Improving Gas Efficiency in Smart Contracts: Data-Driven Insights and LLM-Assisted Remediation

Yijie Ruan, Zhipeng Gao, Jiachi Chen, *Member, IEEE*, Lingfeng Bao, *Member, IEEE*, and Xiaohu Yang, *Member, IEEE*

***Abstract*—Smart contracts, primarily written in Solidity, are Turing-complete programs on platforms like Ethereum, requiring gas fees for deployment and execution. Gas quantifies computational costs, and inefficient contracts result in unnecessary expenses for developers and users. Gas optimization at the source code level has been studied in various related works; however, existing methods for summarizing gas-inefficient patterns primarily rely on author-defined rules or heuristic approaches, and their evaluations lack a labeled dataset. In this paper, we conduct a comprehensive empirical study on the issue of gas optimization in smart contracts. We begin by gathering audit reports from Code4rena, a well-known smart contract audit platform. These reports include both expert evaluations, conducted by professionals known as Wardens, and automated analyses generated by the platform's static analysis tool, 4naly3er. After filtering out false-positive gas optimization instances from the automated reports, we identify 2,095 instances of gas-inefficient patterns across 54 projects. We categorize these inefficiencies into 24 types using thematic analysis and find that static analysis tools often produce false positives and negatives. To address this, we propose a hybrid method combining static analysis and large language models (LLMs) to detect and repair gas inefficiencies. The static analysis tool identifies potential optimization opportunities, while the LLM refines these findings and suggests effective repairs. Our evaluation shows that our approach achieves a precision rate of 82.28% and a recall rate of 88.46%, and can save 919 units of gas per function on average during execution.**

***Index Terms*—Smart contracts, gas optimization, patterns, language models.**

Received 12 March 2025; revised 1 December 2025; accepted 23 January 2026. Date of publication 18 February 2026; date of current version 17 March 2026. This work was supported in part by Zhejiang Province "JianBingLingYan+X" Research and Development Plan 2025C02020, in part by Zhejiang Provincial Natural Science Foundation of China under Grant LQK26F020002, in part by the National Science Foundation of China under Grant 62372398, Grant 62572322, and Grant 72342025, and in part by the Fundamental Research Funds for the Central Universities under Grant 226-2025-00067. Recommended for acceptance by M. Zhou. *(Corresponding author: Lingfeng Bao.)*

Yijie Ruan, Zhipeng Gao, and Jiachi Chen are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310058, China (e-mail: 22321183@zju.edu.cn; zhipeng.gao@zju.edu.cn; chenjiachi@zju.edu.cn).

Lingfeng Bao and Xiaohu Yang are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310058, China, and also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310051, China (e-mail: lingfengbao@zju.edu.cn; yangxh@zju.edu.cn).

Digital Object Identifier 10.1109/TSE.2026.3658163

I. Introduction

ETHEREUM [1] stands as one of the most widely recognized public blockchains today, renowned not only for its decentralized, shared ledger that enables all users to engage in ledger update activities, but also for its ability to function as a "world computer" capable of hosting and executing programs. These programs are socalled smart contracts. A smart contract is an autonomous computer program that, once started, executes automatically and mandatorily according to the program logic defined beforehand [2]. In recent years, smart contracts have emerged as a critical and promising technology in the field of blockchains and decentralized applications [3]. Smart contracts are typically developed in a high-level language [4], with Solidity [5] being the most widely used. Due to the transparency, immutability, and decentralization of blockchain systems, they are widely adopted in various fields such as finance, security, and manufacturing [6].

Ethereum relies on the Ethereum Virtual Machine (EVM) as a global stack-based virtual machine for executing the bytecode of smart contracts. Each node in the Ethereum network maintains a synchronized copy of the blockchain, replaying all transactions to maintain consensus over the network's state. When a smart contract receives a transaction, it is executed on the EVM, consuming computing resources. To prevent malicious activities like Denial-of-Service (DoS) attacks that exploit these resources [7], Ethereum implements a gas mechanism [8]. Gas is a fee system where transactions, whether deploying or invoking smart contracts, incur gas costs. These costs help to prevent abuse by charging users based on the computational load they impose on the network. The deployment gas cost is determined by the size of the bytecode compiled from the smart contract, while the invocation gas is influenced by the number and type of operations executed in the contract on the EVM. This gas mechanism ensures the efficiency and security of Ethereum by discouraging excessive resource use.

Since the gas fee is paid in real money, gas optimization receives significant attention [9], [10], [11]. A gas-inefficient pattern refers to coding practices in smart contracts that lead to unnecessary gas consumption, thus increasing execution costs. To address this, the general approach of gas optimization is to detect instances of gas-inefficient patterns in smart contracts on

either high-level Solidity source code or low-level bytecode and then replace them with more gas-efficient code snippets. Gasper [12] is a pioneer in bytecode-level gas optimization, identifying inefficiencies such as redundant code and loops. Subsequent tools like Gas Reducer [13] and Gas Checker [14] are not publicly available. At the source level, approaches include GASOL [15], which optimizes SLOAD/SSTORE operations using local variables, and SOLIOS [16], which enhances loop patterns while increasing deployment costs and imposing loop verification constraints. Empirical studies further examine optimization strategies: Keilty et al. [17] analyze gas optimization in Solidity and adapt it to the Move language, proposing 11 optimization patterns, while Jiang et al.[18] leverage GPT-4 to detect gas-related code smells, with reported issues manually reviewed and classified into 26 distinct categories. However, to the best of our knowledge, these methods for summarizing gas-inefficient patterns rely on human experience or heuristic approaches, which can be subjective and inconsistent. Moreover, their tests are all based on the smart contracts in Ethereum without labeling.

In order to address the above problems, we started by collecting data from Code4rena to make a data-driven classification and build a labeled dataset for gas optimization in smart contracts. While various audit platforms contribute to detecting gas inefficiencies, Code4rena [19] stands out by explicitly providing a dedicated Gas Optimization section in its audit reports, highlighting various inefficiencies and suggested improvements. Code4rena publishes reports from expert reviews conducted by professionals, known as Warden audit reports, as well as automated reports generated by its static analysis tool, `4naly3er` [20]. The findings in these two types of reports are mutually exclusive. After filtering out false-positive gas optimization instances from the automated reports using a manual verification process, we identified 2,095 instances of gas-inefficient patterns, which were classified into 24 distinct types (as detailed in Section III-C) across 54 projects. We also collected the recommendations that fix gas inefficiencies for each instance. By analyzing the dataset, we observed that many inefficiencies reported by static analysis tools were either inaccurate (false positives) or overlooked (false negatives).

Therefore, considering the limitation of static analysis tool and the demonstrated effectiveness of large language models (LLMs) in tasks related to code analysis and generation [21], we developed an approach combining LLMs with static analysis for detecting and repairing gas-inefficiencies. Specifically, we use the static analysis tool to detect potential gas optimization opportunities within contract functions based on our findings at multiple levels of analysis. To enhance the LLM's ability to make informed decisions, we structure relevant context from the static analysis results, including variable dependencies, control flow constraints and other key factors. This information, along with the function's source code and customized prompts, enables the LLM to assess inefficiencies more accurately and recommend more suitable repairs.

For evaluation, from 10 Code4rena repositories issued between March 2024 and June 2024, we isolated 316 gas-inefficient instances with optimizations applied based on audit recommendations. We evaluated the effectiveness of our approach in three aspects: gas-inefficiency detection, repair, and gas savings. Our approach achieves 82.28% accuracy in detection, with 88.46% of the detected patterns fully repaired. Additionally, the method saves an average of 991 gas units per function, demonstrating notable efficiency improvements.

In summary, the contributions of our work are as follows:

1) We conduct an empirical study by collecting 2,095 real-world gas-inefficient instances from an audit platform, consolidating them into 24 distinct gas optimization categories across 54 projects and analyzing these categories to assess the limitations of static analysis tools.
2) We propose an approach that combines static analysis with large language models (LLMs) to detect and repair gas-inefficient patterns in Solidity code. Experiment results show that our method achieves 82.28% detection accuracy, 88.46% repair success, and saves an average of 991 gas units per function.

The remainder of this article is organized as follows. Section II introduces the background knowledge of the paper. Section III shows our data collection and classification process. Section IV presents our approach. Section V conducts the experimental evaluation. Section VI discusses the threats to validity and analyzes representative failure cases. Section VII reviews the related work on smart contract optimization, gas-related vulnerabilities, and gas estimation. Section IX provides data availability. Finally, Section VIII concludes the paper.

II. Background

In this section, we introduce background knowledge necessary for understanding our work.

A. Transaction and Gas Fees

In the EVM, a transaction is the fundamental unit of work that can be included in a block. It triggers a state transition, updating account data and transferring Ether across the network. Each transaction consumes gas, a unit that measures computational effort. The gas meter is the system that calculates the amount of gas to be charged for deploying and invoking a smart contract. The gas price refers to the amount of money the sender is willing to pay per unit of gas, which is determined by market conditions.

When the transaction is executed, it keeps a tally of the amount of gas used according to the gas meter. The gas used by a transaction consists of summing the gas associated with the size of its payload, the virtual machine instructions it executes, and the global storage it accesses. The total gas fee for a transaction is calculated by multiplying the gas used by the gas price.

B. Code4rena

Code4rena [19] is an organization that runs semi-public "audit contests" to crowd source smart contract security audits and gas optimization issues are commonly reported in these audit contests. An audit report on Code4rena is a comprehensive document assessing the security and functionality of smart

```
-   l1Bridge = _l1Bridge;
-   l2TokenProxy = _l2TokenProxy;
-   if (block.chainid != ERA_CHAIN_ID) {
-       //non-revert logic
-   } else {
-       require(_l1LegecyBridge != address(0), "bf2
    ");
-   }
+   if (block.chainid == ERA_CHAIN_ID) {
+       require(_l1LegecyBridge != address(0), "bf2
    ");
+   } else {
+       //non-revert logic
+   }
+   l1Bridge = _l1Bridge;
+   l2TokenProxy = _l2TokenProxy;
```

Fig. 1. A finding related to gas optimization in an audit report. The suggestion is to revert earlier and avoid unnecessary state variable reads by executing block.chainid == ERA_CHAIN_ID first to save gas.

contracts or decentralized applications (DApps) submitted for review. It generally contains a summary, audit scope, methodology, detailed findings with identified issues and recommendations, a conclusion, and any relevant appendices. While there are other smart contract audit platforms like Solodit[1], Audit Collection[2], and Audit Hero[3], Code4rena stands out by uniting a diverse community of security experts (known as Wardens) who participate in real-time, incentivized audits. The findings identified by wardens are subsequently reviewed by judges and confirmed by sponsors, ensuring reliable and trustworthy results. The community-driven model promotes transparency and utilizes a wider range of expertise, ultimately improving the quality of assessments. The platform's emphasis on real-world testing and interaction with the crypto community distinguishes it as a reliable choice for prominent projects seeking comprehensive security solutions, such as OpenSea[4], the world's first and largest web3 marketplace for NFTs and crypto collectibles; Panoptic[5], which focuses on decentralized options trading; and ENS[6], the Ethereum-based decentralized naming protocol.

Moreover, Code4rena's audit reports provide a separate section for gas optimization, making it convenient for us to collect reliable gas optimization data. For instance, the audit report for project Zksync[7] reveals 51 findings related to gas optimizations, with each finding typically linked to a specific function within a smart contract. Fig. 1 illustrates one of the findings from this audit report. In this example, the Warden suggests optimizing the contract by using the == operator instead of != and by reverting earlier when certain conditions are met. This optimization aims to reduce unnecessary gas consumption by avoiding redundant state variable checks.

Additionally, Code4rena utilizes a static analysis tool named `4naly3er` [20] to generate audit reports automatically. The results generated by `4naly3er` are stored directly in the corresponding Code4rena project repositories. Importantly, the results from `4naly3er` and the community auditors' findings do not overlap. The automated tool focuses on static code analysis to flag optimizations and vulnerabilities, while community auditors provide in-depth, context-aware evaluations, often factoring in broader contract behavior and interactions.

In summary, in this study, we choose to use data from Code4rena to study gas optimization because it provides reliable data from wardens, supplemented by automated data from static analysis.

III. Gas Optimization Pattern

In this section, we will describe our process of collecting data, as well as the classification and analysis of gas optimization patterns.

A. Data Collection

We first used a web crawler to collect all 209 audit reports from Code4rena and gather their corresponding automated reports generated by `4naly3er`, covering the period from October 31, 2022, to October 20, 2024. We then filtered out the audit reports that do not include a section on gas optimization, resulting in 54 audit reports from various projects. Lastly, we extracted instances related to gas optimization from each report. To ensure the accuracy and relevance of our dataset, we filtered out false positive cases from the automated report. Instances where incorrect optimization suggestions were identified were discarded, all of which originated from the automated reports. To enhance the reliability of our dataset, we implemented a manual verification process involving two experienced annotators. Each annotator independently reviewed the automated reports to identify false positives. Discrepancies between the annotators' assessments were resolved through discussion, ensuring a consensus on the validity of each identified gas-inefficient pattern. Incorporating human expertise in the verification process addresses the limitations of static analysis tools, particularly their inability to fully comprehend contextual nuances. Table I shows data from sampled repositories, highlighting the number of gas-inefficient patterns reported by wardens and automated tools, along with the associated false positive counts. Additionally, when similar gas optimization issues were found across multiple functions within the same contract, only one representative instance was retained to avoid redundancy.

Let's take a look at RevertLend as an example. In this repository, we identified 15 instances of gas-inefficient patterns from the warden report and 36 from the automated report produced by `4naly3er`. Among these, 11 instances from the automated report turned out to be false positives, resulting in 40 confirmed cases of gas inefficiencies in RevertLend. In one automated report example shown in Fig. 2, the recommendation was to optimize the self-increment statement code in Line 8 and Line 9. The suggested optimization was based on the idea that `a=a+b` is generally more gas-efficient than `a+=b` when dealing with state variables (excluding arrays and mappings). However, upon closer inspection, this recommendation turned out to be a false positive. In the provided code, `fees0` and `fees1` are not state

[1]https://solodit.xyz/
[2]https://github.com/xxxeyJ/Awesome-Blockchain-Security
[3]https://audit-hero.com/finding
[4]https://opensea.io/
[5]https://panoptic.xyz/
[6]https://ens.domains/
[7]https://code4rena.com/reports/2024-03-zksync

TABLE I
THE NUMBER OF GAS INEFFICIENT FINDINGS FROM WARDEN REPORTS AND AUTOMATED REPORTS (INCLUDING FALSE POSITIVES)

Repository	#Findings in Warden Report	#Findings in Automated Reports	#FP Findings in Automated Reports
Axelar	16	33	11
Revertlend	15	36	11
Wenwin Contest	10	20	7
AiArena	19	44	18
TimeSwap	20	31	12
DripsProtocol	13	32	6
Llama	21	31	10
Spectra	12	20	6
LivePeer	14	28	12
Zksync	51	65	15
Others	655	909	131
Total	**846**	**1,249**	**239**

```
1  function _getAmounts(PositionState memory state)
2      internal
3      view
4      returns (uint256 amount0, uint256 amount1,
           uint128 fees0, uint128 fees1)
5  {
6      ....
7      (fees0, fees1) = _getUncollectedFees(state,
           state.tick);
8      fees0 += state.tokensOwed0;
9      fees1 += state.tokensOwed1;
10 }
```

Fig. 2. An example of a false positive from an automated report: the tool suggested optimizing the self-increment statements (Lines 8-9) assuming `fees0` and `fees1` are state variables, but they are actually local variables. Applying the suggested optimization would not improve gas efficiency, illustrating the tool's misclassification.

variables; rather, they are local variables defined within the `_getAmounts` function. Therefore, applying this optimization would not affect gas efficiency in this particular context, making the suggestion inaccurate. The automated report incorrectly flagged this instance due to a misunderstanding of the relevant variables.

We finally obtained a total of 2,095 gas optimization instances, each including relevant optimization suggestions and code locations. This dataset helps us analyze prevalent techniques and identify effective strategies for reducing gas consumption in smart contracts, offering valuable insights for developers aiming to improve efficiency.

B. Data Classification

For the 2,095 gas optimization instances collected, two authors conducted a thematic analysis using an open card sorting approach [24] to analyze the code changes and reasons behind them. The analysis primarily involved reviewing the recommendations and corresponding source code for validation. Each category was derived based on common optimization patterns, including optimizations related to contract deployment, compilation, and storage. Additionally, assembly tricks and deprecated optimizations, though less frequent, were identified to provide a deeper understanding of gas optimization practices.

The categorization process followed these steps:

1) The authors carefully reviewed both the recommendations and corresponding code changes to fully understand the context of each gas optimization.
2) They generated descriptive phrases explaining how the recommendations were formulated and the reasons behind them.
3) Recommendations with similar meanings were aggregated into clusters, with category names assigned to each group.
4) The authors then refined the categorization by merging semantically similar categories or structuring them as subcategories under broader themes.

After categorizing the gas optimization findings, the authors reviewed and assigned categories to each instance, resolving ambiguities through discussion until they reached a consensus. During the process, the two authors independently labeled the optimization instances and synchronized their labeling decisions biweekly to align their understanding and update the category schema as necessary. To assess the reliability of their independent classifications, we calculated Cohen's Kappa on a subset of the annotated data, which yielded a value of 0.87, indicating a high level of inter-rater agreement and demonstrating the robustness and reproducibility of the classification scheme.

Table II presents various gas optimization categories that appear more than 15 times, along with their descriptions and the associated detection levels. examples of each category can be found in the dataset package. The detection level column indicates the scope at which each optimization can be detected, such as at the function, variable, condition statement, operation, or loop statement level. The already reported column denotes whether the pattern was already reported in prior studies. To determine this, we conducted a systematic literature review of gas optimization techniques published in the past five years, focusing on top software engineering venues (ICSE, FSE, ISSTA, ASE, Etc.). In total, we collected 22 papers related to gas optimization, of which 14 directly addressed source-code-level optimizations. Two authors independently reviewed and cross-checked these papers to identify recurring patterns, ensuring the reliability of the extracted categories. While some of the identified patterns overlap with prior work, which was largely summarized based on expert experience or analytical reasoning, our study is fundamentally grounded in real auditing practice. As our dataset is derived from real-world auditing outputs, this overlap also serves as an empirical validation that these patterns are repeatedly observed in practice rather than merely hypothesized. more importantly, our analysis further reveals ten previously unreported gas-inefficiency patterns.

C. Gas Optimization Categories

1) Save Gas On Deployment: Deployment gas costs are the fees required to initially deploy a smart contract on the Ethereum blockchain. Although typically a one-time expense, these costs can be substantial for complex contracts.

Mark revert functions payable. Marking functions as payable can reduce gas costs by eliminating redundant Ether transfer checks. This optimization is particularly useful

TABLE II
GAS OPTIMIZATION CATEGORIES

	Categories	Description	Detection Level	Already Reported
Save Gas On Deployment	Mark revert functions payable	Functions guaranteed to revert when called by normal users can be marked payable.	function	Yes [20, 22]
	Use custom errors	Use custom errors instead of revert strings to save gas.	condition statement	No
	Remove unused variables/functions	Avoid declaring unused variables or unused internal function.	function, variable	Yes [12]
Save Gas On Compilation	Split require/if statements	Splitting require/if statements that use && saves gas.	condition statement	No
	Use unchecked blocks	Using unchecked can reduce gas cost by skipping overflow and underflow checks.	logical	Yes [20]
	Use pre-increment	++i costs less gas compared to i++ or i += 1 (same for –i vs i– or i -= 1).	operation	Yes [20]
	Remove unnecessary casting/expressions	Certain return values or expressions are not required.	logical	No
	Use private for constants	Using private rather than public for constants to saves gas.	variable	Yes [20]
	Use bitwise shifts	Use shift right/left instead of division/multiplication if possible.	operation	Yes [20]
	Reorder comparison statements	Reordering conditions in functions to make cheaper check first.	condition statement	No
	Use = over +=	a = a + b is more gas effective than a += b for state variables.	operation	No
	Cache array length pre-loop	Cache array length outside of loop.	loop statement	Yes [20]
	Emit\require before state updates	Moving emit statements can save gas by reducing temporary variable usage.	condition statement	No
	Repetitive code optimization	Consolidate repetitive code into loops to save gas and enhance readability.	logical	No
Save Gas On Storage	Use calldata for immutable arguments	Use calldata instead of memory for function arguments that do not get mutated.	variable	Yes [22]
	Cache re-reading state variables	State variables should be cached in stack variables rather than re-reading them from storage.	variable	Yes [22]
	Declare immutable for efficiency	State variables only set in the constructor should be declared immutable.	variable	Yes [22]
	Optimize variable storage layout	Pack the variables into fewer storage slots by re-ordering the variables or reducing their sizes.	variable	Yes [22]
	Skip single-use local variables	Do not declare local variables used only once.	variable	No
	Check value before state update	check before updating state variable with same value.	variable	No
	Optimize bool variables	Using bool data type for storage in Solidity can incur overhead.	variable	Yes [20]
	Cache state variables before loops	Cache state variables outside of loop.	variable, loop statement	Yes [12], [23]
Others	Assembly tricks	Use inline assembly for some arithmetic operations	operation	Yes [20]
	Outdated skill	Use != 0 instead of > 0 for unsigned integer comparison	operation	No

for functions meant to receive Ether, especially within access-controlled contracts. However, it requires careful implementation to avoid potential vulnerabilities, such as unauthorized transfers.

`Use custom errors.` In Solidity, custom errors defined with the error statement are more gas-efficient than traditional require statements. They offer improved exception handling by reducing both deployment and execution costs, thanks to ABI encoding, which stores and retrieves errors more efficiently.

`Remove unused variables/functions.` In Solidity, avoiding unused variables and internal functions is essential for reducing gas costs and keeping code clean. Unused elements add unnecessary bytecode, increasing deployment costs and reducing execution efficiency.

2) Save Gas On Compilation: The following tricks are known to improve gas efficiency in the Solidity compiler.

`Splitting require/if statements.` Splitting `require()/if` statements with the `&&` operator can save around three gas per execution by minimizing redundant evaluations, though it does slightly increase deployment costs. This optimization is particularly valuable for functions that are frequently called, enhancing contract efficiency.

`Using unchecked blocks.` Using `unchecked` blocks in Solidity allows developers to bypass default overflow and underflow checks, saving gas in cases where overflow is unlikely. This optimization is particularly useful in scenarios such as:

- Loops with natural upper bounds, where iteration limits prevent overflow.
- Mathematical operations with pre-sanitized inputs, ensuring values stay within safe ranges.
- Incremental counters that increase by a small amount with each transaction, making overflow highly improbable.

For each arithmetic operation, it is essential to confirm that overflow is naturally prevented by the code's context. If so, unchecked can reduce gas usage. Properly assessing the usage of unchecked requires analyzing the context and intent behind each arithmetic operation, a nuanced evaluation that automated tools may lack.

`Use pre-increment.` The gas difference between `i++` and `++i` comes from stack handling: `i++` stores the current value before incrementing, leaving two values on the stack, whereas `++i` increments immediately and stores only the final value.

`Remove unnecessary casting/expression.` Avoiding unnecessary assignments and casting in Solidity can save gas by eliminating extra computational steps. When certain return values are not needed, omitting them enhances efficiency, especially in functions with frequent operations. This approach streamlines contracts and reduces gas costs.

`Using private for constants.` Using private visibility for constants in Solidity can optimize gas usage by avoiding non-payable getter functions, minimizing storage costs, and reducing unnecessary entries in the method ID table. When external access to these constants is needed, an efficient custom external function can be used instead.

`Use bitwise shifts.` In the Ethereum Virtual Machine (EVM), bit-shifting operations like shr and shl cost only 3 gas, compared to multiplication and division (mul and div), which each cost 5 gas. Bit-shifting avoids overflow, underflow, and division checks, but it must be used carefully to prevent errors.

Reorder comparison statement. To save gas in Solidity, move conditional checks that could revert a transaction to the start of a function, avoiding unnecessary gas usage on state reads and other costly operations. Additionally, replacing inequality checks (!=) with equality checks (==) can save gas, as equality checks are optimized by the EVM.

Detecting this optimization through static analysis tools, like Slither [25], is challenging because such tools typically focus on syntactic patterns rather than the logical order of checks, making it difficult to assess whether switching comparison types or order is optimal in a given context.

Use = over +=. In Solidity, using the assignment operator (=) instead of the plus-equals operator (+=) when updating state variables can save gas by reducing the number of SLOAD operations, minimizing access to the contract's state. However, this optimization must be balanced with considerations for code readability and maintainability.

Cache array length pre-loop. Caching an array's length outside a loop in Solidity optimizes gas usage by avoiding repeated reads, reducing SLOAD or MLOAD operations in each iteration. This practice enhances performance in array-based loops, making it a recommended strategy for gas-efficient smart contract development.

Require/revert/return before state update. In Solidity, placing the require statement before updating a state variable can save gas by eliminating the need for an extra temporary variable. This optimization reduces the gas overhead of unnecessary storage operations, making contract execution more efficient while ensuring that event emissions are correctly handled.

Repetitive code optimization. Optimizing repetitive if statements in Solidity by consolidating them into loops reduces gas costs and enhances readability, improving overall efficiency without altering functionality. Static analysis tools often struggle to detect this pattern, as they're designed to spot straightforward anti-patterns or syntax-based optimizations. High-level refactoring, such as looping instead of duplicating logic, is harder for static analyzers to identify because it involves understanding broader code structure and intent, rather than simple rule matching.

3) Save Gas On Storage: The following tricks are known to improve gas efficiency in the Solidity storage.

Use calldata for immutable arguments. Accessing function inputs directly from calldata in Solidity is more gas-efficient than loading data from memory, as calldata incurs fewer operations and has lower gas costs. Memory should only be used when data modifications are necessary, since calldata is immutable.

Cache re-reading state variables. In Solidity, caching storage variables is a common gas-saving strategy since reading from storage incurs high gas costs (over 100 gas per read) and writes are even more expensive. By caching values in memory, contracts can minimize costly storage reads and writes, enhancing both gas efficiency and performance.

Declare immutable for efficiency. In Solidity, marking variables as constant or immutable is a gas-efficient practice since these values are embedded directly into the contract's bytecode, bypassing costly storage reads. This optimization conserves gas without affecting functionality and is frequently flagged by automated tools.

Optimize variable storage layout. In Solidity, arranging storage variables strategically can yield notable gas savings. By reordering or resizing variables, you can pack multiple variables into a single storage slot, minimizing the number of expensive storage operations required per transaction. For instance, using a `uint48` for timestamps or a similar minimal type for block numbers can support their respective ranges without excess gas usage.

This pattern can be challenging for static analysis tools to detect, as the tools would need to analyze complex variable interactions, type usage, and byte-level storage layouts dynamically. Identifying such opportunities requires contextual insights into the data lifecycle and layout—a level of sophistication not easily covered by traditional static analysis.

Skip single-use local variables. To save gas, avoid declaring local variables that are used only once, as accessing data directly from storage eliminates unnecessary memory copying. This approach reduces memory allocations, cutting gas costs and improving execution efficiency.

Check value before state update. To save gas in Solidity, avoid writing to storage when the new value is the same as the existing one. Checking for differences before updating prevents unnecessary storage operations, enhancing contract efficiency, especially in functions that perform frequent updates.

Optimize bool variables. Using bool data type for storage in Solidity can incur unnecessary gas costs, especially when changing states from false to true after having been true in the past. To mitigate these gas costs, you can utilize uint256 values uint256(1) and uint256(2) to represent true and false respectively.

Cache state variables before loops. Caching state variables outside loops in Solidity can reduce gas costs by avoiding repeated storage reads, optimizing access operations and improving loop efficiency, particularly in functions with high iteration counts or complex calculations.

4) Assembly Tricks: Using inline assembly in Solidity can reduce gas by enabling direct memory manipulation and minimizing opcodes, which bypasses the Solidity compiler's default handling. Inline assembly is particularly effective in scenarios like authentication (e.g., zero-address checks) and optimizing event emissions with multiple data arguments. However, careful consideration is essential to balance readability, security, and optimization benefits. This pattern is mostly flagged in expert reports, with some instances also identified by automated tools.

5) Outdated Skill: In Solidity versions around 0.8.12, the optimization benefit of using $!= 0$ over > 0 for unsigned integer comparisons no longer applies. While this distinction was once relevant for gas savings, updates to Solidity have made both approaches effectively equivalent. If you are working with an older Solidity version, benchmarking may still reveal minor differences. In this study, all 123 instances were flagged by the automated tool; however, 95 of these were false positives due to

TABLE III
THE NUMBER OF GAS INEFFICIENCY INSTANCES FROM WARDEN REPORTS AND AUTOMATED REPORTS FOR EACH CATEGORY

Categories	#Instances in Automated Report	#Instances in Warden Report (FN)	#Instances in Automated Report (FP)	FN Reason	FP reason
Mark revert functions payable	162	0	12	-	LCA
Use custom errors	104	20	0	TVL	-
Remove unused variables/functions	32	9	25	CUD	LCA
Split require/if statements	21	31	0	CCG	-
Use unchecked blocks	82	69	0	CUD	-
Use pre-increment	104	12	6	TVL	IDSP
Remove unnecessary casting/expressions	0	43	0	CCG	-
Use private for constants	76	8	0	TVL	-
Use bitwise shifts	40	6	13	TVL	IDSP
Reorder comparison statements	0	50	0	CCG	-
Use = over +=	82	21	46	TVL	IDSP
Cache array length pre-loop	118	0	0	-	-
Emit\require before state updates	0	23	0	CCG	-
Repetitive code optimization	0	15	0	CCG	-
Use calldata for immutable arguments	49	50	0	TVL	-
Cache re-reading state variables	57	148	27	CUD	IDSP
Declare immutable for efficiency	17	21	0	TVL	-
Optimize variable storage layout	0	57	0	CCG	-
Skip single-use local variables	0	38	0	CCG	-
Check value before state update	28	19	0	TVL	-
Optimize bool variables	92	8	0	TVL	-
Cache state variables before loops	0	23	0	CCG	-
Assembly tricks	56	42	0	CCG	-
Outdated skill	123	0	95	-	LCA

the tool's lack of recognition of the Solidity version or variable type constraints.

D. Limitations of Static Analysis Tool in Code4rena

During the categorization process of gas-inefficient instances, we identified several limitations in the automated reports generated by `4naly3er`. Table III shows the number of gas-inefficient instances identified by wardens and `4naly3er`. The instances detected by wardens represent false negatives (FN) for `4naly3er`. Additionally, we report the number of false positives (FP) generated by `4naly3er`. We also analyze the reasons behind these FNs and FPs, which are outlined below:

`Causes of False Negative.` Our comparative analysis of Warden's audit reports and the automated static analysis tool (`4naly3er`) in Code4rena identifies three primary causes of false negatives in gas optimization detection:

- *Category Coverage Gaps (CCG)* arise when the static analysis tool either (1) lacks support for entire optimization categories or (2) implements incomplete pattern recognition. Categories including *Reorder comparison statements*, *Split require/if statements* and *Repetitive code optimization* fall under the first reason, as the static analysis tool lacks built-in support to detect these optimization opportunities entirely. For the second reason, incomplete pattern recognition, the static analysis tool detects certain instances within a category but fails to generalize across similar patterns. In the *Split require/if statements* optimization, `4naly3er` successfully identify duplicated `require` statements using `&&` operators but fail to detect equivalent `if` statement patterns, despite their identical gas-saving potential.
- *Tool Version Limitations (TVL)* emerges when analyzing historical contracts (2022–2024) with outdated tool versions. Modern Solidity optimizations like custom error handling, `pre-increment` operators, `calldata` usage for immutable arguments, and boolean packing remain undetected by legacy tools, as evidenced by Warden's identification of seven undocumented categories in automated reports.
- *Contextual Understanding Deficits (CUD)* refers to the inability of automated tools to effectively model and interpret the complex relationships between variables across different conditions. This is critically apparent in category *Use unchecked blocks*, where About 45% of the instances of this pattern are detected in Warden's report. Properly assessing the usage of unchecked requires analyzing the context and intent behind each arithmetic operation, a nuanced evaluation that automated tools may lack. The example in Fig. 3 demonstrates a gas optimization using the `unchecked` block. However, the static tool fails to detect it because it does not understand the condition `if (_oldPTRate == _ptRate && _ibtRate > _oldIBTRate)`, which guarantees that the subtraction `_ibtRate - _oldIBTRate` is safe. The tool's line-by-line analysis approach overlooks the interaction between these variables across multiple conditions, leading to missed optimization opportunities.

`Causes of False Positives`. When analyzing the automated tool report, we observe numerous false positives across

```
function _computeYield(
    address _user,
    uint256 _userYieldIBT,
    uint256 _oldIBTRate,
    uint256 _ibtRate,
    uint256 _oldPTRate,
    uint256 _ptRate,
    address _yt
) external view returns (uint256) {
    if (_oldPTRate == _ptRate && _ibtRate >
        _oldIBTRate) {
-        newYieldInIBTRay = ibtOfPTInRay.mulDiv(
    _ibtRate - _oldIBTRate, _ibtRate);
+        unchecked {
+            uint256 ibtRateMinusOldIbtRate =
    _ibtRate - _oldIBTRate;
+        }
+        newYieldInIBTRay = ibtOfPTInRay.mulDiv(
    ibtRateMinusOldIbtRate, _ibtRate);
    } else {
        ......
    }
}
```

Fig. 3. An example from Warden's audit report demonstrating the use of `unchecked` blocks for gas optimization. The automated static tool fails to recognize that the comparison `_ibtRate > _oldIBTRate` ensures the safety of the subtraction operation, allowing the use of unchecked for gas savings without risking overflow.

various categories. These inaccuracies are largely due to two key factors:

- *Lack of Context Awareness (LCA)* arises when automated tools overlook important context, leading to false positives. In the category *Remove unused variables/functions*, automated tools frequently flag internal functions as unused, resulting in numerous false positives. Internal functions in a contract are not only visible within the current contract but also to derived contracts. This visibility allows these functions to be invoked in inherited contracts, even if they appear unused locally. The tool fails to account for this context, leading to misreporting of these functions as unused.
  Additionally, in Section III-C5, we observe that in Solidity versions around 0.8.12, the optimization benefit of using $!=0$ over >0 for unsigned integer comparisons no longer applies. Older versions of static analysis tools may still flag $!=0$ as the more gas-efficient approach, ignoring the changes introduced in Solidity optimization. As a result, false positives occur because the tool fails to recognize the updated behavior and constraints of newer Solidity versions.
- *Inaccurate Detection of Specific Patterns (IDSP)* occurs when the tool misidentifies patterns, resulting in false positives. In the *Use Pre-increment* category, false positives arise because the modification may alter assignment behavior. In the *Use Bitwise Shifts* category, false positives occur due to a misidentified optimization opportunity. In the *Use* $=$ *over* $+=$ and *Cache Re-reading State Variables* categories, the tool reports 46 and 27 instances, respectively, where these rules are incorrectly applied to non-state variables or cases involving arrays or mappings. In such cases, the distinctions between $=$ and $+=$, as well as the need to cache state variables, do not apply, but the tool fails to account for these exceptions, leading to false positives.

Challenges of static analysis. Additionally, based on the analysis of FNs and FPs in automated reports, we summarize the reasons why static analysis tools struggle to accurately identify gas optimizations.

- *Contextual Relationship (CR):* Static analysis tools often struggle with categories like *unchecked blocks*, *repetitive code*, and *require/revert/return before state updates* due to limited contextual awareness. For example, detecting unchecked blocks requires understanding each operation's context for safety, while identifying repetitive code depends on distinguishing intentional repetition from redundancy. Similarly, categories like *require/revert/return before state updates* depend on the contract's logical flow, which static tools may misinterpret. Additionally, static analysis tools face challenges in detecting category *cache re-reading of state variables*, often misidentifying necessary operations as inefficiencies due to the lack of context on state changes.
- *Semantic Understanding (SU):* Categories such as *assembly tricks*, *reordering comparison statements*, and *optimizing variable storage layout* are also difficult for static analysis due to the tools' limited semantic understanding. Assembly code, for instance, often requires context-specific optimizations that static tools fail to recognize. Reordering comparisons and optimizing variable storage layout rely on the logical sequence of operations, which static tools may overlook. Moreover, recognizing pre-increment and post-increment optimizations or eliminating unnecessary casting and expressions requires a nuanced understanding of code behavior, which static analysis tools typically miss.

IV. Approach

In this section, we describe our approach in detail.

A. Overview

The empirical findings in Section III-C indicate that gas optimization opportunities in smart contracts exist at various levels, including functions, variables, and condition/loop statements. However, we also observe that relying solely on static analysis for detecting gas inefficiencies leads to numerous false positives and negatives. To address this limitation, we employ a static analysis tool to extract relevant code elements and leverage a large language model (LLM) to identify optimization opportunities and repair them automatically. This decision is driven by the demonstrated effectiveness of LLMs in tasks related to code analysis and generation [21].

Fig. 4 presents the workflow of our approach, consisting of two steps. First, we use Slither to analyze a smart contract repository, extracting state variables and traversing function expressions to identify potential gas optimization opportunities based on our empirical findings, while also isolating each function's source code. Second, we combine the function code with its identified optimization opportunities and employ an LLM to accurately detect and repair these inefficiencies.

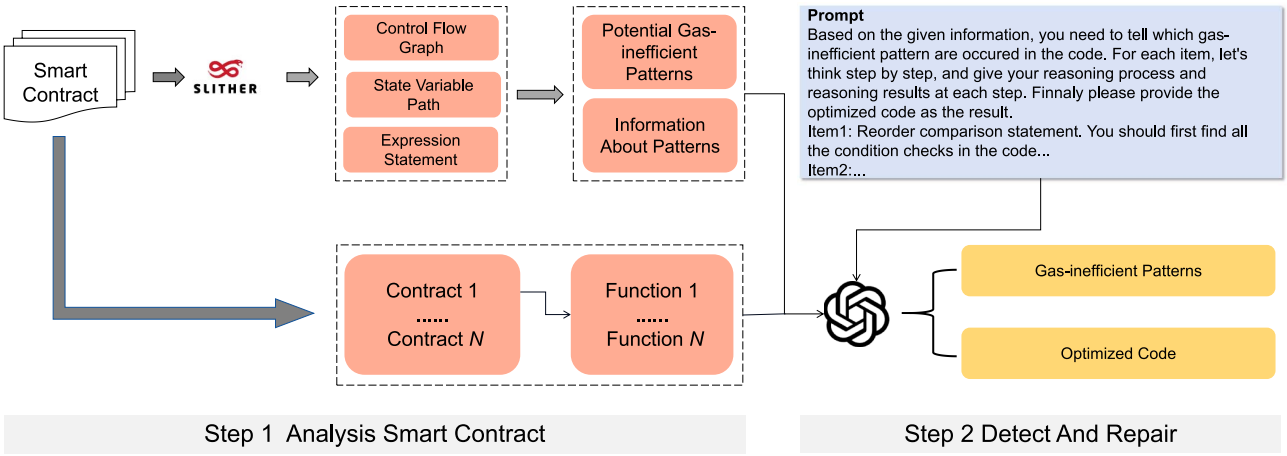

Fig. 4. Workflow of our approach.

Before executing our method, we perform a pre-processing step. First, we identify the Solidity version used in the repository. Next, we adapt the repository to its specific framework to ensure successful compilation, which is essential for Slither's analysis. After that, we extract the function's source code based on the contract and function names within the Solidity files. To prevent comments from influencing LLM's results in subsequent steps, we remove them before applying regular expressions to locate and extract the function's source code. We then proceed to explain our approach in detail.

B. Step 1: Analysis Smart Contract

Our approach utilizes static analysis tools to detect gas optimization opportunities in smart contracts. We analyze state variables, extract function expressions, and traverse control flow to identify potential optimizations, associating relevant information with each expression node for detection.

We utilize Slither as our static analysis tool instead of `4naly3er`, as `4naly3er` only provides gas optimization opportunities with corresponding code lines and lacks intermediate results. As a widely recognized tool, Slither enables efficient retrieval of all contract variables and their declaration statements. It also facilitates the processing of a function's control flow graph (CFG), allowing us to traverse nodes, identify modifiers and conditional statements, and determine if expressions involve variables being read or written.

We identify various categories of gas inefficiency and examined their detection levels within the source code (see Table II), such as variables, conditional statements, loop statements, and more. As a result, we use the static analysis tool to extract relevant information from these levels, including:

- **Variable Access Tracking.** In the sections outside the smart contract functions, the contract declares relevant state variables and defines useful structures. To facilitate analysis, we extract the types of these variables along with their assignment statements. For the category of *optimizing the variable storage layout*, we go a step further by including the extracted information and preserving the comments associated with each variable. These comments assist the LLM in making more accurate judgments regarding the reduction or reordering of variable sizes to improve storage efficiency.
  Inside the function, the contract declares local variables and performs read and write operations on state variables, parameters, and local variables. we analyze the usage of all variables, including these variables. By scanning the function line by line, we track the declaration, reading, and writing of each variable, recording their access frequency. Writing to state variables is flagged to detect potential *check value before state update* patterns, ensuring necessary conditions are verified beforehand. Similarly, repeated reads of state variables are monitored to identify *caching re-reading of state variables*, where storing a value in memory could save gas.
- **Control Flow Statements.** Key control flow statements such as `require`, `if`, and `assert` govern the execution logic of smart contracts. When encountering a control flow statement that may halt execution (e.g., `require` or `if-revert`), we examine whether it precedes a state update to detect *require/revert/return before state updates* patterns. Statements containing `&&` operators are analyzed to identify opportunities for splitting them into sequential checks for clarity and efficiency. Additionally, functions with multiple control flow statements are reviewed for *reordering comparison statements* category, where cheaper or earlier checks can be prioritized to reduce execution costs.
- **Loop Statements.** Loops are a common source of gas inefficiencies. To address this, we integrate Slither's control flow graph (CFG) with expression nodes, combining CFG's structural execution insights with the granular variable access details from expression nodes. This hybrid analysis allows us to detect categories like re-reading state variables inside loops, where caching the values outside the loop would reduce redundant storage access and save gas.

- **Operation.** Beyond the above categories, we also identify additional categories. For *using pre-increment(pre-decrement)* category, when encountering a statement with self-increment or self-decrement, it can be potentially replaced by pre-increment(decremant) while the function execution is not influenced. What's more, we carefully examine the code for the presence of the keyword `pragma` to specify the Solidity version, and identify some outdated gas optimization opportunities, for example consider *using != 0 over > 0 for unsigned integer comparisons* when Solidity version is 0.8.12 or below.
- **Other Situations.** Within the contract functions, modifiers are important to control function behavior and enforce conditions before execution. We detect categories *Mark revert function payable* by analyze the access control of the function. Categories including *Use unchecked blocks* and *Remove unneccessary casting/expressions* involve logical relation between each expression. We directly suppose the potential gas optimization opportunities refer to these categories with the help of the LLM's reasoning ability.

Finally, through analyze the smart contract, we output a comprehensive list of potential gas optimization opportunities detected within each function along with some information about these opportunities and extract the source code of each function. These identified gas optimization opportunities serve as a foundation for further analysis, allowing the large language model to perform more specialized detection and targeted gas optimizations. By flagging possible inefficiencies and providing contextual insights, this process ensures that the large language model can focus on resolving gas-related concerns with greater precision and accuracy, leading to improved performance and cost efficiency in the smart contracts.

C. Step2: Detect And Repair

In this step, we leverage GPT-4o to assist in accurately detecting and repairing gas-inefficient patterns. However, due to context limitations inherent in GPT-4o, we enhance its analysis by packaging the source code of the function alongside the potential gas optimization opportunities identified in Step 1. This data, along with related information, is structured to support the large language model's detection capabilities. The relevant information includes condition statements, variable declaration statements (for those read and written by the function), the Solidity version of the smart contract and other additional details. This supplementary data helps the large language model better detect and analyze these patterns.

To further improve the reasoning process, we utilize Chain of Thought (CoT), a technique that enhances the reasoning abilities of large language models. Cot prompting involves providing a sequence of intermediate reasoning steps within the input prompt, guiding the large language model to logically decompose a task step-by-step before arriving at a final conclusion. Early research [26] demonstrated that incorporating a chain-of-thought prompting sequence, which outlines intermediate reasoning steps, could significantly enhance the performance of LLMs in complex reasoning tasks. Subsequent studies further validate the effectiveness of CoT prompting [27], [28]. By prompting the large language model to break down complex tasks into intermediate steps, CoT facilitates a more structured and accurate approach to problem-solving.

In our methodology for detecting gas inefficiencies in smart contracts, we apply CoT prompting to increase detection accuracy. For each identified inefficiency within a function, we craft a specific CoT prompt to guide the large language model through a structured analysis. The prompt explicitly presents potential optimization opportunities, guiding the model to focus on likely inefficiencies, and structures the analysis into step-by-step reasoning so that the model can systematically evaluate each issue and prioritize cost-effective improvements. By requesting the reasoning process alongside the optimized code, the model's decisions become interpretable and verifiable, ensuring that the suggested optimizations are practical and reliable. After presenting the potential issues, we prompt the large language model with the phrase, "*For each item above, let's think step by step*", encouraging it to systematically evaluate each potential inefficiency.

To assess the impact of CoT prompting on detection accuracy, we conducted an ablation study comparing three variants of our prompt design: (1) without using any CoT strategy, (2) using CoT but omitting the large language model's reasoning output, and (3) our full CoT-based prompt that includes both reasoning steps and conclusions. Detection accuracies were 42.76%, 53.85%, and 85.44% for the three variants, respectively, demonstrating the substantial benefit of including both reasoning steps and conclusions in the prompt. These findings confirm that explicitly prompting the large language model to reason step-by-step and allowing it to output intermediate thinking plays a crucial role in improving detection performance.

Fig. 5 shows an example of prompt applied to the code shown in Fig. 1. In this prompt, we direct the LLM to first review all condition checks in the code, then identify checks at the same logical tier. The large language model is then instructed to prioritize ranking the checks with the least cost or earliest exit conditions in the same logical tier. After the reasoning and thinking of the large language model, it will output the completion. In the completion, after reasoning through the optimization process, the large language model provides a detailed explanation for each decision and outputs the optimized code reflecting the identified improvements.

V. Experiment

Through the experimental evaluation, we aim to answer the following research questions (RQs):

- **RQ1:** How effective is our approach in detecting gas-inefficient patterns?
- **RQ2:** How effective is our approach in repairing gas-inefficient patterns?
- **RQ3:** What is the potential gas savings for smart contract invocations achieved through our approach?

A. Experiment Setup

`Implementation.` We implement our approach in Python and obtain the control flow graph of the functions on the Solidity

Prompt

Now, you are a professional researcher of smart contracts. Please analyze the following smart contract and identify any gas optimization opportunity, specifically:

- *Preserve the original function exactly as it is, without addingany modifiers or making alterations to it.*
- *No external dependencies or imports are introduced.*

Here are some potential gas optimization opportunities inside the code.

- *Reorder Comparison Statement. Here are the condition statement in the code:[CONDITION STATEMENTS].*
  - You should first identify them in the same logical tier.
  - Then, you should prioritize rank the checks with less cost or earlier exit condition in the same logical tier.
  - If the order of condition checks in the code is not the same with your rank, the repair the gas inefficient situation based on your rank.
- ...

Output the reasoning process of each item, and return the optimized code as the result. The code is as follows: Solidity version:0.8.0 [CODE]

Completion

Here is the reasoning process of each item:

- *We separate the condition statements by the logic tier. For the function logic tier, we ranked block.chainid == ERA_CHAIN_ID check ahead of the block.chainid != ERA_CHAIN_ID check because of earlier exit of block.chainid == ERA_CHAIN_ID check ...*
- ...

Here the optimized code based on the analysis above.

```
solidity
//SPDX-License-Identifier:MIT
pragma solidity 0.8.0:
...
```

Fig. 5. Prompt template.

compiler with the compilation commands `slither folder print cfg`.

Dataset. We selected 10 Code4rena repos issued between March 2024 and June 2024. These repos are under different frameworks including foundry, hardhat and solc, with the version ranged from 0.4.0 to 0.8.0. Through a careful screening process, we isolated 316 instances where gas-inefficient patterns were identified referring to more than 150 functions. Each identified issue was carefully analyzed, and following the audit's recommendations, appropriate adjustments were implemented. The final dataset includes both the original contract functions and their repaired counterparts, offering significant insights into gas optimization strategies and their real-world impact.

B. RQ1: The Effectiveness of Gas-inefficient Pattern Detection

In our evaluation, we assess the effectiveness of our approach in detecting gas-inefficient patterns in smart contracts by comparing it against three distinct baselines and experimenting with various LLMs.

In our experimental results, a true positive is defined as a situation where a function contains a pattern instance that requires optimization, and our detection method successfully identifies this instance. Conversely, a false positive refers to instances where a function does not have an optimization-needed pattern, but the detection method mistakenly identifies one.

We verify the precision of our method by manually inspecting the outputs generated by the large language model, checking whether the detected gas-inefficient patterns correctly correspond to those that need modification. This ensures the model's accuracy in identifying genuine inefficiencies. To verify false positives, we replace the optimized code in our dataset with the original, unoptimized version, then rerun the detection process. After that, we manually review the new results. If the optimized function incorrectly triggers a detection for the specific gas-inefficient category, which should have been fixed, we consider it a false positive instance.

LLM Selection. We test the performance of different LLM in our approach, including GPT-4o(GPT4GO), DeepSeek-R1 (DS4GO), Qwen-Plus (Qwen4GO), and Llama3.3-70B-Instruct (Llama4GO), to evaluate the performance of different models.

Our findings indicate that GPT-4o and DeepSeek-R1 outperform Llama3.3-70B-Instruct and Qwen-Plus in this task. Notably, DeepSeek-R1 excels in identifying specific categories, such as the use of unchecked blocks. However, DeepSeek-R1 occasionally provides an initial correct suggestion, followed by a reconsideration that leads to an incorrect conclusion. This behavior results in GPT-4o slightly surpassing DeepSeek-R1 in overall performance.

Baselines. The first baseline (LLM only) involves directly inputting various contract functions into an LLM, which is tasked with identifying gas-inefficient patterns based solely on the contract's structure and behavior. The second baseline (LLM + static) adopts a more structured approach by incorporating static analysis methods. This process extracts specific information regarding function behavior, such as variable reads and writes, which are key indicators of gas efficiency. Besides providing this detailed information to the LLM, the model is also better equipped with the predefined inefficiency categories (see Section III-C). The third baseline (LLM4GO - static) serves as an ablation study to assess the impact of static analysis by removing it from GPT-4o's setup. Due to the better performance among four LLMs, we directly test three baselines with GPT-4o.

We also experimented with other static analysis tools, including python-solidity-optimizer [29] and Slither [25], as well as other tools. However, some of these tools [12], [14], [15] do not provide open-source implementations, while others [22], [30] are currently non-functional. Among the tools we are able to run, python-solidity-optimizer had compatibility issues with recent Solidity versions. We addressed this by updating the Solidity syntax in the python-solidity-parser library. We finally selected Slither and python-solidity-optimizer as additional baselines for evaluation to ensure a comprehensive comparison.

Result. In the experimental results, as presented in Table IV, GPT only results in a very low recall but a relatively high precision. The reason for this is that the LLM was applied without any supplementary context, so its judgments were based solely on its surface-level interpretation of the code structure. Static-only functions primarily as an exhaustive detector that flags all potential gas-inefficient sites, which yields high recall but tends to include many false positives. This design is intentional, as the static stage aims to maximize coverage of candidate issues. However, because it lacks semantic validation of the suggested fixes, it cannot guarantee that the flagged optimizations are correct or beneficial without further reasoning or manual inspection. In practical auditing workflows, such raw static outputs would still require substantial manual triage, which is both time-consuming and difficult to scale. Therefore, the LLM in our pipeline is not intended to replace static analysis, but rather to serve as a post-filtering and re-ranking layer. By sacrificing a small amount of recall, LLM4GO achieves a significant improvement in precision, making the final results more focused and actionable for auditors. GPT + Static, which integrates static analysis with the LLM, improved the recall and showed a more balanced precision. By providing the LLM with additional information, it was able to make more informed decisions. GPT4GO - Static underperforms compared to GPT4GO due to the absence of static analysis. Without static filtering, the LLM must process a wider range of potential inefficiencies, increasing the risk of incorrect detection. Additionally, some inefficiencies rely on structural details that static analysis extracts. Without this information, the LLM struggles to detect certain patterns. Furthermore, static analysis tools such as python-solidity-optimizer and Slither showed disappointing results. Python-solidity-optimizer detected only five instances of gas-inefficient patterns, while Slither identified 17 instances, leading us to exclude them from further comparisons. These results indicate that the limitations are not unique to 4naly3er but are common among static analysis approaches. Prior studies have also highlighted similar weaknesses: static analysis tools often rely on pre-defined rules and struggle to generalize to new patterns [31]; they may also exhibit low recall and high false positive rates [32]. This further justifies the role of LLMs in our pipeline, as they can complement static techniques by providing broader coverage and reducing undetected inefficiencies.

TABLE IV
PRECISION AND RECALL OF OUR APPROACH AND TWO BASELINES ON THE DATASET

metric	Precision	Recall	F1 Score
GPT only	74.05%	6.65%	12.20%
Static only	57.45%	100.00%	72.98%
GPT + Static	61.39%	23.10%	33.57%
GPT4GO - Static	60.13%	72.47%	65.72%
GPT4GO	**82.28%**	**85.44%**	**83.83%**
DS4GO	81.01%	84.49%	82.72%
Llama4GO	72.15%	70.89%	71.51%
Qwen4GO	73.10%	82.28%	77.42%

RQ1 Summary: Integrating static analysis with LLMs enhances precision while maintaining recall. GPT-4o performs best, with DeepSeek-R1 excelling in specific cases.

C. RQ2: Effectiveness of Gas-inefficient Pattern Repair

In this section, we tasked the large language model (LLM) with attempting to repair the source code based on the detected gas-inefficient patterns. Following the detection result in RQ1, we test the effectiveness of gas-inefficient pattern repair with different baselines and different LLMs. For different methods, out of the instances detected, we conducted a thorough review of the repairs generated by the LLM.

We evaluated the correctness of the LLM-repaired code by comparing it with the optimized code in our dataset. A repair is considered exactly correct if it successfully implements all expected optimization points, producing results that align with the intended optimized version. If only some of the expected optimizations are applied and the improvement is incomplete, the repair is classified as partially repaired. Conversely, an incorrect repair is further divided into two types: functionality-preserved incorrect, where the inefficiency remains unresolved but the

TABLE V
REPAIR RESULT OF OUR METHOD AND TWO BASELINES ON THE DATASET

metric	Exactly Correct	Partially Correct	Functionality-Preserved Incorrect	Semantic-Deviation Incorrect
GPT only	80.95%	19.05%	-	-
GPT + Static	82.19%	17.81%	-	-
GPT4GO -Static	80.00%	13.68%	5.27%	1.05%
GPT4GO	**88.46%**	**10.00%**	**1.54%**	**-**
DS4GO	85.94%	11.72%	1.95%	0.39%
Llama4GO	84.21%	13.16%	1.31%	1.31%
Qwen4GO	84.85%	11.69%	3.03%	0.43%

program's semantics are preserved, and semantic-deviation incorrect, where new inefficiencies or semantic deviations are introduced.

To increase confidence in correctness, we first flattened the contracts and applied Echidna, a property-based fuzzing tool, to both the original and repaired versions. This allowed us to check whether the core invariants of the contracts were preserved under randomized testing. The results showed that none of the repaired contracts violated their key invariants, suggesting that the essential behavioral properties remained intact. To further mitigate potential risks, we carried out a manual double-check through cross-validation by two independent reviewers. Both reviewers have substantial experience in smart contract development and a deep understanding of Solidity syntax and semantics, enabling them to carefully assess the repaired code. Both reviewers independently examined the original and repaired contracts, focusing on logic, control flow, and state updates. Discrepancies (fewer than 5% of cases) were discussed jointly and resolved through consensus, ensuring a reliable assessment of functional consistency. This review confirmed that over 95% of the repaired instances retained their original functional characteristics while also delivering the intended gas-efficiency improvements. The small fraction of discrepancies mainly arose from relatively minor issues, such as subtle deviations introduced by inline assembly optimizations and adjustments to revert statement contents that slightly modified error message formatting without affecting the core logic. While such differences generally do not compromise fundamental functionality, they highlight the need for further refinement and careful review to ensure complete semantic parity.

Result. Table V presents the repair accuracy of different methods for gas-inefficient patterns. Among the tested LLMs, GPT-4o achieved the highest accuracy and the lowest error rate, while DeepSeek-R1 also shows competitive performance. Notably, all models exhibited some incorrect repairs, primarily in complex optimizations such as comparison reordering and variable storage layout, which require a more precise estimation of statement-level gas costs for effective modifications.

GPT only and GPT + Static repaired all instances exactly or partially correct, with no instance repaired incorrectly. The higher accuracy observed in these baselines may be attributed to the relatively simpler nature of the detected instances, which likely reduced the complexity of the required modifications. The effectiveness of static analysis integration is evident, as GPT + Static outperforms the GPT-only baseline by leveraging additional program structure insights, leading to more exactly correct repairs and fewer partially correct ones.

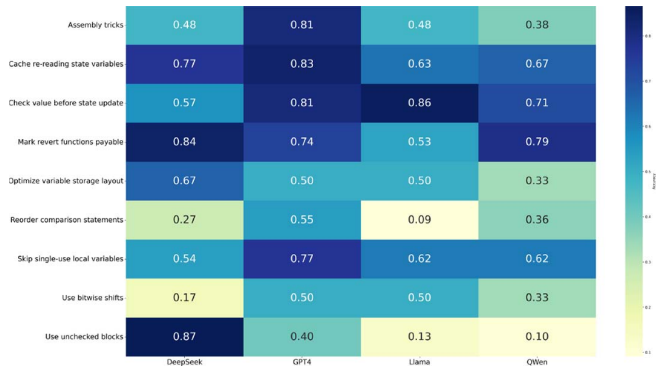

Fig. 6. Heatmap of repair accuracy for GPT4, DeepSeek, LLaMA, and QWen across different gas optimization patterns.

GPT4GO - Static shows more incorrect repairs than GPT4GO, underscoring the importance of static analysis, as the absence of structural insights leads to more errors.

Beyond improving the efficiency of the gas optimization process, the results also highlight that LLMs can enhance the overall accuracy of gas-inefficient pattern repair. By reasoning over contextual code structures and leveraging optimization knowledge, LLMs reduce the likelihood of incomplete or erroneous transformations. This demonstrates that LLMs are not only effective in automating optimization steps but also valuable in ensuring correctness and reliability in gas-efficient contract development.

As shown in Fig. 6, different LLMs exhibit varying repair performance across specific gas optimization patterns. We observe that DeepSeek achieves better repair performance than other LLMs on rules such as `Use unchecked block` and `Optimize variable storage layout`, indicating its strength in logical restructuring. However, on variable-related rules such as `Cache re-reading state variables`, `Skip single-use local variables` and `Check value before state update`, its performance lags behind other LLMs. Additionally, GPT performed better in `Reorder comparison statements`, showing its advantage in optimizing conditional logic. Overall, GPT and DeepSeek demonstrate better and more consistent repair quality, achieving strong results across most categories of optimization rules.

Case Study. Fig. 7 illustrates a "partially correct" optimization. In this case, the ExecuteParams struct was optimized by reducing the deadline data type to uint64, which was appropriate for the required range and reduced storage costs. However, all methods missed the opportunity to reorder the struct's variables to further minimize storage slots, resulting in a partial optimization.

RQ2 Summary: GPT-4o achieves the highest accuracy and lowest error rate for gas-inefficient pattern repair, outperforming other LLMs. Static analysis integration significantly improves repair precision, especially for simpler instances.

```
struct ExecuteParams {
    uint256 tokenId;
    bytes swapData;
-   uint128 liquidity;
    uint256 amountRemoveMin0;
    uint256 amountRemoveMin1;
-   uint256 deadline; // for uniswap operations -
    operator promises fair value
+   uint64 deadline;
    uint64 rewardX64;
+   uint128 liquidity;
}
```

Fig. 7. An example of "partially true" optimization, which is about the optimization of variable storage layout.

D. RQ3: Efficiency of Gas-Saving

To address Research Question 3 (RQ3), we compared the gas consumption of the source code before and after implementing our optimizations. Among the projects in our dataset, 8 were based on the Foundry framework [33], and 2 on the Hardhat framework [34]. Specifically, we used Forge, integrated within the Foundry framework, and the hardhat-gas-reporter plugin for the Hardhat framework to measure the gas savings.

Table VI presents a detailed analysis of the gas consumption for various functions within the LoopFi smart contract. the data is measured both before and after our optimizations were applied, using key metrics such as minimum (min), average (avg), and maximum (max) gas consumption across multiple function calls. The table effectively illustrates the differences in gas usage between the two scenarios, offering insight into the performance gains achieved through the optimization process. For instance, the gas consumption for the starClaimDate function remains constant, indicating no significant change. However, functions like claimAndStake and setLoopAddresses show notable improvements, with reductions in both average and maximum gas consumption. on the other hand, there are a few cases where slight gas increases were observed. These increases are likely attributed to environmental factors such as memory allocation or caching mechanisms, even though the core logic of the functions remained unchanged. This emphasizes that factors outside direct code modifications, such as memory management, can influence gas consumption.

As shown in Table VI, our optimization approach resulted in a total gas savings of 124,136,910 units across 21 different functions in LoopFi. This is a significant improvement and highlights the efficiency of the method. In other projects, our optimization technique demonstrated similar results, further showcasing its broad applicability and effectiveness. On average, for each function, our approach saved 991 gas units, which represents a meaningful reduction in gas usage per invocation.

Generalizability Study. We collect hot 100 contracts from Ethereum[8]. We filter out contracts without verified source code and remove duplicates, resulting in a dataset of 56 unique contracts deployed on Ethereum. We then apply our method to these contracts and identify a total of 790 gas-inefficient patterns. After manual and tool-assisted validation, we confirm that the majority of these findings are valid. On average, each contract contains around 14.1 optimization opportunities, and the estimated gas saving per contract execution is approximately 68,396 gas. These results demonstrate that our approach is effective and can generalize beyond the curated dataset, yielding practical savings in real-world deployed contracts.

It is worth noting that the difference in gas saving between Ethereum contracts and project-specific contracts primarily originates from the testing tools used. For project-level experiments, gas consumption is measured using frameworks such as Hardhat with detailed gas reporting, while for Ethereum contracts, estimations are obtained mainly from `solc`'s built-in gas report. This methodological difference leads to variations in the granularity and accuracy of the measurements, but does not affect the validity of the comparative analysis.

RQ3 Summary: Our optimization approach achieved significant gas reductions. On average, each function saved 991 gas units, demonstrating the efficiency and practicality of our method in real-world deployments.

VI. Discussion

A. Internal Threat To Validate

We identify two main internal threats to validity. First, our evaluation is limited by the range of large language model configurations examined. Specifically, although we use the latest available models, we did not systematically explore variations in configuration settings such as temperature, max tokens, or prompt phrasing. Since these factors can substantially influence detection precision and fix generation quality, the absence of such controlled experiments may obscure nuanced performance differences. Second, although repaired contracts were validated using property-based testing, and double-checked by two independent reviewers with strong expertise in smart contract development, these methods cannot guarantee full semantic equivalence. In particular, existing tools are not designed for comprehensive functionality verification, and subtle deviations may still remain undetected.

B. External Threats to Validity

Our framework is effective not only for contracts curated from Code4rena but also for Ethereum smart contracts in general, indicating its applicability beyond a single auditing source. Although Solidity is the most widely used language in this space, generalizing to other smart contract platforms such as Solana (Rust), Polkadot (Ink!), or Ethereum's Vyper requires additional validation. These platforms differ in language syntax, execution environments, and gas models, which may affect the framework's detection capabilities and adaptation strategies. Future work may consider extending and adapting our approach across multiple blockchain ecosystems to validate its broader applicability.

C. Fail Case Analysis

1) Reason 1: Lack of Contextual Semantic Understanding: The large language models often fail to identify opportunities

[8]https://etherscan.io/topstat#HotContracts

TABLE VI
A EXAMPLE OF FORGE RESULT APPLIED IN LOOPFI

Function Name	Before Optimization			After Optimization			Gas Saved On Average	#Calls
	min	avg	max	min	avg	max		
TIMELOCK	359	359	359	337	337	337	22	3,073
allowToken	24,052	35,135	46,219	24,008	35,204	46,401	-69	2
balances	737	737	737	715	715	715	22	3,840
claim	25,041	53,916	67,883	25,019	53,896	67,864	20	1,792
claimAndStake	24,997	76,764	121,903	24,975	75,542	119,884	1,222	1,280
convertAllETH	25,875	92,142	112,041	25,846	92,118	112,018	24	2,817
emergencyMode	362	362	362	362	362	362	0	1
isTokenAllowed	700	700	700	678	678	678	22	1
lock	24,385	70,993	86,242	24,380	70,991	86,249	2	2,049
lockETH	23,709	65,197	70,436	23,704	65,192	70,431	5	5,635
lockETHFor	23,898	47,217	70,627	23,893	47,212	70,622	5	5,635
lockFor	24,663	46,027	86,522	24,636	45,989	86,507	38	769
loopActivation	446	446	446	441	441	441	5	4,098
owner	428	428	428	384	384	384	44	2
recoverERC20	26,279	27,571	59,170	26,389	27,696	59,292	-125	513
setMergencyMode	23,824	28,841	28,848	23,780	28,797	28,804	44	770
setLoopAddresses	26,540	73,120	75,245	26,557	52,371	53,549	20,749	5,891
setOwner	23,987	26,053	28,119	23,987	26,128	28,269	-75	2
startClaimDate	442	442	442	442	442	442	0	1,793
totalSupply	385	385	385	385	385	385	0	2,560
withdraw	24,113	32,305	55,604	24,114	32,303	55,317	2	2,560
Total		1,826,217,324			1,702,080,414		124,136,910	

for gas optimization because they lack contextual semantic understanding. While they may detect common structural patterns, such as reordering statements to reduce gas usage, they do not fully capture the deeper relationships and dependencies between operations or the logical preconditions that determine the most efficient execution order. This limitation prevents the model from reasoning about which checks are cheaper, which operations depend on others, and how reordering can impact overall performance. As a result, models may miss simple yet effective optimizations or propose changes that are suboptimal in terms of gas efficiency.

In the example shown in Fig. 8, the correct gas-efficient fix is to place the InsufficientBalance check after verifying the externalToken mapping. This ordering ensures that cheaper validations are performed first, minimizing unnecessary computation and improving overall efficiency. The model failed to apply this fix because it only relied on general optimization patterns `reorder comparison statement`, without considering the contextual relationships between specific operations in the function. In this case, the dependency is that the balance check is only meaningful once the externalToken has been validated; reordering without this awareness leads to a missed optimization. This example illustrates that effective gas optimization requires reasoning about the execution dependencies and cost hierarchy of operations, rather than applying generic rules in isolation.

2) Reason 2: Pattern-Based Semantic Hallucination: Large language models may generate suboptimal or incorrect gas optimizations due to pattern-based semantic hallucination. While they can recognize common structural patterns or transformations, such as replacing multiplication with cheaper bit-shift operations, they often lack true understanding of the underlying arithmetic semantics. This can lead to transformations that appear beneficial but do not preserve the original logic, highlighting the limitation of relying solely on pattern recognition without contextual reasoning.

```
    function deposit(address curvesTokenSubject,
        uint256 amount) public {
        if (amount % 1 ether != 0) revert
            NonIntegerDepositAmount();

        uint256 tokenAmount = amount / 1 ether;
+       if (tokenAmount > curvesTokenBalance[
    curvesTokenSubject][address(this)]) revert
    InsufficientBalance();
        address externalToken = externalCurvesTokens
            [curvesTokenSubject].token;

        if (externalToken == address(0)) revert
            TokenAbsentForCurvesTokenSubject();
        if (amount > CurvesERC20(externalToken).
            balanceOf(msg.sender)) revert
            InsufficientBalance();
-       if (tokenAmount > curvesTokenBalance[
    curvesTokenSubject][address(this)]) revert
    InsufficientBalance();

        CurvesERC20(externalToken).burn(msg.sender,
            amount);
        _transfer(curvesTokenSubject, address(this),
             msg.sender, tokenAmount);
    }
```

Fig. 8. An example showing the correct fix: the InsufficientBalance check is placed after validating the external token mapping. The model failed to apply this ordering due to a lack of contextual semantic understanding.

In the example shown in Fig. 9, the correct gas check expression hookCallGasLimit $*$ 64 / 63 should be preserved or transformed carefully to maintain its numeric equivalence. The model instead proposed hookCallGasLimit $<<$ 6 / 63, which changes the intended gas threshold and would lead to incorrect behavior. This example demonstrates that effective gas

```
    function callHook(
        address target,
        bytes memory data,
        bool ignoreFailure,
        uint32 hookCallGasLimit,
        uint32 hookGasBuffer
    ) internal returns (bool) {-
-       if (gasleft() < (hookCallGasLimit * 64 / 63
    + hookGasBuffer)) revert NotEnoughGas();
+       if (gasleft() < (hookCallGasLimit << 6 / 63
    + hookGasBuffer)) revert NotEnoughGas();
        ...
    }
```

Fig. 9. A gas-saving arithmetic optimization: multiplication/division is replaced with an equivalent left-shift, preserving the gas threshold.

```
struct TokenConfig {
    AggregatorV3Interface feed; // chainlink feed
    uint32 maxFeedAge;
    uint8 feedDecimals;
    uint8 tokenDecimals;
-   uint32 twapSeconds;
    IUniswapV3Pool pool; // reference pool
    bool isToken0;
+   uint32 twapSeconds;
    Mode mode;
    uint16 maxDifference; // max price difference
        x10000
}
```

Fig. 10. A missed storage layout optimization: reordering `twapSeconds` before `pool` allows better packing of storage slots and reduces gas costs. The model fails to apply this optimization despite it being present in the prompt's intent.

optimization requires not only recognizing structural patterns but also reasoning about the arithmetic semantics in context, as overlooking such dependencies can result in incorrect or suboptimal transformations.

3) Reason 3: Semantic Drift in Task Interpretation: Unlike semantic hallucination, which introduces logically incorrect transformations, semantic drift refers to cases where the model's modifications follow the optimization intent superficially but miss the actual goal, resulting in changes that look valid yet fail to achieve the intended efficiency gains. For example, when the instruction aims to reorder struct fields to save storage slots, the model may instead attempt to change field types. Although such changes appear to follow an optimization intent, they diverge from the actual requirement and do not achieve the intended gas savings. This kind of drift occurs because models tend to prioritize familiar or surface-level transformations rather than reasoning about domain-specific rules like EVM storage layout.

As shown in Fig. 10, the optimal fix is to place the 32-bit field `twapSeconds` before the 160-bit `pool` reference, enabling tighter storage packing and saving one slot per struct instance. The model, however, retained the original order, reflecting semantic drift: its modifications deviated from the intended optimization goal, either overlooking the required reordering or introducing irrelevant changes such as altering field types. This drift occurs because the model tends to prioritize familiar patterns or perceived "safe" transformations rather than reasoning about low-level storage efficiency. As a result, without explicit understanding of EVM storage layout, large language models may miss impactful gas optimizations or produce fixes that appear valid but fail to achieve the intended efficiency gains.

VII. Related Work

In this section, we briefly review the existing research on gas optimization, and gas estimation for smart contracts.

A. Gas Optimization

Source code level optimization: The optimization of contract source code is intuitive, whose main idea is to find the instances of some gas-inefficient patterns in the source code and replace them with efficient ones. In 2017, Chen et al. [12] was the first to investigate and highlight the widespread presence of inefficient code in smart contracts, identifying how these inefficiencies often lead to excessive gas costs. A 2020 empirical study [35] found that specific object-oriented metrics are statistically correlated with gas consumption, offering insights into estimating deployment costs. Later work [36] showed that smart contracts are generally less readable than traditional software and that specific readability metrics are correlated with gas consumption. Albert et al. [15] introduced GASOL, a tool designed to optimize gas consumption by focusing primarily on the SLOAD and SSTORE opcodes. Their approach detects repeated storage access and refactors the code by introducing local variables to store data, minimizing subsequent storage accesses. However, their optimization is limited to just these two instructions, highlighting the need for further extension to cover additional gas-consuming operations. Li et al. [9] effectively reduced gas consumption in smart contracts by eliminating redundant array bound checks in the EVM. Nelaturu et al. [23] proposed a gas optimization approach for loop operations in Solidity, focusing on redundant computations and storage accesses. Using SOLIOS [16], they identified patterns and refactored the code, but found that introducing a summary function increased deployment costs. They also faced limitations in verifying arithmetic operations with solc-verify [37], indicating the need for better optimization and verification tools. Brandstätter et al. [29] discussed 25 code optimization strategies for Solidity smart contracts, developing a prototype that detects and partially automates optimizations. Kong et al. [38] introduced a method to detect gas-inefficient code snippets by identifying six manually defined patterns related to improper storage use and the misuse of costly operations. For each identified pattern, they also offered specific optimization techniques to address the inefficiencies. Nguyen et al. [22] presented GasSaver, an automated tool designed to analyze smart contract source code. GasSaver utilizes three extended logic rules and four new rules to detect gas-inefficient code, offering recommendations for code modifications aimed at lowering transaction costs. Porkodi et al. [39] introduced various techniques to optimize gas costs in Ethereum smart contracts, such as data type preference, dead code elimination, loop operations, and Monte-Carlo methods to reduce transaction fees and vulnerabilities. He et al. [40] analyzed common causes of gas waste in Solidity contracts and developed PeCatch, a

static tool with eight checkers that detected 383 previously unknown issues in popular libraries. Jiang et al. [18] proposed using LLMs to identify 26 gas-wasting code smells in Solidity, 13 of which are novel, providing detailed reports and example contracts from both prior literature and recent deployments.

`Bytecode level optimization:` Bytecode optimization also receives lots of attention. An early work, ebso [41], a superoptimizer that targets EVM bytecode directly by encoding instruction semantics into SMT constraints to automatically synthesize cheaper equivalents. Chen et al. [12] identified the problem of the smart contracts being overcharged and designed a tool named GASPER to detect the gas costly patterns in the smart contracts. But the paper only focuses on two categories of the problem such as identifying useless code and problems involving loops. This work is then extended as Gas Reducer [13] and Gas Checker [14]. The Gas Reducer focuses on identifying the high gas consumption patterns and then replaces them with equivalent code that is efficient in optimizing the gas cost and the Gas Checker focuses on optimizing by parallelizing the process of the symbol execution. But both the extension works are not open sourced thus they are not publicly available. Asem Ghaleb [42] et al. proposed eTainter, a static analysis tool based on bytecode-level taint tracking, targets gas-related vulnerabilities that can lead to DoS attacks. Kaur et al. [43] introduced GASaVER, a symbolic execution tool that detects mergeable loops in smart contracts, optimizing bytecode to reduce gas fees by addressing costly gas patterns with efficient alternatives. In contrast to existing approaches, Liu et al. [44] introduced FunRedisp, a bytecode refactoring tool that reduces invocation gas consumption in Solidity smart contracts by optimizing function dispatch. FunRedisp identifies frequently invoked functions and moves them to the front of the dispatch, saving 125.17 gas units per transaction with minimal compilation overhead.

B. Gas Estimation

Accurately estimating gas consumption is crucial for optimizing gas usage. Previous studies explored various methods for gas estimation. Easley et al. [45] researched about the impact on transaction fees, rewards for miners and wait time. Then Deep neural network was preferred by Tedeschi et al. [46] to detect the oscillation in the transaction cost of the Ethereum network in turn reducing the wait time and optimizing the cost spent by the users. Challenges such as random block creation and unknown miner policies are observed. Based on the 19 identified Solidity code smells at the source code level, GasMet [47] statically evaluates the code quality of a smart contract from the gas consumption perspective. Li et al. [9] proposed a machine learning-based approach for gas estimation in loop functions. They collected transaction traces, replayed them on a local blockchain, and built a gas estimator model. Challenges like collecting specific traces and handling long opcode sequences are addressed by using Ethereumjs and three trace abstractions. Butler et al. [48] investigated post-London Hard Fork transaction dynamics on Ethereum, focusing on gas price forecasting. They compared machine learning models like Direct-Recursive Hybrid LSTM, CNN-LSTM, and Attention-LSTM, combined with wavelet denoising and matrix profile data processing. Hybrid LSTM models outperform others, enabling better gas price selection and reducing transaction risks.

VIII. Conclusion

In this paper, we first gather gas optimization data from Code4rena and related automated tools. After conducting an overall review, we categorize the data based on gas-inefficient patterns and analyze the limitation of the static analysis tool. Based on the analysis, we develop a combined approach with Slither and LLMs to detect and address gas inefficiencies in source code. Slither performs an initial static analysis to identify potential optimization patterns, which the LLM then verifies, confirming pattern presence and suggesting improvements. Our testing shows that this approach achieves high precision in identifying gas-inefficient patterns while maintaining a low false positive rate.

Data Availability Statement

The datasets used in our empirical study and evaluation, the examples for each category, and the source code of our approach are available at https://github.com/mgdj/LLM4GO.

References

[1] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.
[2] J. Kehrli, "Blockchain 2.0 - From bitcoin trans. To smart contract applications," 2016. [Online]. Available: https://www.niceideas.ch/roller2/badtrash/entry/blockchain-2-0-from-bitcoin
[3] P. Sharma, R. Jindal, and M. D. Borah, "A review of smart contract-based platforms, applications, and challenges," *Cluster Comput.*, vol. 26, pp. 395–421, 2023, doi: 10.1007/s10586-021-03491-1.
[4] LLL, "Ethereum low-level lisp-like language," 2021. [Online]. Available: https://lllddocs.readthedocs.io/en/latest/index.html
[5] Solidity, "Solidity: A statically-typed curly-braces programming language," 2022. Accessed: Oct. 23, 2024. [Online]. Available: https://soliditylang.org/
[6] Z. Zheng et al., "An overview on smart contracts: Challenges, advances and platforms," *Future Gener. Comput. Syst.*, vol. 105, pp. 475–491, Apr. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19316280
[7] T. Chen et al., "An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks," in *Information Security Practice and Experience*, J. K. Liu and P. Samarati, Eds. Cham, Switzerland: Springer Int. Publishing, 2017, pp. 3–24.
[8] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, Tech. Rep. 151, 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[9] C. Li, "Gas estimation and optimization for smart contracts on Ethereum," in *Proc. 36th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2021, pp. 1082–1086.
[10] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "ETHIR: A framework for high-level analysis of Ethereum bytecode," in *Automated Technology for Verification and Analysis*. Cham, Switzerland: Springer International Publishing. 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:29167862
[11] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of super-optimized smart contracts using max-SMT," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham, Switzerland: Springer Int. Publishing, 2020, pp. 177–200.
[12] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, 2017, pp. 442–446.

[13] T. Chen et al., "Towards saving money in using smart contracts," in *Proc. 40th Int. Conf. Softw. Eng.: New Ideas Emerg. Results (ICSE-NIER)* New York, NY, USA: Assoc. Comput. Machinery, 2018, pp. 81–84, doi: 10.1145/3183399.3183420.

[14] T. Chen et al., "GasChecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1433–1448, Jul.–Sep. 2021.

[15] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas analysis and optimization for Ethereum smart contracts," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham, Switzerland: Springer Int. Publishing, 2020, pp. 118–125.

[16] B. Mariano, Y. Chen, Y. Feng, S. K. Lahiri, and I. Dillig, "Demystifying loops in smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA: Assoc. Comput. Machinery, 2021, pp. 262–274, doi: 10.1145/3324884.3416626.

[17] E. Keilty, K. Nelaturu, A. Kastania, and A. Veneris, "Gas optimization patterns in move smart contracts on the aptos blockchain," in *Proc. 5th Conf. Blockchain Res. Appl. Innov. Netw. Services (BRAINS)*, 2023, pp. 1–4.

[18] J. Jiang et al., "Unearthing gas-wasting code smells in smart contracts with large language models," *IEEE Trans. Softw. Eng.*, vol. 51, no. 4, pp. 879–903, Apr. 2025.

[19] "Code4rena." Accessed: Sep. 23, 2025. [Online]. Available: https://code4rena.com/

[20] "4naly3er." Accessed: Sep. 23, 2025. [Online]. Available: https://github.com/Picodes/4naly3er

[21] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, 2020, pp. 1877–1901.

[22] Q.-T. Nguyen, B. S. Do, T. T. Nguyen, and B.-L. Do, "GasSaver: A tool for solidity smart contract optimization," in *Proc. 4th ACM Int. Symp. Blockchain Secure Crit. Infrastructure (BSCI)*, New York, NY, USA: Assoc. for Computing Machinery, 2022, pp. 125–134, doi: 10.1145/3494106.3528683.

[23] K. Nelaturu, S. M. Beillahi, F. Long, and A. Veneris, "Smart contracts refinement for gas optimization," in *Proc. 3rd Conf. Blockchain Res. Appl. Innov. Netw. Services (BRAINS)*, 2021, pp. 229–236.

[24] D. Spencer, *Card Sorting: Designing Usable Categories.* Brooklyn, NY, USA: Rosenfeld Media, 2009.

[25] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Piscataway, NJ, USA: IEEE Press, May 2019, pp. 8–15, doi: 10.1109/WETSEB.2019.00008.

[26] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. 36th Int. Conf. Neural Inf. Process. Syst. (NIPS),* Red Hook, NY, USA: Curran Associates Inc., 2022, pp. 24824–24837.

[27] G. Feng, B. Zhang, Y. Gu, H. Ye, D. He, and L. Wang, "Towards revealing the mystery behind chain of thought: A theoretical perspective," in *Proc. 37th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, Red Hook, NY, USA: Curran Associates Inc., 2023, pp. 70757–70798.

[28] F. Shi et al., "Language models are multilingual chain-of-thought reasoners," 2022, *arXiv:2210.03057.*

[29] T. Brandstatter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, 2020, pp. 281–290.

[30] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018, doi: 10.1145/3276486.

[31] E. Sannini, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "The gas burn identity: Early detection gas-intensive functions patterns in smart contracts, " vol. 1, 2025.

[32] K. Li et al., "Static application security testing (SAST) tools for smart contracts: How far are we?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1447–1470, Jul. 2024, doi: 10.1145/3660772.

[33] "Foundry." GitHub. Accessed: Feb. 10, 2025. [Online]. Available: https://github.com/foundry-rs/foundry/

[34] "Hardhat." Accessed: Feb. 10, 2025. [Online]. Available: https://hardhat.org/

[35] N. Ajienka, P. Vangorp, and A. Capiluppi, "An empirical analysis of source code metrics and smart contract resource consumption," *J. Softw. Evol. Process.*, vol. 32, no. 10, Oct. 2020, Art. no. e2267, doi: 10.1002/smr.2267.

[36] A. Vacca, M. Fredella, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "An empirical investigation on the trade-off between smart contract readability and gas consumption," in *Proc. 30th IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, New York, NY, USA: ACM, 2022, pp. 214–224, doi: 10.1145/3524610.3529157.

[37] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments*, S. Chakraborty and J. A. Navas, Eds. Cham, Switzerland: Springer Int. Publishing, 2020, pp. 161–179.

[38] Q.-P. Kong et al., "Characterizing and detecting gas-inefficient patterns in smart contracts," *J. Comput. Sci. Technol.*, vol. 37, no. 1, pp. 67–82, Feb. 2022, doi: 10.1007/s11390-021-1674-4.

[39] S. Porkodi and D. Kesavaraja, "Escalating gas cost optimization in smart contract," *Wirel. Pers. Commun.*, vol. 136, no. 1, pp. 35–59, Jun. 2024, doi: 10.1007/s11277-024-11066-7.

[40] M. He et al., "How to save my gas fees: Understanding and detecting real-world gas issues in solidity programs," 2024. [Online]. Available: https://arxiv.org/abs/2403.02661

[41] J. Nagele and M. A. Schett, "Blockchain superoptimizer," 2020. [Online]. Available: https://arxiv.org/abs/2005.05912

[42] A. Ghaleb, J. Rubin, and K. Pattabiraman, "eTainter: detecting gas-related vulnerabilities in smart contracts," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: Assoc. Comput. Machinery, 2022, pp. 728–739, doi: 10.1145/3533767.3534378.

[43] K. Kaur, S. Tomar, and M. Tripathi, "Gas fee reduction by detecting loop fusible patterns in Ethereum smart contract," in *Proc. IEEE Int. Conf. Adv. Networks Telecommun. Syst. (ANTS)*, 2022, pp. 458–463.

[44] Y. Liu and W. Song, "FunRedisp: Reordering function dispatch in smart contract to reduce invocation gas fees," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: Assoc. Comput. Machinery, 2024, pp. 516–527, doi: 10.1145/3650212.3652146.

[45] D. Easley, M. O'Hara, and S. Basu, "From mining to markets: The evolution of bitcoin transaction fees," *J. Financial Econ.*, vol. 134, no. 1, pp. 91–109, 2019. [Online]. Available: https://ideas.repec.org/a/eee/jfinec/v134y2019i1p91-109.html

[46] E. Tedeschi, T.-A. S. Nordmo, D. Johansen, and H. D. Johansen, "On optimizing transaction fees in bitcoin using AI: Investigation on miners inclusion pattern," *ACM Trans. Internet Technol.*, vol. 22, no. 3, pp. 1–28, Jul. 2022, doi: 10.1145/3528669.

[47] A. Di Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora, "Profiling gas consumption in solidity smart contracts," *J. Syst. Softw.*, vol. 186, Apr. 2022, Art. no. 111193. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221002697

[48] C. Butler and M. Crane, "Blockchain transaction fee forecasting: A comparison of machine learning methods," *Mathematics*, vol. 11, no. 9, 2023, Art. no. 2212. [Online]. Available: https://www.mdpi.com/2227-7390/11/9/2212