SolSAT: Enhancing Gas Optimization in Solidity Smart Contracts through Static Analysis and Programming Language Insights

Hesam Sarkhosh
hsarkhos@uwaterloo.ca

University of Waterloo
CS 842: Advanced Topics in Programming Language Design and Implementation
Final Project
Instructor: Professor Stephen Watt
December 2024

1. Abstract

Ethereum introduced smart contracts (SCs) in 2015 [1], revolutionizing blockchain by allowing decentralized applications. However, programming in Solidity, Ethereum's language, is distinct from traditional software development. Gas costs in Ethereum directly translate to monetary costs. Misestimating gas or inefficiencies can lead to failed transactions or unnecessary expenses [2]. Gas optimization is a critical concern in Ethereum smart contract development due to its direct impact on the cost-efficiency and performance of decentralized applications. This project investigates the influence of programming language constructs and runtime behaviours on gas consumption in Solidity, the primary language for Ethereum contracts. We examine key areas such as memory and storage management, control flow, arithmetic operations, variable definitions, and function interactions. Additionally, we have developed a prototype static analysis tool to automate the detection of these inefficiencies. This tool identifies and suggests improvements for ten common gas-inefficient patterns in Solidity, including: struct and state variable packing, boolean packing, efficient integer usage, string optimization, fixed-size array usage, unnecessary zero initialization, memory vs. calldata optimization, storage release, and function visibility optimization [2][3]. By applying these optimizations, developers can significantly reduce gas costs and improve the overall efficiency of their smart contracts.

2. Introduction

To establish the context of this project, the following background information is provided:

2.1 Background

Blockchain

Imagine a group of friends — Alice, Bob, Charlie, and Dave — who all love playing chess. They often compete against each other and want to keep a record of their matches to figure out who among them is the best player. Initially, they think that one of them, say Alice, could simply record the results in her notebook. However, this raises some concerns: What if Alice makes a mistake in recording, or if her notebook is lost, damaged, or even intentionally altered to favour her scores? Relying on a single point of failure to maintain the records could lead to disputes, loss of trust, and data accuracy issues.

To address this, the group decides on a different approach. Instead of just Alice recording the results, each player maintains their own personal notebook where they independently write down the outcome of every match they play. In this way, Alice, Bob, Charlie, and Dave each keep a full record of all the games. This setup creates a distributed system where everyone has a replica of the data. If any one of them were to lose their notebook or accidentally make an error, they could refer to the others' records to cross-check and ensure consistency. This replicated notebook works as a *Decentralized Ledger*, which its purpose is to record a set of transactions that is maintained and updated by multiple independent nodes.

However, this setup leads to a new challenge. Since each friend has their own version of the records, they need a way to ensure that all their notebooks stay in sync. For example, if Alice and Bob both record the outcome of a game, but Bob forgets to log a later match between Charlie and Dave, there will be a discrepancy in their records. They need a consistent and reliable way to agree on the results of each match and ensure that everyone’s notebook remains accurate and up to date. This agreement process is known as *Consensus*.

Bitcoin introduced in 2005, was the first fully implemented version of a decentralized ledger, utilizing a chain of transaction blocks (i.e., blockchain) to record state changes.

Smart Contracts

The term “Smart Contract” was first used by *Nick Szabo* in 1997 to generally describe the automation of legal contracts [4]. The term has been, and continues to be, used to denote legal agreements that can be fully or partially coded and enforced via software. In modern usage, the term typically refers to scripts that are executed simultaneously on various nodes of a distributed ledger, such as blockchain. Operating on a blockchain and not having a single point of failure

enables a smart contract to be securely executed by a network of mutually untrusted nodes, and enforce the terms of an agreement without the need for an external trusted authority.

Ethereum

In 2014, Ethereum was introduced with a novel perspective: instead of solely using the decentralized ledger as a state management machine to record changes in a linear manner, it could also be leveraged to run custom scripts and manage states using the same decentralized features [1]. This was the very first implementation of a Smart Contract execution platform.

Ethereum introduced its own domain specific language called "*Solidity*" which was tailored for developing smart contracts on top of Ethereum. Solidity is based on 256bit words and is a Turing-complete language [2], meaning that the language is capable of expressing any computation that can be performed by a computer, given sufficient time and memory (and also gas, but we'll elaborate this later on). These contracts are executed within the Ethereum Virtual Machine (EVM), providing a secure runtime environment.

Ethereum also provides developers an online IDE for solidity called Remix (remix.ethereum.org) which enables developing as well as deploying smart contracts. On Figure 1, you can see the environment of Remix as well as the syntax of the Solidity language.

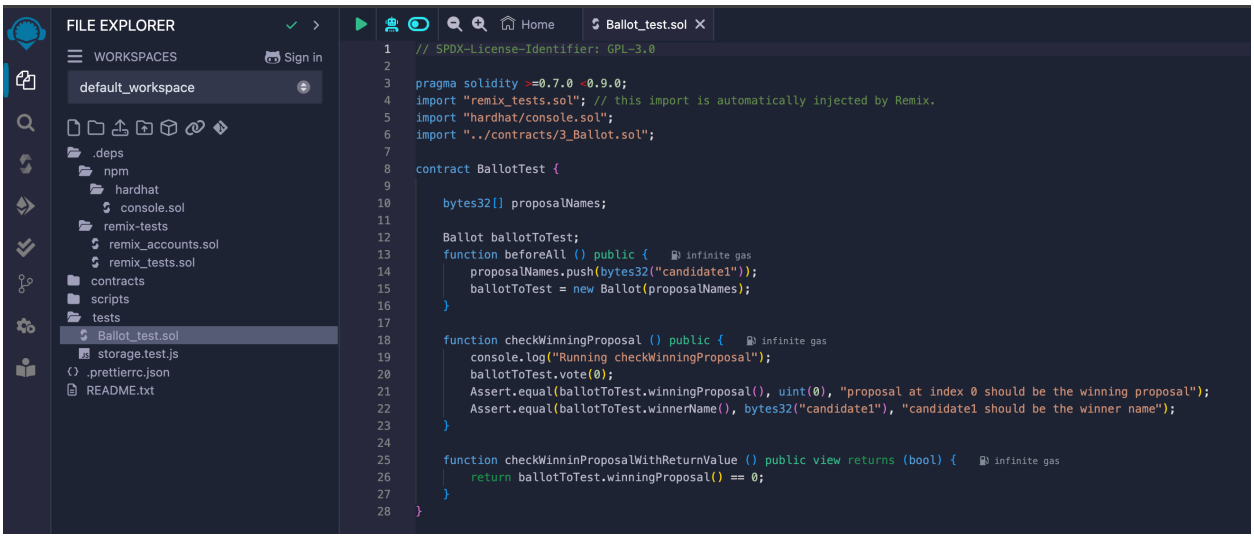


Figure 1 - Remix IDE Environment and Solidity syntax

In Solidity, when a SC is compiled, it is converted into a sequence of "operation codes", also known as opcodes. These are identified by abbreviations, for example ADD for addition, MUL for multiplication, etc. All the opcodes and their description are available in the so-called Ethereum yellow paper [5], the document which first described this system. Each opcode has a predetermined amount of gas assigned to it, which is a measure of the computational effort

required to perform that particular operation. Bytecodes are similar to opcodes but are represented by hexadecimal numbers. The EVM executes bytecodes.

Executing Smart Contracts comes at a cost. Gas serves as a fee for performing computations on the network [2]. In Ethereum, this fee is paid in Ether, the native cryptocurrency and official gas token of the Ethereum network. Before initiating a transaction, users must specify a gas limit, which sets the maximum amount of gas they are willing to pay. Setting the gas limit too low can result in a failed transaction, while setting it too high risks incurring unnecessary costs, especially in the presence of bugs.

Figure 2 shows the amount of fee (gas) due for the various operations in Ethereum. By default, the minimum amount of gas for an operation that affects the state of the EVM, is 21000 gas. For example, this is the amount required to send Ethers from one account to another. To execute a

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
JUMP	8	Unconditional Jump
SSTORE	5,000/20,000	Storage operation
BALANCE	400	Get balance of an account
CALL	25,000	Create a new account
CREATE	32,000	Create a new account

Figure 2 - Gas due for the various operations in Ethereum

function of a SC this amount will be 21000 gas plus the gas needed to perform each of the required opcodes. An exception to this behaviour is when the called function is read-only and simple. Such a function is called a *view function*, and its execution is free and immediate, because it does not change the state of the EVM. So, calling a view function within a Call in a local node does not cost gas, while calling the same function from a deployed SC within a Transaction costs gas.

As you can see in the figure 2, SSTORE command or calling storage operation to manipulate the storage is very expensive. To understand what that means we should take a look at different types of memory and storage used in Ethereum smart contracts [2]:

- Stack:
  - Holds small local variables.
  - Nearly free to use but has limited capacity.
  - Accessed with instructions like `PUSH`, `POP`, `COPY`, and `SWAP`.
- Memory:
  - Temporary storage for values generated during execution.
  - Reset at the start of every function call.
  - Accessed with `MLOAD`, `MSTORE`, and `MSTORE8`.

- Storage:
  - Persistent storage for smart contract state variables.
  - Unique to each contract, allowing access only to its own storage.
  - Accessed with `SLOAD` and `STORE`.
- Calldata:
  - Read-only data location for external function call parameters.
- Event Log:
  - Stores data from events raised by smart contracts.
  - Accessible only by external applications, not by smart contracts.

2.2 Motivation

The amount of gas required for each operation in a smart contract is closely tied to the quality of the contract's code. Two notable studies, by Marchesi et al. [2] and Khanzadeh et al. [3], demonstrate that adhering to specific design patterns can significantly optimize gas usage in Ethereum smart contracts. These optimizations are not merely helpful but crucial, as they directly impact the cost-effectiveness, scalability, and usability of decentralized applications. Efficient gas usage reduces transaction costs, making blockchain solutions more accessible and attractive to users, while also alleviating network congestion by minimizing unnecessary computational overhead. By following these design principles, developers can enhance the performance of their smart contracts and contribute to the broader adoption of Ethereum-based applications.

However, not all developers have access to such research or are familiar with these design patterns. This knowledge gap can lead to inefficient smart contract designs, resulting in higher costs and reduced performance. Therefore, the aim of this project is to develop a static analysis tool that can automatically identify gas-inefficient patterns in Solidity code and provide actionable recommendations to developers. By integrating these insights into their workflow, developers can make informed changes to their code, improving gas efficiency without requiring in-depth knowledge of the underlying optimization techniques. This tool would not only lower the barrier to entry for optimizing smart contracts but also promote the adoption of best practices across the Ethereum ecosystem, fostering a more efficient and sustainable blockchain environment.

2.3 Objectives

The objectives of this project are threefold:

Identify General Factors Affecting Gas Price:
The first objective is to analyze and understand the broader factors influencing gas prices on the

Ethereum network. These include intrinsic factors, such as the computational complexity of operations and storage usage, as well as extrinsic factors like network congestion and transaction demand. By understanding these influences, the project aims to establish a foundational framework for identifying opportunities to optimize smart contract execution.

Determine Code Patterns Impacting Gas Consumption:
The second objective is to pinpoint specific code patterns in Solidity that contribute to excessive gas consumption. This entails examining common practices and design choices in smart contract development that result in inefficient resource usage. By identifying these patterns, the project seeks to provide developers with actionable insights and practical guidelines for optimizing their contracts, ultimately reducing costs and improving performance.

Develop a Static Analysis Tool for Gas Optimization:
The third objective is to design and implement a static analysis tool that can automatically detect gas-inefficient patterns in Solidity smart contracts. This tool will analyze the code, identify inefficiencies, and provide actionable recommendations for optimization. By integrating this tool into developers' workflows, it will democratize access to gas optimization strategies, lower the barrier to entry for creating efficient contracts, and promote widespread adoption of best practices within the Ethereum ecosystem.

3. Analysis

Gas serves as a deterrent against misuse and attacks, ensuring that only economically justified operations are performed. Efficient gas usage reduces transaction costs, encourages adoption of decentralized applications, and mitigates environmental concerns associated with resource-intensive operations. In this section we aim to provide a analysis of the factors affecting Gas Price so we can use it in developing the static analysis tool.

3.1 General Factors affecting Gas Price

The first objective is to analyze and understand the various factors influencing gas prices on the Ethereum network. This understanding encompasses both the technical and practical dimensions of gas consumption:

Technical Factors

(1) **Operation Costs:** Each Ethereum Virtual Machine (EVM) opcode has a predetermined gas cost. For example, simple arithmetic operations like addition cost 3 gas units, while storage-

related operations such as SSTORE may cost up to 20,000 gas units. This highlights the disproportionately high cost of storage compared to computation or memory. (refer to figure 1)

(2) **Storage Types:** Different types of memory (e.g., stack, memory, storage) have varying gas costs. Persistent storage is the most expensive, making efficient memory management a critical aspect of smart contract development.

(3) **Transaction Basics:** Sending Ether requires a minimum of 21,000 gas, and additional complexity in contract interactions increases this baseline cost.

Network Factors

(4) **Gas Price Dynamics:** The "gas price," set by users, determines the cost per unit of gas and influences transaction prioritization by miners. Higher gas prices lead to faster transaction processing, creating a competitive market influenced by network demand.

(5) **Network Congestion:** High transaction volumes during peak usage lead to increased gas prices, amplifying the cost of complex or large-scale operations.

3.2 Design Patterns

Here, I aim to list the code design patterns that, when followed, can enhance gas optimization. I am merging two separate lists to form a unified list: the first is derived from Marchesi et al. [2], and the second from Khanzadeh et al. [3].

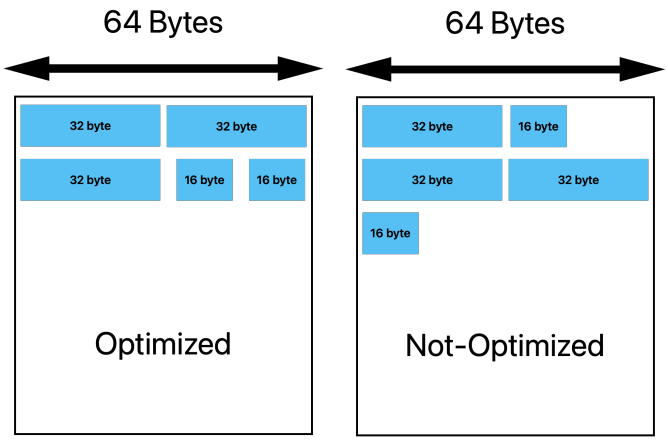
Storage Optimization Patterns

1. Struct Packing (SP)

Rearrange struct fields to minimize memory slots used.

```
// Before: Inefficient struct
struct Person {
    uint256 id;     // 32 bytes
    bool active;    // 1 byte
    uint256 age;    // 32 bytes
}

// After: Optimized struct
struct Person {
    uint256 id;     // 32 bytes
    uint256 age;    // 32 bytes
    bool active;    // 1 byte
}
```



2. State Variable Packing (SVP)

Rearrange state fields to minimize memory slots used. Same thing as number 1 but instead of struct, the state variables (contract global variables).

3. Boolean Packing

Store multiple booleans in a uint256.

```
uint256 packedBooleans;

function setBoolean(uint index, bool value) public {
    if (value) {
        packedBooleans |= (1 << index);
    } else {
        packedBooleans &= ~(1 << index);
    }
}

function getBoolean(uint index) public view returns (bool) {
    return (packedBooleans & (1 << index)) != 0;
}
```

4. Uint* vs Uint256

The Ethereum Virtual Machine (EVM) uses a specific format for storing integers, known as "big endian" format. When a variable of type uint* (an unsigned integer less than 256 bits) is used in a contract, it must first be converted to the uint256 format, which consumes more gas. To mitigate this issue, developers can use unsigned integers with a maximum bit size of 128 bits (also known as Variables Packing pattern). This can help reduce the amount of gas consumed when cramming many variables into a single storage slot. Additionally, using uint256 variables instead of uint* can help avoid the need for conversion and save gas. (uint*=> uint256)

5. Bytes vs Strings

The use of strings in smart contracts should be kept to a minimum, as they consume more gas than bytes. Constant strings must fit inside 32 bytes to prevent wasting memory. To avoid including unnecessary strings in the bytecode, it is important to minimize the use of strings and use bytes instead, where possible. This can help reduce the amount of gas consumed by the contract and improve its overall efficiency. (string => byte32)

```
bytes32 public efficient = "Efficient";
string public inefficient = "Inefficient";
```

6. Fixed Size Arrays

In Solidity, variables with a fixed size are more efficient in terms of gas consumption when compared to those with a variable size. This is because variables with a fixed size have a predetermined amount of memory allocated for them, whereas variables with a variable size have to have memory allocated dynamically, which consumes more gas.

```
// Inefficient: Dynamic size array
uint[] public dynamicArray;

// Efficient: Fixed size array
uint[10] public fixedArray;
```

7. Explicitly Initialize with Zero

In Solidity, variables when a variable is declared without an explicit value, it is automatically initialized with a default value of zero. To optimize gas consumption in a Solidity contract, it is important to be mindful of this behaviour and avoid explicitly initializing variables with zero when it is their default value. This small change can lead to a significant reduction in the overall gas consumption of the contract.

8. Memory vs Calldata

The cheapest place to store value is in call data. So changing the memory parameter to calldata will save us some more gas.

```
// Function using memory for struct argument
function doSomethingMemory(Data memory data) public
     returns (uint) {
    return data.id;
}

// Function using calldata for struct argument
function doSomethingCalldata(Data calldata data)
     external returns (uint) {
    return data.id;
}
```

9. Freeing Storage

In Solidity you can use “delete” command to release the storage if you are not using it anymore to get some gas back.

10. Internal and External Functions

Use restrictive visibility (internal, private vs external, public) as much as possible.

Visibility	Contract	Derived Contract	External Contract
public	✓	✓	✓
private	✓	✗	✗
internal	✓	✓	✗
external	✓	✓	✓

11. Constant Variables (CV)

Use constant for unchangeable values. Since the compiler does not save these variables, they are substituted with constant expressions that may be calculated to a single value by the optimizer.

12. Immutable Variables (IV)

Immutable variables are similar to constant variable but the difference is they can be declared once in the constructor or in the declaration time. However they cannot be read in the constructor. After declaration they cannot be changed. Note that immutable modifier can be used by Solidity version >= 0.6.5.

```
contract ImmutableVariable {
    uint public immutable creationTime;

    constructor() {
        creationTime = block.timestamp;  // Set once at
            deployment
    }

    function getCreationTime() public view returns (uint) {
        return creationTime;
    }
```

Operation Optimization Patterns

13. Reducing Expressions

Simplify logic expressions.

```
bool result = !(a || b); // Instead of: !a && !b
```

14. Short-Circuiting

Leverage short-circuit evaluations.

```
if (x > 0 || expensiveFunction()) {
    // Short-circuit if x > 0
}
```

15. Write Values Directly

uint256 public value = 420; // Instead of calculating: 20 * 21

16. Single Line Swap

Swap without a temporary variable.

(a, b) = (b, a);

17. Limit Number of Functions

Combine smaller functions if possible.

18. Limit Modifiers

Use inline modifier logic.

```
contract LimitModifiers {
    modifier OutofRange(uint value) {
        require(value <= 100, "Value out of range");
        _;
    }
    function process(uint value) public OutofRange(value)
         returns(uint) {
        return value * 2;
    }}
```

Before Limiting Modifiers

```
contract LimitModifiers {
    function process(uint value) public returns(uint) {
        require(value <= 100, "Value out of range");
        return value * 2;
    }}
```

After Limiting Modifiers

Loop Optimization Patterns

19. Avoid Nested Loops when possible

20. Combine Multiple Loops when possible

21. Avoid Repetitive Arithmetic in Loops

Cache results of expensive calculations.

```
uint cachedValue = expensiveCalculation();
for (uint i = 0; i < data.length; i++) {
    use(cachedValue);
}
```

22. Cache Storage Variables

Do not access or change storage values directly during loops.

```
// The initial contract
  contract SimpleContract {
  uint256 public s_total;
  function add(uint256 n) public {
     for (uint256 i = 0; i < n; i++) {
        s_total += i;
    }
 }
// Optimized contract with caching
  contract OptimizedContract {
  uint256 public s_total;
  function add(uint256 n) public {
     uint256 _s_total = s_total;
     for (uint256 i = 0; i < n; i++) {
        _s_total += i;
     }
     s_total = _s_total;
   }
  }
```

23. Cache Array Length and Array Items

When accessing attribute variables such as length for arrays we are being charged with more gas, we can work around this concept by storing the length of an array in a local variable rather than accessing it every loop iteration. Also when accessing members of an array with it specific index we are also causing more gas, hence we can also cache the value of array member to reduce even some more gas cost.

24. Optimize Loop Increment

Use `++i` instead of `i += 1`.

Mapping Optimization Patterns

25. Use Mapping Instead of Arrays

Solidity supports only arrays and maps for representing lists of data. Mappings are less expensive than arrays, which are packable and iterable. It is advised that mappings be used to handle data lists in order to conserve gas, unless it is necessary to iterate or feasible to pack data types. This is advantageous for both Storage and Memory. Using a mapping with an integer index as the key, you may manage an ordered list.

```
// Using array
contract ArrayContract {
uint256[] public numbers;
function addNumber(uint256 number) public {
  numbers.push(number);
 }

function getNumber(uint256 index) public view returns (
    uint256) {
  return numbers[index];
 }
}

// Using mapping
contract MappingContract {
mapping(uint256 => uint256) public numbers;
uint256 public size;
function addNumber(uint256 number) public {
  numbers[size] = number;
  size++;
}

function getNumber(uint256 index) public view returns (
    uint256) {
  return numbers[index];
 }
}
```

4. Implementation

A static analysis tool was developed using Python3 language and Slither library to identify gas-inefficient design patterns in Solidity smart contracts. We called this tool “SolSAT” standing for “Solidity Static Analysis Tool”.

Python was chosen for its simplicity, readability, and extensive libraries. It provides a robust ecosystem for data analysis, machine learning, and web development, making it an ideal language for building static analysis tools. Slither, a powerful static analysis tool specifically designed for Solidity smart contracts, was selected for its ability to parse Solidity code, analyze its structure, and identify potential vulnerabilities and optimization opportunities. By combining Python's versatility with Slither's specialized capabilities, the tool can effectively analyze Solidity contracts and provide actionable insights to developers.

4.1 Slither

Slither is a Solidity & Vyper static analysis framework written in Python3. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses [6].

GitHub Page: https://github.com/crytic/slither

Official Documentations: https://crytic.github.io/slither/slither.html

4.2 Design Pattern Detection

As a prototype version of the final tool, the current code currently implements the detection of the first 10 gas-inefficient design patterns introduced:

- **Struct Packing (SP):** Identifies structs with fields that can be packed into fewer storage slots.

- **State Variable Packing (SVP):** Detects state variables that can be packed into fewer storage slots.
- **Boolean Packing:** Identifies opportunities to pack multiple booleans into a single `uint256`.
- **Inefficient Integer Types:** Detects the use of inefficient integer types (e.g., `uint256` instead of `uint128`).
- **String vs. Bytes32:** Identifies the unnecessary use of `string` types and suggests using `bytes32` instead.
- **Dynamic Arrays:** Detects the use of dynamic arrays and suggests using fixed-size arrays where possible.
- **Zero Initialization:** Identifies auto zero initialization of variables and recommends explicitly initializing the variable.
- **Memory vs. Calldata:** Identifies function struct parameters that can be passed by calldata instead of memory.
- **Storage Release:** Detects opportunities to release unused storage slots using the `delete` keyword.
- **Visibility Optimizations:** Identifies functions that can be made `internal` or `private` to reduce gas costs.

4.3 Tool Functionality

The tool analyzes the Solidity source code and generates a report detailing:

- **Detected Patterns:** The specific patterns identified in the code.
- **Location:** The precise location of the pattern in the source code (contract, function, variable name etc.)
- **Recommended Optimization:** Specific suggestions for improving the code's gas efficiency.

4.4 Future Work

The tool will be further enhanced to:

- **Expand Pattern Detection:** Incorporate additional gas optimization techniques and patterns.
- **Provide More Precise Recommendations:** Generate tailored recommendations based on specific code contexts.
- **Integrate with IDEs:** Develop a plugin for popular IDEs to provide real-time feedback to developers.

- **Explore Machine Learning:** Utilize machine learning techniques to improve pattern recognition and optimization suggestions.

By continuously refining and expanding the tool's capabilities, we aim to empower developers to write more efficient and gas-optimized Solidity contracts.

References

- *[1] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. By Vitalik Buterin (2014)," 2014. Available: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf*

- *[2] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), Feb. 2020, doi: https://doi.org/10.1109/iwbose50093.2020.9050163.*

- *[3] Sourena Khanzadeh, Noama Samreen, and M. H. Alalfi, "Optimizing Gas Consumption in Ethereum Smart Contracts: Best Practices and Techniques," Oct. 2023, doi: https://doi.org/10.1109/qrs-c60940.2023.00056.*

- *[4] W. Zou et al., "Smart Contract Development: Challenges and Opportunities," IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. 1–1, 2019, doi: https://doi.org/10.1109/tse.2019.2942301.*

- *[5] G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER," 2022. Available: https://ethereum.github.io/yellowpaper/paper.pdf*

- *[6] "crytic/slither," GitHub, Jun. 02, 2021. https://github.com/crytic/slither*