSwapThreshold= 6969696969696 * 10**_decimals;
21      uint256 public constant _maxTaxSwap= 6969696969696 * 10**_decimals;
22
23      address private uniswapV2Pair;
24      bool private tradingOpen;
25      bool private inSwap = false;
26      bool private swapEnabled = false;
27
28      event Transfer(address indexed from, address indexed to, uint256 value);
29
30
31      function owner() internal returns (address) {
32          return address(0x0);
33      }
34
35      function balanceOf(address) internal returns (uint256) {
36          return 10;
37      }
38
39
40      function _transfer(address from, address to, uint256 amount) public {
41          require(from != address(0), "ERC20: transfer from the zero address");
42          require(to != address(0), "ERC20: transfer to the zero address");
43          require(amount > 0, "Transfer amount must be greater than zero");
44          uint256 taxAmount=0;
45          if (from != owner() && to != owner()) {
46              require(amount <= _maxTxAmount, "Exceeds the _maxTxAmount.");
47              require(balanceOf(to) + amount <= _maxWalletSize, "Exceeds the maxWalletSize.");
48              require(amount <= _maxTaxSwap, "Exceeds the _maxTaxSwap.");
49
50              _buyCount++;
51
52              uint256 contractTokenBalance = balanceOf(address(this));
53              if (contractTokenBalance>_taxSwapThreshold) {
54                  uint256 contractETHBalance = address(this).balance;
55              }
56          }
57          emit Transfer(from, to, taxAmount);
58      }
59
60  }
```

1. Code smell explanation. Solidity replaces all constant state variables by their value during compilation. This means they will not be placed in storage and thus operations on them are much cheaper than that on storage variables. As a result, if we are certain that a state variable will not change at all. then we should add a constant modifier to it.

2. Example. To further illustrate this code smell, we take as an example the function *_transfer* of the contract *BBB*, which is deployed at *0x340de5cb9b177ff1e3d00e6aa3082f979fca621e*. The unoptimized codes are shown in Listing 33 and the optimized version is in Listing 34. In addition, the gas consumption result is listed in Table 17. In particular, based on the assumption that the variables _maxTxAmount, _maxWalletSize, _taxSwapThreshold, and _maxTaxSwap are non-changing, we add a constant modifier to them in this example.

0.19 Code Smell 18. Function visibility.

Table 18: Gas consumption table for code smell 18.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	418790	418790	**0.0%**	47316	27416	**42.057%**
Execution cost	339980	339980	**0.0%**	25712	5812	**77.395%**

Listing 35: Unoptimized example of Code Smell 18

```
1
2  pragma solidity ^0.8.0;
3
4  contract AkshunSeasonPassNft  {
5
6      string contractURI;
7      event ContractURIUpdated(string);
8      error ParamInvalid(uint8 paramPosIdx);
9
10 function updateContractURI(string memory _contractURI)
11         public
12
13     {
14         // Validate input params.
15
16         if (bytes(_contractURI).length == 0) revert ParamInvalid(0);
17
18         // Update/set state vars.
19
20         contractURI = _contractURI;
21
22         // Emit events.
23
24         emit ContractURIUpdated(_contractURI);
25     }
26
27 }
```

Listing 36: Optimized example of Code Smell 18

```
1
2  pragma solidity ^0.8.0;
3
4  contract AkshunSeasonPassNft  {
5
6      string contractURI;
7      event ContractURIUpdated(string);
8      error ParamInvalid(uint8 paramPosIdx);
9
10 function updateContractURI(string memory _contractURI)
11         external
12
13     {
14         // Validate input params.
15
16         if (bytes(_contractURI).length == 0) revert ParamInvalid(0);
17
18         // Update/set state vars.
19
20         contractURI = _contractURI;
21
22         // Emit events.
23
24         emit ContractURIUpdated(_contractURI);
25     }
26
27 }
```

1. Code smell explanation. This code smell notes the fact that functions of different visibility (i.e. public v.s. external) consume different amounts of gas, where public functions cost more gas. This is because for public functions, the input parameters are copied into memory automatically, which costs gas, while external functions could directly read from calldata. Therefore, if we are certain that the function will not be called internally, we should declare it as external instead of public.
2. Example. To further illustrate this code smell, we take as an example the function *updateContractURI* of the contract *AkshunSeasonPassNft*, which is deployed at *0x7e9F2D2583FEF83aF0dDA74E457B6320228B20dE*. The unoptimized codes are shown in Listing 35 and the optimized version is in Listing 36. In addition, the gas consumption result is listed in Table 18. In this example, we make the assumption

that the "updateContractURI" function will not be called internally, and change its visibility to external. This contributes to a huge amount of saved gas.

0.20 Code Smell 19. Dead codes.

Table 19: Gas consumption table for code smell 19.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	158393	149733	**5.4674%**	22351	22073	**1.2437%**
Execution cost	97947	89941	**8.1738%**	1147	869	**24.237%**

Listing 37: Unoptimized example of Code Smell 19

```
1
2 pragma solidity ^0.8.0;
3
4 contract ExampleContract {
5     function exampleFunction(uint x) public returns (uint256) {
6         if ( x > 5) {
7             if ( x*x < 20) {
8                 return 5 * x;
9             }
10            else {
11                return 4 * x;
12            }
13        }
14        else {
15            return x;
16        }
17    }
18
19 }
```

Listing 38: Optimized example of Code Smell 19

```
1
2 pragma solidity ^0.8.0;
3
4 contract ExampleContract {
5     function exampleFunction(uint x) public returns (uint256) {
6         if ( x > 5) {
7                 return 4 * x;
8         }
9         else {
10            return x;
11        }
12    }
13
14 }
```

1. Code smell explanation. Any code that is not used or cannot be reached during execution is wasteful and consumes unnecessary gas. Therefore, they should be removed.

2. Example. To further illustrate this code smell, we take the example from an existing paper [2]. The unoptimized codes are shown in Listing 37 and the optimized version is in Listing 38. In addition, the gas consumption result is listed in Table 19. It can be observed from the codes that once we enter the branch where "x > 5," it is never possible for "x*x < 20" to be true. Therefore, we could remove the corresponding branch as these are dead codes.

0.21 Code Smell 20. Using revert instead of require for error handling.

Table 20: Gas consumption table for code smell 20.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	261062	210431	**19.394%**	22312	22024	**1.2907%**
Execution cost	197220	150171	**23.856%**	980	692	**29.387%**

Listing 39: Unoptimized example of Code Smell 20

```
1
2  pragma solidity ^0.8.0;
3
4  contract DOLLARAI {
5      uint256 private _redisFeeOnBuy = 0;
6      uint256 private _taxFeeOnBuy = 30;
7      uint256 private _redisFeeOnSell = 0;
8      uint256 private _taxFeeOnSell = 45;
9
10     function setFee(uint256 taxFeeOnBuy, uint256 taxFeeOnSell) public {
11         require(taxFeeOnBuy >= 0 && taxFeeOnBuy <= 99, "Buy tax must be between 0% and 99%");
12         require(taxFeeOnSell >= 0 && taxFeeOnSell <= 99, "Sell tax must be between 0% and 99%");
13
14         _taxFeeOnBuy = taxFeeOnBuy;
15         _taxFeeOnSell = taxFeeOnSell;
16     }
17 }
```

Listing 40: Optimized example of Code Smell 20

```
1
2  pragma solidity ^0.8.0;
3
4  contract DOLLARAI {
5      uint256 private _redisFeeOnBuy = 0;
6      uint256 private _taxFeeOnBuy = 30;
7      uint256 private _redisFeeOnSell = 0;
8      uint256 private _taxFeeOnSell = 45;
9
10     error InvalidRangeOfInput();
11
12     function setFee(uint256 taxFeeOnBuy, uint256 taxFeeOnSell) public {
13         if (taxFeeOnBuy < 0) revert InvalidRangeOfInput();
14         if (taxFeeOnBuy > 99) revert InvalidRangeOfInput();
15         if (taxFeeOnSell < 0) revert InvalidRangeOfInput();
16         if (taxFeeOnSell > 99) revert InvalidRangeOfInput();
17
18         _taxFeeOnBuy = taxFeeOnBuy;
19         _taxFeeOnSell = taxFeeOnSell;
20     }
21 }
```

1. Code smell explanation. As a new feture from Solidity 0.8.4, "revert" costs less gas for both deployment and running when applied on custom errors. This means that if possible, we should use "revert" with custom errors instead of "require".

2. Example. To further illustrate this code smell, we take as an example the function *setFee* of the contract *DOLLARAI*, which is deployed at *0xfC31f0457DaB6A52432a033f13111981f464b74a*. The unoptimized codes are shown in Listing 39 and the optimized version is in Listing 40. In addition, the gas consumption result is listed in Table 20. In particular, the gas is saved because we have replace the "require" statements with "revert" with custom errors. Since "require" statements internally uses "revert" statements upon failure.

0.22 Code Smell 21. Minimization of event message string.

Table 21: Gas consumption table for code smell 21.

Type of cost	$\mathcal{D}_u$	$\mathcal{D}_o$	$\mathcal{S}_{\mathcal{D}}$	$\mathcal{R}_u$	$\mathcal{R}_o$	$\mathcal{S}_{\mathcal{R}}$
Transaction cost	178025	137837	**22.574%**	25092	23152	**7.7315%**
Execution cost	116565	78529	**32.630%**	4028	2088	**48.162%**

Listing 41: Unoptimized example of Code Smell 21

```

pragma solidity >=0.7.6;

contract LEGO {
    event logOnChain(string, string,string);

    function loggingOnChain() public {
      emit logOnChain("pKeuLv", "IOpBBS", "pOmZqnWO");
      uint256 e = 87;
      uint256 p = 56 % 73 % 56;
      uint256 rzzlcjlfc = 76 % 94;
      uint256 joblgy = 77 % 86 % 41;
      uint256 tfsleihel = 20 % 17;
      if (e == 56 * 71 && p == 36 * 4 % 39 % 57 && rzzlcjlfc == 95 + 56 + 85 * 80 && joblgy == 20 % 27 && tfsleihel == 72 % 3 * 91) return;
    }
}
```

Listing 42: Optimized example of Code Smell 21

```

pragma solidity >=0.7.6;

contract LEGO {
    event logOnChain(string);

    function loggingOnChain() public {
      emit logOnChain("pKeuLv & IOpBBS & pOmZqnWO");
      uint256 e = 87;
      uint256 p = 56 % 73 % 56;
      uint256 rzzlcjlfc = 76 % 94;
      uint256 joblgy = 77 % 86 % 41;
      uint256 tfsleihel = 20 % 17;
      if (e == 56 * 71 && p == 36 * 4 % 39 % 57 && rzzlcjlfc == 95 + 56 + 85 * 80 && joblgy == 20 % 27 && tfsleihel == 72 % 3 * 91) return;
    }
}
```

1. Code smell explanation. The size of input data for the emission of events costs gas, where each byte incurs a cost of 8 gas. Therefore, we should be careful with this and consider minimizing the amount of data to include in a event message.
2. Example. To further illustrate this code smell, we take as an example the function *loggingOnChain* of the contract *LEGO*, which is deployed at *0x111ACf72AA4A1fdA8500ED9f1Ba3F2374c02a21e*. The unoptimized codes are shown in Listing 41 and the optimized version is in Listing 42. In addition, the gas consumption result is listed in Table 21. In particular, the gas is saved because we have reduced the input data size from 3 strings into only one. To further clarify the source of gas saving, for the optimized version, its event data is: "0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000001a704b65754c76202620494f70424253202620704f6d5a716e574f000000000000". On the contrary, the unoptimized version has: "0x000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000000000000000000000000000000000000a000000000000000000000000000000000000000000000000000000000000000e00000000000000000000000000000000000000000000000000000000000000006704b65754c7600000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000006494f704242530000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000008704f6d5a716e574f000000000000000000000000000000000000000000000000", It can be observed that the unoptimized version

adds zero paddings for each of the event strings, so if there are 3 separate strings, they each account for extra zero paddings and the logged data eventually take up more storage space.

0.23 Code Smell 22. Replacing MUL/DIV of powers of 2 by SHL/SHR.

Table 22: Gas consumption table for code smell 22.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	172617	144589	**16.237%**	21847	21604	**1.1122%**
Execution cost	111759	85335	**23.643%**	783	540	**31.034%**

Listing 43: Unoptimized example of Code Smell 22

```
1
2 pragma solidity ^0.8.0;
3
4 contract PancakeChainlinkOracle {
5     uint256 public constant Q96 = 2 ** 96;
6     uint256 public constant s = 2 ** 96;
7
8     function latestAnswer() public returns (uint) {
9         uint256 priceX96 = 1;
10        return Q96 * s / priceX96;
11    }
12 }
```

Listing 44: Optimized example of Code Smell 22

```
1
2 pragma solidity ^0.8.0;
3
4 contract PancakeChainlinkOracle {
5     uint256 public constant Q96 = 2 ** 96;
6     uint256 public constant s = 96;
7
8     function latestAnswer() public returns (uint) {
9         uint256 priceX96 = 1;
10        return (Q96 << s) / priceX96;
11    }
12 }
```

1. Code smell explanation. This code smell saves gas by specifying that a MUL or DIV opcode costs 5 gas, while the corresponding SHL and SHR only costs 3 gas. This means when multiplying or dividing by powers of 2, it is more gas-efficient to instead use bit shifts.
2. Example. To further illustrate this code smell, we take as an example the function *latestAnswer* of the contract *PancakeChainlinkOracle*, which is deployed at *0x708d6c06df93fafd08f64f20564cebcc70dee12e*. The unoptimized codes are shown in Listing 43 and the optimized version is in Listing 44. In addition, the gas consumption result is listed in Table 22. In particular, the gas saving in this example not only came from setting MUL to SHL, but also in that when MUL is used, more preparatory work in the form of a large chunk of bytecodes is observed.

0.24 Code Smell 23. Struct variable reordering.

Table 23: Gas consumption table for code smell 23.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	276923	276959	**-0.013%**	91601	89701	**2.0742%**
Execution cost	207451	207451	**0.0%**	70537	68637	**2.6936%**

Listing 45: Unoptimized example of Code Smell 23

```
1
2  pragma solidity ^0.8.0;
3
4  contract ExampleContract {
5
6      struct Struct {
7          uint mem1 ;
8          address mem2 ;
9          uint mem3 ;
10         bool mem4 ;
11     }
12
13     Struct public data ;
14
15     function setData () internal {
16         data = Struct (1 , address (0) , 1 , true ) ;
17     }
18
19     function getData () internal returns (Struct memory) {
20         return data ;
21     }
22
23     function exampleFunction() public returns  (Struct memory) {
24         setData();
25         return getData();
26     }
27 }
```

Listing 46: Optimized example of Code Smell 23

```
1
2  pragma solidity ^0.8.0;
3
4  contract ExampleContract {
5
6      struct Struct {
7          uint mem1 ;
8          address mem2 ;
9          bool mem4 ;
10         uint mem3 ;
11     }
12
13     Struct public data ;
14
15     function setData () internal {
16         data = Struct (1 , address (0) , true , 1 ) ;
17     }
18
19     function getData () internal returns (Struct memory) {
20         return data ;
21     }
22
23     function exampleFunction() public returns  (Struct memory) {
24         setData();
25         return getData();
26     }
27 }
```

1. Code smell explanation. By reordering the variables inside a struct, we could arrange them arranged to form a more compact layout such that they occupy less storage space and thus both storing and accesses to them would save gas.

2. Example. To further illustrate this code smell, we take the example from an existing paper [9]. The unoptimized codes are shown in Listing 45 and the optimized version is in Listing 46. In addition, the gas consumption result is listed in Table 23. In particular, in the unoptimized contract, 4 storage slots are taken, with each member variable taking one slot. On the other hand, in the optimized version, by exchanging the order of mem3 and mem4, the variables mem2 and mem4 get

packed into one storage slot since mem2 only takes 20 bytes and mem4 only 1 byte. This largely saves storage space and thus the gas required to access the state variables.

0.25 Code Smell 24. Loop invariant codes.

Table 24: Gas consumption table for code smell 24.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	215033	216353	**-0.613%**	101534	63135	**37.818%**
Execution cost	153571	154771	**-0.781%**	80330	41931	**47.801%**

Listing 47: Unoptimized example of Code Smell 24

```
1
2 pragma solidity ^0.8.0;
3
4 contract ExampleContract  {
5     uint x = 1;
6     uint y = 2;
7
8     function exampleFunction(uint k) public returns (uint256 amountOut) {
9         uint sum = 0;
10        for ( uint i = 1 ; i <= k ; i++) {
11            sum = sum + x + y;
12        }
13        return sum;
14    }
15 }
```

Listing 48: Optimized example of Code Smell 24

```
1
2 pragma solidity ^0.8.0;
3
4 contract ExampleContract  {
5     uint x = 1;
6     uint y = 2;
7
8     function exampleFunction(uint k) public returns (uint256 amountOut) {
9         uint sum = 0;
10        uint s = x + y;
11        for ( uint i = 1 ; i <= k ; i++) {
12            sum = sum + s;
13        }
14        return sum;
15    }
16 }
```

1. Code smell explanation. This code smell occurs when some lines of codes, which would produce the same output upon each execution, is carried out in each iteration of a loop. This wastes gas because these lines of codes could be moved outside the loop to be executed by just once.

2. Example. To further illustrate this code smell, we take the example from an existing paper [2]. The unoptimized codes are shown in Listing 47 and the optimized version is in Listing 48. In addition, the gas consumption result is listed in Table 24. In particular, the operation "x + y" would produce the same result every time the loop executes, which wastes gas. Instead, in the optimized version, we move the computation ourside the loop and store the result in an intermediate variable "s" and thus avoid the repetitive computations.

0.26 Code Smell 25. Avoid expensive operations inside loops.

Table 25: Gas consumption table for code smell 25.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	170799	171435	**-0.372%**	102621	80848	**21.216%**
Execution cost	109559	110159	**-0.547%**	81417	59644	**26.742%**

Listing 49: Unoptimized example of Code Smell 25

```
1
2 pragma solidity ^0.8.0;
3
4 contract ExampleContract  {
5     uint sum = 0;
6     function exampleFunction(uint x) public returns (uint256 amountOut) {
7         for ( uint i = 1 ; i <= x ; i++) {
8             sum += i;
9         }
10        return sum;
11    }
12 }
```

Listing 50: Optimized example of Code Smell 25

```
1
2 pragma solidity ^0.8.0;
3
4 contract ExampleContract  {
5     uint sum = 0;
6     function exampleFunction(uint x) public returns (uint256 amountOut) {
7         uint tmp = sum;
8         for ( uint i = 1 ; i <= x ; i++) {
9             tmp += i;
10        }
11        sum = tmp;
12        return tmp;
13    }
14 }
```

1. Code smell explanation. It is recommended to avoid performing gas-expensive operations inside a loop (e.g. accessing storage variables, emitting events), and if possible, restructure the codes to move them out of the loop.

2. Example. To further illustrate this code smell, we take the example from an existing paper [2]. The unoptimized codes are shown in Listing 49 and the optimized version is in Listing 50. In addition, the gas consumption result is listed in Table 25. In the optimized version, a new memory variable is introduced (i.e. tmp) to hold the intermediate computation results, and finally assigned back to sum. This largely saves gas because in the unoptimized version, during each iteration of the loop, an update to the state variable "sum" is performed, which wastes gas since repeated writes to state variables is very expensive.

0.27 Code Smell 26. Using bytes32 for string representation.

Table 26: Gas consumption table for code smell 26.

Type of cost	$\mathscr{D}_u$	$\mathscr{D}_o$	$\mathscr{S}_{\mathscr{D}}$	$\mathscr{R}_u$	$\mathscr{R}_o$	$\mathscr{S}_{\mathscr{R}}$
Transaction cost	535430	407026	**23.981%**	33697	29939	**11.152%**
Execution cost	449628	330316	**26.535%**	11341	8191	**27.775%**

Listing 51: Unoptimized example of Code Smell 26

```
1
2  pragma solidity ^0.8.0;
3
4  contract VoteForLaunch {
5      uint32 public MAX_VOTING_DAYS = 10 * 24 * 3600;
6      mapping(string => bool) public reservedTicks;     // check if tick is occupied
7      event NewApplication(string tick, address applicant, uint40 expireAt, string cid, uint128 deposit);
8
9      function newVote(string memory _tick, uint40 _expireSeconds, string memory _cid) public {
10         require(_expireSeconds <= MAX_VOTING_DAYS, "more than max days to vote");
11         require(!reservedTicks[_tick], "reserved ticks can not apply");
12
13         emit NewApplication(_tick, msg.sender, uint40(block.timestamp + _expireSeconds), _cid, 10);
14     }
15 }
```

Listing 52: Optimized example of Code Smell 26

```
1
2  pragma solidity ^0.8.0;
3
4  contract VoteForLaunch {
5      uint32 public MAX_VOTING_DAYS = 10 * 24 * 3600;
6      mapping(bytes32 => bool) public reservedTicks;     // check if tick is occupied
7      event NewApplication(bytes32 tick, address applicant, uint40 expireAt, bytes32 cid, uint128 deposit);
8
9      function newVote(bytes32 _tick, uint40 _expireSeconds, bytes32 _cid) public {
10         require(_expireSeconds <= MAX_VOTING_DAYS, "more than max days to vote");
11         require(!reservedTicks[_tick], "reserved ticks can not apply");
12
13         emit NewApplication(_tick, msg.sender, uint40(block.timestamp + _expireSeconds), _cid, 10);
14     }
15 }
```

1. Code smell explanation. In Solidity, bytes32 is a more efficient representation for string literals than the string type. In particular, if we are certain that the length of a string will not exceed 32 bytes, then we should use it as bytes32.

2. Example. To further illustrate this code smell, we take as an example the function *newVote* of the contract *VoteForLaunch*, which is deployed at *0xb9250f2dc0706f172F3565c11fcF9f7CFB2F27A7*. The unoptimized codes are shown in Listing 51 and the optimized version is in Listing 52. In addition, the gas consumption result is listed in Table 26. In this example, the string input types are changed into bytes32, which utilizes the better memory layout of the bytes32 variable types and thus saves gas. Note that this is based on the premise that the input size would not exceed 32 bytes, and the "string" type is still needed for inputs with arbitrary length that could be longer than 32 bytes.

Bibliography

[18a] *BSON Types*. https://www.mongodb.com/docs/v4.4/reference/bson-types/. Accessed: 2023-08-18 (cited on page 15).

[Che+17] Ting Chen et al. "Under-optimized smart contracts devour your money". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pages 442–446. DOI: 10.1109/SANER.2017.7884650 (cited on pages 5, 42, 49, 50).

[Che+22] Yanju Chen et al. "Synthesis-Powered Optimization of Smart Contracts via Data Type Refactoring". In: 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563308. URL: https://doi.org/10.1145/3563308 (cited on page 5).

[Di +22] Andrea Di Sorbo et al. "Profiling Gas Consumption in Solidity Smart Contracts". In: *J. Syst. Softw.* 186.C (Apr. 2022). ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111193. URL: https://doi.org/10.1016/j.jss.2021.111193 (cited on page 5).

[KTT22] Kawaldeep Kaur, Shubham Tomar, and Meenakshi Tripathi. "Gas Fee Reduction by Detecting Loop Fusible Patterns in Ethereum Smart Contract". In: Dec. 2022, pages 458–463. DOI: 10.1109/ANTS56424.2022.10227770 (cited on page 5).

[Kon+22] Que-Ping Kong et al. "Characterizing and Detecting Gas-Inefficient Patterns in Smart Contracts". In: *Patterns in Smart Contracts. J. Comput. Sci. Technol.* (2022) (cited on page 5).

[Mar+20] Lodovica Marchesi et al. "Design Patterns for Gas Optimization in Ethereum". In: *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 2020, pages 9–15. DOI: 10.1109/IWBOSE50093.2020.9050163 (cited on page 5).

[Nel+21] Keerthi Nelaturu et al. "Smart Contracts Refinement for Gas Optimization". In: *2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*. 2021, pages 229–236. DOI: 10.1109/BRAINS52497.2021.9569819 (cited on page 5).

[Ngu+22] Quang-Thang Nguyen et al. “GasSaver: A Tool for Solidity Smart Contract Optimization”. In: BSCI ’22. Nagasaki, Japan: Association for Computing Machinery, 2022, pages 125–134. ISBN: 9781450391757. DOI: 10.1145/3494106.3528683. URL: https://doi.org/10.1145/3494106.3528683 (cited on pages 5, 47).

[PLI17] JaeYong Park, Daegeon Lee, and Hoh In. “Saving Deployment Costs of Smart Contracts by Eliminating Gas-wasteful Patterns”. In: *International Journal of Grid and Distributed Computing* 10 (Dec. 2017), pages 53–64. DOI: 10.14257/ijgdc.2017.10.12.06 (cited on page 5).

[18b] *UNIX_TIMESTAMP*. https://mariadb.com/kb/en/unix_timestamp/. Accessed: 2023-08-18 (cited on page 15).

[Zha+23] Ziyi Zhao et al. “GaSaver: A Static Analysis Tool for Saving Gas”. In: *IEEE Transactions on Sustainable Computing* 8.2 (2023), pages 257–267. DOI: 10.1109/TSUSC.2022.3221444 (cited on page 5).