Journal of Engineering Research

Volume 8
Issue 1 *issue 1*

Article 25

2024

In-Depth Examination of Gas Consumption in E-Will Smart Contract: A Case Study

Manal Mansour
Faculty of Engineering, Shoubra, Benha University, Egypt, manal.abdelmawgood@feng.bu.edu.eg

May Salama
Faculty of Engineering, Shoubra, Benha University, Egypt, may.mohamed@feng.bu.edu.eg

Hala Helmi
Information Technology and Computer Science, ITCS, Nile University, Giza, Egypt,
hala.zayed@feng.bu.edu.eg

Mona F.M Mursi
Faculty of Engineering, Shoubra, Benha University, Egypt, mona.mursi@feng.bu.edu.eg

Recommended Citation

Mansour, Manal; Salama, May; Helmi, Hala; and F.M Mursi, Mona (2024) "In-Depth Examination of Gas Consumption in E-Will Smart Contract: A Case Study," *Journal of Engineering Research*: Vol. 8: Iss. 1, Article 25.
Available at: https://digitalcommons.aaru.edu.jo/erjeng/vol8/iss1/25

In-Depth Examination of Gas Consumption in E-Will Contract: A Case Study

Manal Mansour 1*, May A. Salama1, Hala H. Zayed 2, Mona F.M. Mursi1
1 Faculty of Engineering, Shoubra, Benha University, Egypt.
2 Information Technology and Computer Science, ITCS, Nile University, Egypt.
*Corresponding author(s). E-mail(s): Manal.abdelmawgood@feng.bu.edu.eg.
Contributing authors: may.mohamed@feng.bu.edu.eg; hala.zayed@feng.bu.edu.eg; mona.mursi@feng.bu.edu.eg.

Abstract- In recent years, blockchain technology, coupled with smart contracts, has played a pivotal role in the development of distributed applications. Numerous case studies have emerged, showcasing the remarkable potential of this technology across various applications. Despite its widespread adoption in the industry, there exists a significant gap between the practical implementation of blockchain and the analytical and academic studies dedicated to understanding its nuances.

This paper aims to bridge this divide by presenting an empirical case study focused on the e-will contract, with a specific emphasis on gas-related challenges. By closely examining the e-will contract case study, we seek to provide a clearer understanding of the real-world implications of blockchain technology, addressing the potential challenges related to gas consumption.

Keywords- Blockchain; Empirical Study; Smart Contracts; Gas Consuming; E-Will.

I. Introduction

Smart contracts leveraging blockchain technologies have gathered significant attention in both emerging business applications and the scientific community. This is due to their special characteristics such as immutability, security, integrity, and tractability. The process of developing high-performing and secure contracts on Ethereum, the leading smart contract platform today, is a challenging task as developing smart contracts on the Ethereum network requires a different engineering approach from that most web and mobile developers are familiar with. The decisions made by developers such as the types of variables used, the types of data structures used, the number of cycles, the kind of instructions, and how they are initialized highly affect the gas consumption of a smart contract [1].

Additional challenges arise from Gas consumption, Gas is the metric that manages the execution of smart contracts on platforms like Ethereum. Many studies expressed the importance of closely monitoring gas consumption, with primary reasons being that gas equates to real money and transactions may fail if there's not enough gas. Moreover, accurately estimating the required gas for a specific smart contract execution is challenging[2]. Any mistake in writing or designing smart contracts has significant financial impacts compared to bugs in regular applications [3].

A study in [4] discussed 16 different detected exception types in the Ethereum network, and further grouped them into 6 major categories. the study stated that, out of the gas problem (occurs when the actual gas cost is greater than the provided gas cost) along with explicit revert problem (that causes a rollback of the state to just before the transaction), are the most commonly seen types of exceptions in the Ethereum network. When the study considered all the transactions, out of gas alone accounts for more than 90% of all exceptions detected, with explicit revert taking another 8%.

Another study in [5] analyzes the whole Ethereum blockchain with 6.3 million contracts deployed on the entire blockchain in 2019. They report numerous gas-focused vulnerability contracts holding a total value exceeding $2.8B.

In [6], the study mentions that if the cost of executing a function increases with time. It may cause the program to stop working at a certain point. So, computing the gas consumption helps identify such programming errors. If a smart contract is not optimized, it costs more than the required gas, and therefore, the user will be overcharged. Hence, optimization techniques are required to apply to smart contracts before deploying them onto the main network.

The objective of this work is to address the issue of gas consumption by utilizing a solidity scan tool, outlining a range of suggested patterns and advice for smart contract development to minimize gas usage.

The case study chosen for this study is E-will. Aa according to the rapid advancement of the social economy, our society is experiencing a gradual aging trend, leading to a rise in family disputes over inheritance. Wills play a crucial role in allowing individuals to dictate how their assets are distributed after their death, mitigating potential conflicts among heirs, and averting legal appeals. The research introduces a traceable online E-will system based on blockchain and smart contract technology as a case study for testing gas consumption.

This paper is structured as follows: Section 2 presents a background for Ethereum Virtual Machine (EVM) and the Gas concept in Ethereum. Section 3 shows in detail the structure of the E-will contract's framework. Testing the contract's gas consumption to identify potential issues along with suggested solutions to reduce gas consumption is presented in section 4. Finally, section 5 concludes the work.

II. Background

A. Smart Contract and Ethereum Virtual Machine

A smart contract is a self-executing digital contract with the terms of the agreement directly encoded into computer programs. These contracts run on blockchain platforms, and one of the most notable platforms for smart contracts is Ethereum. Ethereum Virtual Machine (EVM) is the runtime environment in which smart contracts on the Ethereum blockchain operate. Figure 1 illustrates the order of smart contract execution on the Ethereum blockchain. First, the agreement is reached, and a developer uses Solidity, a programming language designed for developing smart contracts on Ethereum, to encode it. The code is compiled into bytecode for the Ethereum Virtual Machine (EVM). Miners play a role in processing the contract onto the blockchain. After deployment, the contract undergoes processing on the specified event date, activated by the written code. The contract's execution results in the release of payment to the designated party and this transaction can be subsequently verified by anyone [7].

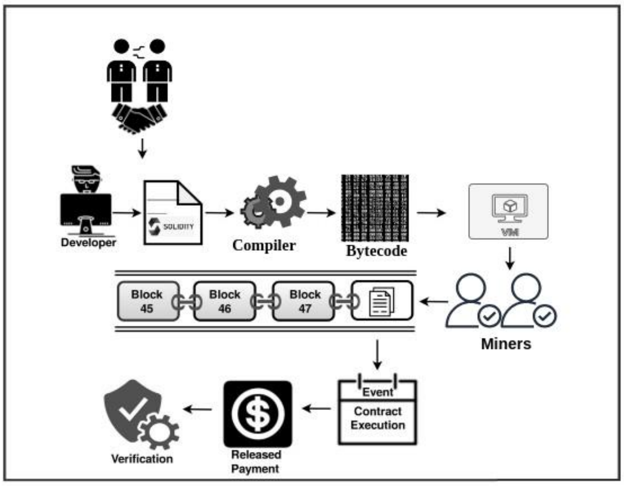
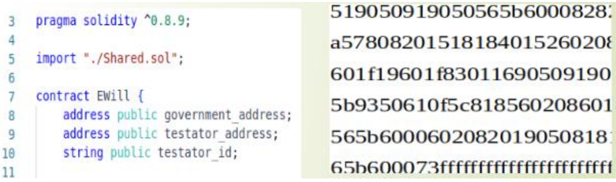


Figure 1: The Entire Process of Executing Smart Contract over Blockchain

Bytecodes, represented by hexadecimal numbers, are close to opcodes and are the codes executed by the Ethereum Virtual Machine (EVM). Figure 2 shows an example of the byte code generated from the E-will smart contract.



```
pragma solidity ^0.8.9;

import "./Shared.sol";

contract EWill {
    address public government_address;
    address public testator_address;
    string public testator_id;

```

```
519050919050565b6000828
a5780820151818401526020
601f19601f8301169050919
5b9350610f5c81856020860
565b6000602082019050818
65b600073ffffffffffffffffffff
```

Figure 2:Snippet of E-will Bytecode.

B. Gas Metric in Ethereum

Gas, in Ethereum, is denominated in gwei, one-billionth of one Ether the cryptocurrency in Ethereum network, and serves as a computational metric. It represents the cost associated with interacting and transacting with smart contracts. It is computed using a straightforward formula 1 [8]:

$$GasConsumed = GasPricexGasUsed \quad (1)$$

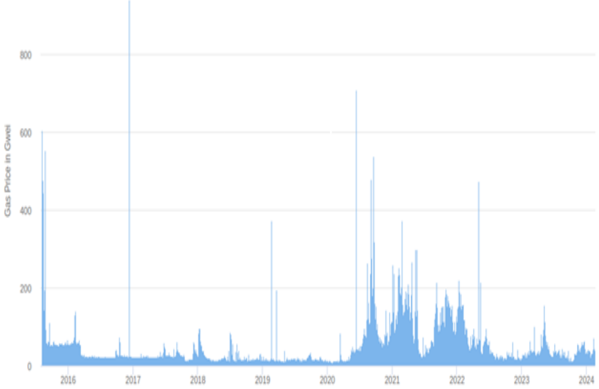


Figure 3: Average Gas Price

These gas fees help to keep the Ethereum network secure. By requiring a fee for every computation executed on the network, it prevents attackers from spamming the network and avoids accidental or hostile infinite loops or other computational wastage in code [5].

The Gas price is not constant. Figure 3[9] shows an example of the variation of Gas prices over time. The quantity of gas used is dictated by the specific operations executed within the smart contract. In Solidity, when a smart contract undergoes compilation, it transforms into a series of "operation codes" or opcodes, denoted by abbreviations such as ADD for addition and MUL for multiplication. The study in [10], initially outlined this system, which contains a comprehensive list of all opcodes along with their descriptions. Each opcode is assigned a specific amount of gas, serving as a metric for the computational effort needed to execute that particular operation as illustrated in table 1.

Table 1: Gas Costs in Ethereum

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
JUMP	8	Unconditional Jump
SSTORE	5,000/20,000	Storage operation
BALANCE	400	Get the balance of an account
CALL	25,000	Create a new account
CREATE	32,000	Create a new account
TRANSFERE	21,000	Transfer money

For example, the execution of the view function in the smart contract is cost-free, and there is no charge associated with these operations. However, if you intend to transfer funds from one wallet to another through the smart contract, you are required to cover the minimum gas limit as well as the gas cost associated with each opcode line.

C. Gas Limit

Each block has a gas limit, representing the maximum gas consumption allowed for all included transactions. This limit dictates the maximum number of transactions

permitted in a block. In the permissioned Ethereum network, the gas limit is set to the maximum limit allowed for the block, and if the computations of the block exceed this limit the contract will raise 'out of gas flag or revert'. In this case, the consumed gas will not be returned to the caller. On the other hand, if the computations within the contract are lower than the limit it will return to the caller. So, optimizing the consumed gas within the smart contract by eliminating unbounded mass operations e.g. loops, complex logic, inefficient data structure and large amounts of data can help in mitigating the risk of gas problems [11].
The current gas limit for each block is 30 million gas per block at the time of writing this article 2024, meaning around 1400 transfer transactions that each have a transaction gas limit of 21000 (simple money transfer) can fit in 1 block [12].

III. Case Study: E-Will System

Blockchain technology, being a decentralized ledger system, ensures data integrity by employing the Merkle tree hash technique [13]. This means that any attempt to modify data within the chain renders it invalid. As a result, everyone on the network is informed of every change that occurs, whether during or after the testator's death. Additionally, the automated execution of smart contracts can be used for addressing a critical concern related to wills, particularly the challenge posed by wallet credentials. This issue gained prominence in 2020 when Chainalysis, a cryptocurrency research company, reported that approximately one-fifth of the bitcoins in circulation at that time, valued at over $175 billion, were inaccessible due to the death of their owners. In blockchain, accessing wallets is only possible with the corresponding private key. The strategic implementation of E-wills on the blockchain holds substantial promise in mitigating such challenges. Finally, the immutability characteristic of blockchain offers immutable and tamper-resistant ledger. Once a will is recorded on the blockchain, it becomes a part of a distributed and decentralized network of nodes, immutable timestamping provided by blockchain records the exact time of will creation or modification, offering a verifiable record of events crucial in legal contexts and aiding in the resolution of disputes related to the timing of actions within the E-will system.

A. System Architecture

The proposed system utilizes blockchain technology to develop an innovative online will system. By establishing a permissioned Ethereum chain and implementing smart contracts using Solidity. This study then aims to optimize Gas usage and minimize the issues raised by the contract calls. Figure 4 shows the interaction of the system between the different actors through the blockchain.

Notations:
₲: Government Contract, the base contract that is deployed once in the network and the address is public for all the citizens to deal with.

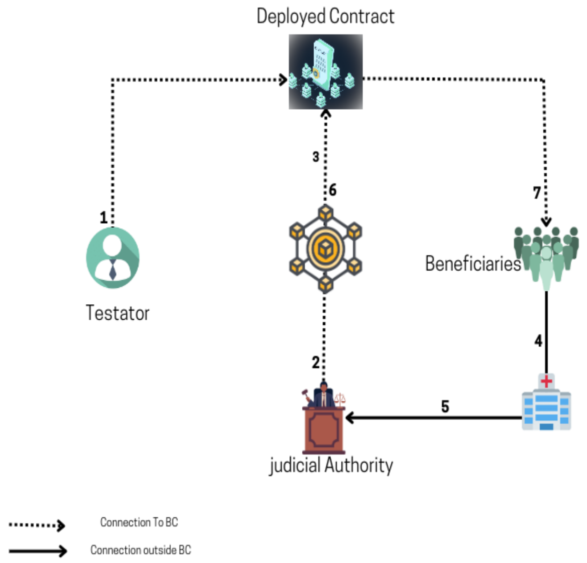


Figure 4: Interaction with E-will Contract.

𝓙𝓐: Judicial Authority manager nodes.
𝓡𝓔: Real estate managers nodes that approve the testator ownership of the asset.

The interaction is carried out among 4 main actors:
1- Judicial Authority 𝓙𝓐: The nodes responsible for accepting or rejecting the registration of wills in the blockchain.
2- Real estate Manager 𝓡𝓔: The nodes that approve the ownership of real assets.
3- Testator: The person who aims to write his will.
4- Beneficiaries: The owner of the money or assets after the testator's death.

B. Steps for Creating E-will:

Step 1: The testator, aiming to create an E-will, initiates the registration process by applying through the public government contract (₲), accessible to all citizens. The E-will contract is deployed with a pending status to be reviewed by the public judicial manager.

Step 2: The judicial manager receives the request, conducts a thorough review, and approves it if the citizen has not issued any wills previously.

Step 3: The contract status is then updated to be approved or rejected, and any instances of illegal behavior detected by the judicial authority result in the destruction of the contract.

Step 4,5: Upon the testator's death, beneficiaries obtain a certificate from the insurance organization and submit it to the judicial manager.

Step 6: The judicial manager, in turn, activates the running method that exists in the contract for the testator after death.

Step 7: The rules that exist in the contract will be achieved consequently. The E-will contract is represented by structures outlined in Algorithm 1 and Algorithm 2, while the UML diagram for the deployed E-will contract is presented in figure 5.

Algorithm 1: Creating E-will

Input ↦ Request submitted to the $\mathbb{G}$ contract using ID.
Process ↦ $\mathcal{JA}$ reviews the request.
approves the deployment if satisfactory.
Otherwise, the self-destruct function is invoked, leading to the contract's termination. Upon approval, the contract status is updated to "approved." The testator can then proceed to add or remove any beneficiaries from their will.
Output ↦ E-will contract with two distinct arrays: one for monetary beneficiaries and the other for assets beneficiaries ready for addition by the testator.

Algorithm 2: After Testator death

Input ↦ $\mathbb{G}$ contract initiates the E-will contract by invoking the 'mark as dead' function within the E-will contract.
Process ↦ The prioritization of money beneficiaries is determined according to the specifications outlined by the testator, and the approval of assets is overseen by the asset manager $\mathcal{RE}$. The transfer of assets to the beneficiaries leads to updates in the corresponding fields.
Output ↦ The asset history is duly updated, and funds are seamlessly transferred to the wallets of the designated money beneficiaries.

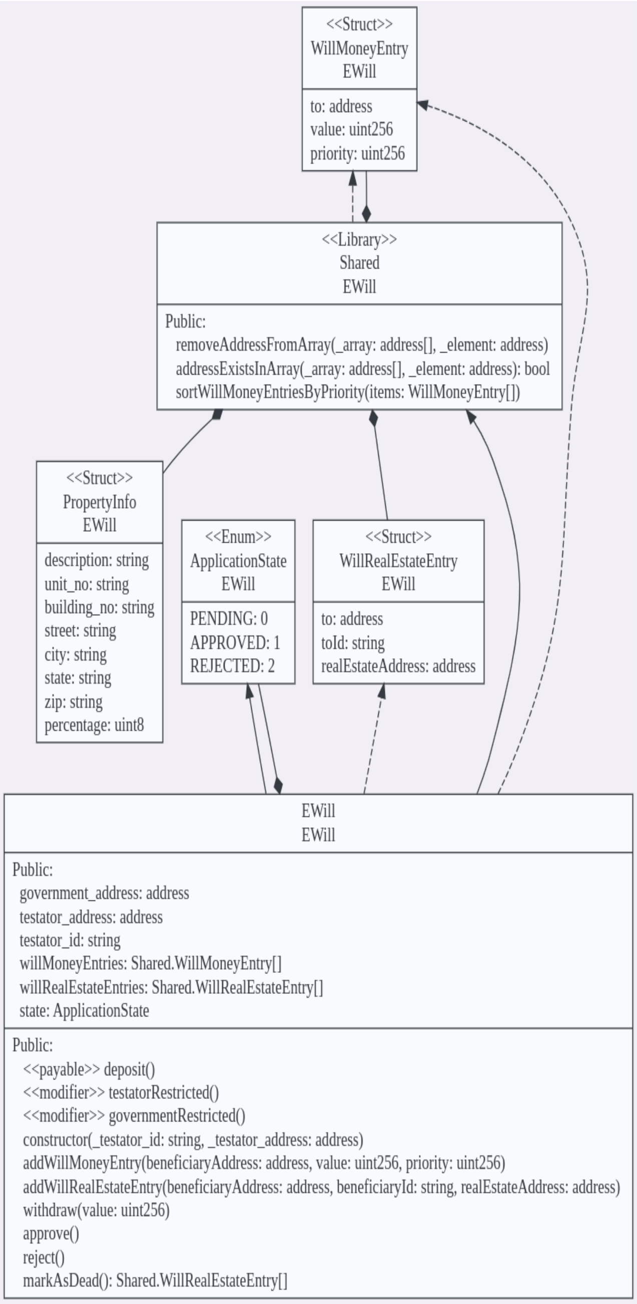

<<Struct>>
WillMoneyEntry
EWill
to: address
value: uint256
priority: uint256
<<Library>>
Shared
EWill
Public:
removeAddressFromArray(_array: address[], _element: address)
addressExistsInArray(_array: address[], _element: address): bool
sortWillMoneyEntriesByPriority(items: WillMoneyEntry[])
<<Struct>>
PropertyInfo
EWill
description: string
unit_no: string
building_no: string
street: string
city: string
state: string
zip: string
percentage: uint8
<<Enum>>
ApplicationState
EWill
PENDING: 0
APPROVED: 1
REJECTED: 2
<<Struct>>
WillRealEstateEntry
EWill
to: address
toId: string
realEstateAddress: address
EWill
EWill
Public:
government_address: address
testator_address: address
testator_id: string
willMoneyEntries: Shared.WillMoneyEntry[]
willRealEstateEntries: Shared.WillRealEstateEntry[]
state: ApplicationState
Public:
<<payable>> deposit()
<<modifier>> testatorRestricted()
<<modifier>> governmentRestricted()
constructor(_testator_id: string, _testator_address: address)
addWillMoneyEntry(beneficiaryAddress: address, value: uint256, priority: uint256)
addWillRealEstateEntry(beneficiaryAddress: address, beneficiaryId: string, realEstateAddress: address)
withdraw(value: uint256)
approve()
reject()
markAsDead(): Shared.WillRealEstateEntry[]

Figure 5: UML Digaram for E-will Contract.

IV. Analysis of E-will Smart Contract

Before deploying any smart contract to the blockchain, it is imperative to establish criteria for testing the functionality of the smart contract. Various tools are available for assessing potential vulnerabilities and attacks that could pose obstacles to the seamless operation of the smart contract OYENTE, SECURIFY[14], MYTHRIL[6], and SOLSCAN. The immutability of the blockchain further emphasizes the criticality of this testing process. Failing to detect the issues during testing could lead to severe consequences. Vulnerabilities left unaddressed may persist in the system, potentially compromising the security and functionality of the smart contract.

A. Detected Gas Vulnerabilities within E-Will using Solidity Scan:

We employed the Solidity scanner to assess gas consumption vulnerabilities within the E-will contract. The identified vulnerabilities include no critical risk functions, and thirteen lines are identified as causing gas consumption. The representation of the vulnerabilities exists in figure 6.

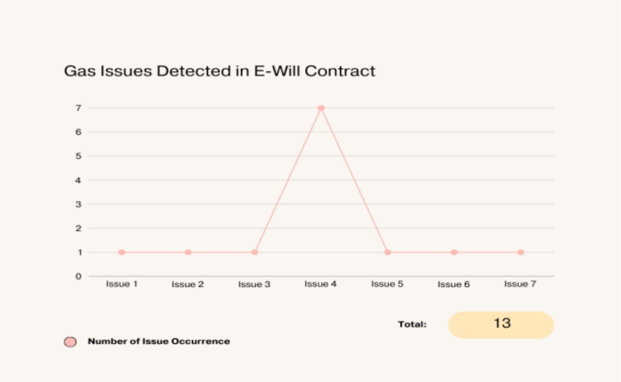
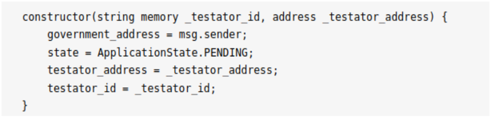


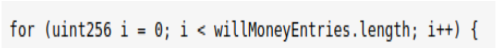
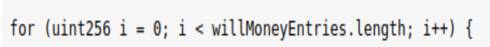
Figure 6: Detected Gas Issues

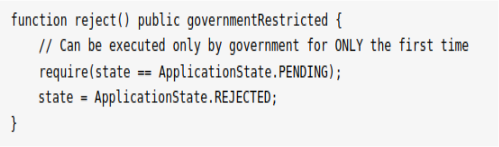
The Gas issues found are 13 lines classified into 7 categories. Each issue is illustrated and the code snippet that causes this issue is represented in table2.

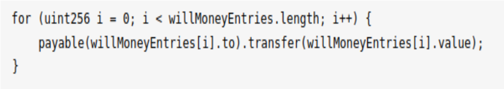
Table 2: Lines Flagged as Gas Consuming

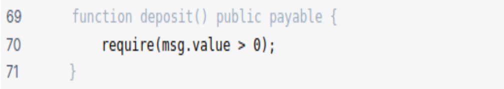
Pattern	Code/Snippet	Occurrence

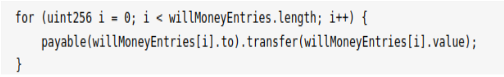
1	
	`constructor(string memory _testator_id, address _testator_address) {`
	`    government_address = msg.sender;`
	`    state = ApplicationState.PENDING;`
	`    testator_address = _testator_address;`
	`    testator_id = _testator_id;`
	`}`
		1

2	 `for (uint256 i = 0; i < willMoneyEntries.length; i++) {`	1
3	 `for (uint256 i = 0; i < willMoneyEntries.length; i++) {`	1

4	
	`function reject() public governmentRestricted {`
	`    // Can be executed only by government for ONLY the first time`
	`    require(state == ApplicationState.PENDING);`
	`    state = ApplicationState.REJECTED;`
	`}`
		7

5	
	`for (uint256 i = 0; i < willMoneyEntries.length; i++) {`
	`    payable(willMoneyEntries[i].to).transfer(willMoneyEntries[i].value);`
	`}`
		1

6	
	`function deposit() public payable {`
	`    require(msg.value > 0);`
	`}`
		1

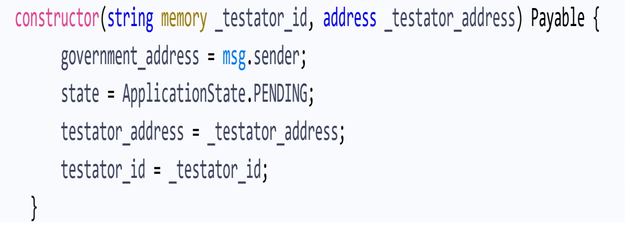
7	
	`for (uint256 i = 0; i < willMoneyEntries.length; i++) {`
	`    payable(willMoneyEntries[i].to).transfer(willMoneyEntries[i].value);`
	`}`
		1

B. Suggested Solutions:

1- Pattern 1: Payable and Non-Payable Function Impact

Let S be the set of constructors defined as payable, Δ represents the potential opcode reduction, and G denotes the saved gas.

Tests show that developers can achieve Δ≈10 opcodes and G units of gas savings by making constructors payable. It is crucial to acknowledge that this optimization strategy introduces risks, as payable constructors can accept ETH during deployment. But in our example, the E-will contract is deployed only by the government contract, so it is considered a trusted node the code is updated to be as presented in Listing 1.

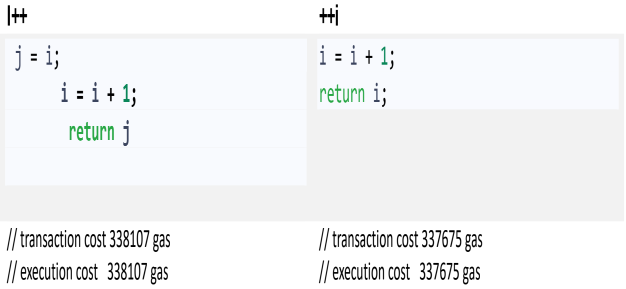


```
constructor(string memory _testator_id, address _testator_address) Payable {
    government_address = msg.sender;
    state = ApplicationState.PENDING;
    testator_address = _testator_address;
    testator_id = _testator_id;
}
```

Listing 1: An Example for a Payable Constructor

2- Pattern (2-3-6) Arithmetic Gas Consumption

The analysis shows that verifying the variable within a loop in Solidity incurs gas consumption. Therefore, any checks inside the loop should be conducted prior to its execution. Additionally, various arithmetic operations may consume differing amounts of gas. In our illustration, we modified i++ to ++i. Throughout the test cases, it was observed that the last method incurs a relatively lower gas consumption compared to the other methods of increments. That is because the steps for applying it in the same function using ++i and i++ are as described in Listing 2.

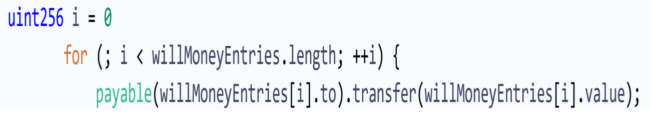


```
I++                                  ++i
j = i;                               i = i + 1;
     i = i + 1;                      return i;
      return j

// transaction cost 338107 gas       // transaction cost 337675 gas
// execution cost   338107 gas       // execution cost   337675 gas
```

Listing 2: Gas Consumption for Mathematic Operations

Listing 3 shows an example of updated loops to reduce Gas consumption.



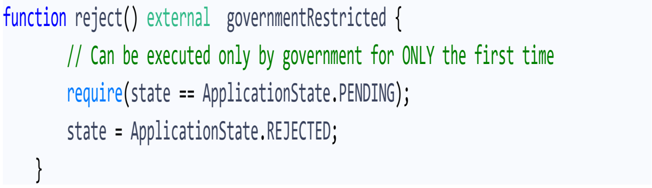
```
uint256 i = 0
      for (; i < willMoneyEntries.length; ++i) {
          payable(willMoneyEntries[i].to).transfer(willMoneyEntries[i].value);
```

Listing 3:Example for Enhance checker and Arithmetic operations.

3- Pattern 4: Public Modifier Gas Consumption

Let F_{public} denote a function with a public visibility modifier identified without internal calls.

The disparity in gas consumption between public (G_{public}) and external ($G_{external}$) functions becomes apparent, particularly with extensive data arrays. This divergence arises from the fact that, in the case of public functions, Solidity duplicates arguments to memory, incurring higher gas costs ($G_{public}> G_{external}$). Public variables implicitly generate a getter function, contributing to both the contract's size and overall gas usage. Conversely, external functions read from call data, which is more cost-effective compared to memory allocation. Additionally, the primary concern lies in the gas consumption induced by the public modifier. This vulnerability is notable because public variables result in an additional 22 gas consumption. Listing 4 shows external implementation instead of the public modifier.



```
function reject() external  governmentRestricted {
       // Can be executed only by government for ONLY the first time
       require(state == ApplicationState.PENDING);
       state = ApplicationState.REJECTED;
   }
```

Listing 4: Example for External call for Government Restricted Function

4- Pattern 5: Maps Vs Arrays Gas Consumption:

The ratio of mapping to mapping & array is modified this pattern is calculated by $N_{mapping}/N_{mapping}+N_{array}$ higher value of equation metric is related to a lower gas consumption [15].

Let L represent the length of the array, and G_{read} denote the gas consumption associated with reading the length of the array during each iteration of the loop. In each iteration, G_{read} exceeds the necessary gas usage. Under the most favorable circumstance, where the length is read from a memory variable and stored in the stack, a savings of approximately 3 gas per iteration can be achieved.

In contrast, the least favorable scenario involves external calls during each iteration, resulting in a significant waste of gas. The array "willMoneyEntries" has been identified for use inside a loop without caching its value in memory. In our example, caching the array length (L) is not feasible due to the testator's ability to add beneficiaries during their lifetime. Therefore, the array can only be cached after the testator's death. The "mark as dead" function (F_{dead}) may be implemented to enforce caching the array size (S) before initiating the distribution of the will.

C. Contract Analysis After Gas Optimization

The E-will contract is subject to a renewed evaluation employing the Solidity scan tool. The findings, elucidated in the accompanying figure, reveal a notable reduction, specifically from 13 to 2 gas alerts, as resulted in figure 7. It is imperative to note that the remaining alerts originate from the array code. The arrays in our contract, pertaining to the addresses of beneficiaries for the testator, are anticipated to maintain a relatively limited size. In contrast, the arrays associated with citizens have the potential to exceed a multitude of millions of entries. Consequently, we have opted for a strategic replacement by implementing a mapping criterion to optimize performance and mitigate gas consumption concerns.

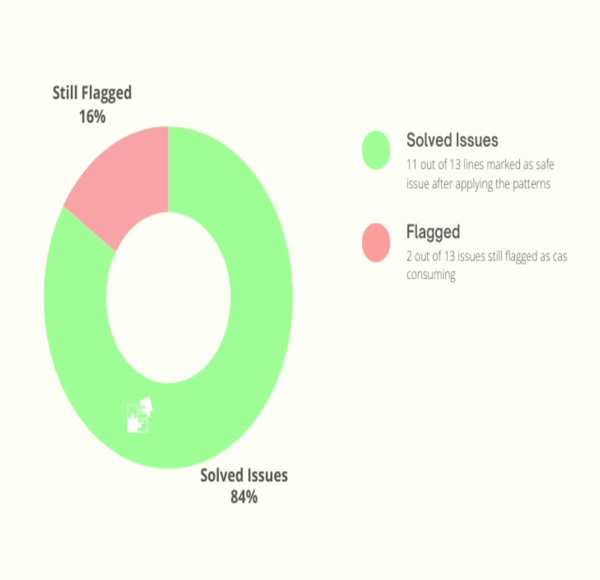


Figure 7: Detected Issues after Modifying the Contract

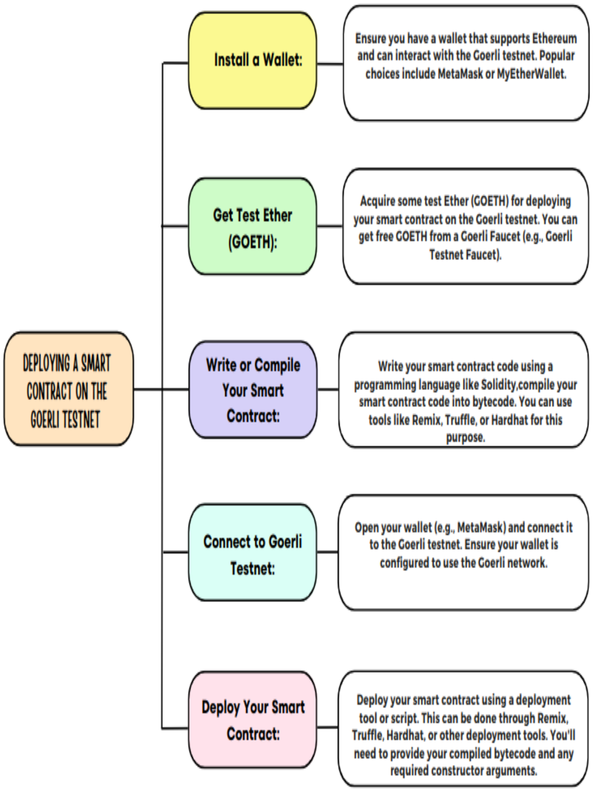
Table 3 shows the findings that can be followed by the solidity developers to mitigate the gas issues.

Table 3:Relations of Gas Consumption vs Identified Metrics

Metric	Gas Consumption G_C	Description
Global Variables (G_V)	$G_C \propto G_V$	Higher no of G_v related to higher gas consumption.
Public Modifiers (P_M)	$G_C \propto P_M$	Higher no of P_m related to higher gas consumption.
External Variables (E_V)	$G_C \propto 1/E_V$	Higher no of E_V lower gas consumption.
String Variable (S_V) and Byte Variable (B_V) =S_V/B_V	$S_v/B_v \propto G_C$	The greater no of S_V related to B_V the greater the gas consumed
Loops Occurrence L_O	$L_O \propto G_C$	The greater L_O the more gas consumed
Mappings and Arrays N mappings/ (N mappings + N arrays)	N mappings/ (N mappings + N arrays) $\propto 1/GC$	higher values of this metric related to a lower gas consumption.

D. Evaluating the money saved in deploying the optimized smart contracts.

We deploy both the original and optimized smart contract in order to calculate the cost savings associated with implementing the optimized smart contracts. We observe that 11740 units of gas are saved when we compare the gas consumption for deploying the optimized contracts with that for deploying the original ones. As ETH price is 3786$ and gas price is 43 Gwei according to the records in March 2024. The total saved money for the optimized contract is around 40 USD.


DEPLOYING A SMART CONTRACT ON THE GOERLI TESTNET
Install a Wallet:
Get Test Ether (GOETH):
Write or Compile Your Smart Contract:
Connect to Goerli Testnet:
Deploy Your Smart Contract:

Ensure you have a wallet that supports Ethereum and can interact with the Goerli testnet. Popular choices include MetaMask or MyEtherWallet.

Acquire some test Ether (GOETH) for deploying your smart contract on the Goerli testnet. You can get free GOETH from a Goerli Faucet (e.g., Goerli Testnet Faucet).

Write your smart contract code using a programming language like Solidity,compile your smart contract code into bytecode. You can use tools like Remix, Truffle, or Hardhat for this purpose.

Open your wallet (e.g., MetaMask) and connect it to the Goerli testnet. Ensure your wallet is configured to use the Goerli network.

Deploy your smart contract using a deployment tool or script. This can be done through Remix, Truffle, Hardhat, or other deployment tools. You'll need to provide your compiled bytecode and any required constructor arguments.

Figure *8*: Steps for Deploying in Goerli Testnet

E. Real Deployment in Georli testnet

Deploying a smart contract on the Goerli testnet[16] allows developers to test their contracts in a simulated Ethereum environment before deploying them on the mainnet. This process involves several key steps. First, developers need to install a compatible wallet like metamask [10], [17], and deploying platforms such as Remix, Truffle, or Hardhat[18]. Figure 8 shows the steps followed to deploy the contract within the real network. After deployment, you can verify your smart contract on a block explorer like Etherscan. Interact with your deployed contract using its address. The Address of our deployed contract in Etherscan is in Listing 5.

```
https://goerli.etherscan.io/address/0xf8e8534d67f7b0
1a0241576d4a8b61b838cb0b06
```

Listing 5: Address of Contract in Goerli Network

Conclusion:

The costs associated with deploying and executing a smart contract are influenced by the implementation decisions made by developers. Inappropriate design choices may lead to higher gas consumption than necessary. In this paper, we explore the structure of the E-will contract by drawing a detailed UML diagram. The code is then compiled, and bytecode is extracted using the REMIX IDE. Solidity Scan is employed to evaluate the code quality, specifically focusing on gas consumption perspectives. An experiment shows that the framework doesn't have any critical issues, but there are 13 Gas flags distributed across 7 categories. The patterns used to address these issues enhance gas consumption by 84%. The E-will contract is then deployed using the MetaMask wallet on the Goerli network, making it available for any further tests and studies.

Acknowledgment

Funding: The authors declare that there is no funding received.

Conflicts of Interest: The authors declare that there is no conflict of interest regarding the publication of this paper.

References:

[1] W. Zou et al., "Smart contract development: Challenges and opportunities Smart contract development: Challenges and opportunities Pavneet Singh KOCHHAR Citation Citation Author Author Smart Contract Development: Challenges and Opportunities," 2021. [Online]. Available: https://ink.library.smu.edu.sg/sis_research

[2] F. Santos, "The DAO: A Million Dollar Lesson in Blockchain Governance," 2018.

[3] S. Rouhani and R. Deters, "Security, performance, and applications of smart contracts: A systematic survey," IEEE Access, vol. 7. Institute of Electrical and Electronics Engineers Inc., pp. 50759–50779, 2019. doi: 10.1109/ACCESS.2019.2911031.

[7] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart Contract: Attacks and Protections," IEEE Access, vol. 8, pp. 24416–24427, 2020, doi: 10.1109/ACCESS.2020.2970495.

[8] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," in IWBOSE 2020 - Proceedings of the 2020 IEEE 3rd International Workshop on Blockchain Oriented Software Engineering, Institute of Electrical and Electronics Engineers Inc., Feb. 2020, pp. 9–15. doi: 10.1109/IWBOSE50093.2020.9050163.

[9] "Etherscan," https://info.etherscan.com/.

[10] "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER EIP-150 REVISION."

[11] Fang Y, Zhou Z, Dai S, Yang J, Zhang H and Lu Y. (2024). PaVM: A Parallel Virtual Machine for Smart Contract Execution and Validation. IEEE Transactions on Parallel and Distributed Systems. 35:1. (186-202). Online publication date: 1-Jan-2024.

[12] https://ethresear.ch/t/on-block-sizes-gas-limits-and-scalability/18444

[13] Chen, Yi-Cheng & Chou, Yueh-Peng & Chou, Yung-Chen. (2019). An Image Authentication Scheme Using Merkle Tree Mechanisms. Future Internet. 11. 149. 10.3390/fi11070149.

[14] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," Jun. 2018, [Online]. Available: http://arxiv.org/abs/1806.01143

[15] A. Di Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora, "Profiling Gas Consumption in Solidity Smart Contracts," Aug. 2020, [Online]. Available: http://arxiv.org/abs/2008.05449

[16] "Testnet," https://goerli.etherscan.io/.

[17] C. Liu and T. Feng, "Blockchain based Multi-signature Smart Contract Electronic Seal Orienting Mobile IoT Terminals," 2023, doi: 10.21203/rs.3.rs-3419170/v1.

[18] "Evaluating Ethereum Development Environments."