# Tetris for Gas:
# State Variable Mutation for Reducing Smart Contract Deployment Costs

CHARALAMBOS MITROPOULOS, Technical University of Crete, Greece
DIMITRIS VLACHOS, National and Kapodistrian University of Athens, Greece
VAGGELIS SAROUKOS, National and Kapodistrian University of Athens, Greece
SOTIRIS IOANNIDIS, Technical University of Crete, Greece
DIMITRIS MITROPOULOS, National and Kapodistrian University of Athens, Greece

Smart contract deployment costs constitute a critical economic consideration in blockchain ecosystems, yet existing gas optimization approaches primarily focus on execution efficiency while neglecting deployment gas reduction. We analyze Solidity storage internals and identify that conventional variable packing–despite reducing storage slots–often increases deployment costs due to compiler-generated complexity involving masking and shifting operations. To address this problem, we present *State Variable Mutation*, the first systematic approach designed specifically for reducing smart contract deployment costs through guided reordering of state variable declarations. Our approach explores variable orderings to identify layouts that minimize gas-expensive storage operations while preserving semantic equivalence and maintaining storage efficiency.

We implement our approach in DGRed, an open-source tool for deployment workflows, and conduct an extensive empirical evaluation on 300 real-world smart contracts. Results demonstrate deployment gas reductions of up to 32.72%, with an average reduction of 15.65% (52,850 gas units per contract), translating to total savings of 15,854,883 gas units across all contracts. Under high network congestion (200 gwei), these savings correspond to $12,381.4. We compare DGRed against state-of-the-art gas optimization tools, including GasSaver and GASOL. Our evaluation shows that DGRed achieves superior deployment gas reductions (15.65% average vs. 4.91% for GasSaver and 2.83% for GASOL) while maintaining 100% semantic preservation and producing valid bytecode for all 300 contracts. We further show that DGRed's state variable mutation produces identical execution gas to the original contract in all 100 contracts evaluated for execution gas impact, confirming that deployment optimization does not affect runtime efficiency. Additionally, DGRed's 15.65% reduction is over 8× larger than the best achievable through Solidity compiler flag tuning alone, demonstrating that the two approaches are complementary and target orthogonal aspects of deployment gas. In contrast, existing tools frequently introduce compilation errors (GasSaver: 148/300 contracts) or generate invalid bytecode (GASOL: 234/300 contracts). DGRed demonstrates both greater effectiveness and reliability, providing developers with a practical, semantic-preserving solution for deployment cost optimization without modifying contract logic or functionality. DGRed only reorders state variable declarations and provably preserves contract semantics; it therefore constitutes a strict improvement: developers can apply it unconditionally as a free optimization, with no runtime trade-off and no risk of behavioral regression.

CCS Concepts: • **Software and its engineering → Software performance**; *Development frameworks and environments*; • **Theory of computation → Program optimization**.

Authors' Contact Information: Charalambos Mitropoulos, Technical University of Crete, Greece, cmitropoulos@tuc.gr; Dimitris Vlachos, National and Kapodistrian University of Athens, Greece; Vaggelis Saroukos, National and Kapodistrian University of Athens, Greece; Sotiris Ioannidis, Technical University of Crete, Greece, sioannidis@tuc.gr; Dimitris Mitropoulos, National and Kapodistrian University of Athens, Greece, dimitro@ba.uoa.gr.


Manuscript submitted to ACM

Additional Key Words and Phrases: Smart contracts, Gas optimization, Deployment costs, Solidity, Blockchain, Ethereum, State variables

**ACM Reference Format:**

Charalambos Mitropoulos, Dimitris Vlachos, Vaggelis Saroukos, Sotiris Ioannidis, and Dimitris Mitropoulos. 2026. Tetris for Gas: State Variable Mutation for Reducing Smart Contract Deployment Costs. 1, 1 (July 2026), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Gas consumption represents a fundamental economic mechanism in blockchain ecosystems [34, 35, 54, 56], directly determining the cost of deploying and executing smart contracts on platforms such as Ethereum [5, 6]. As the computational fuel that powers blockchain operations, gas costs impact the accessibility and scalability of decentralized applications, making gas optimization a valid concern for developers and organizations operating in resource-constrained environments.

In Solidity, the predominant smart contract programming language [14], gas consumption manifests in two distinct phases: *deployment gas*, incurred during the one-time contract creation process, and *execution gas*, consumed during runtime function calls and state modifications [6]. While both types contribute to the overall cost of blockchain operations, they exhibit different characteristics and optimization opportunities. Deployment costs represent an upfront investment that can become prohibitive during high gas price periods, particularly affecting experimental contracts, systems requiring frequent redeployment, and projects with limited initial funding. The economic implications become evident when examining real-world deployment scenarios: our evaluation across 300 smart contracts demonstrates total deployment gas savings of 15,854,883 gas units (Section 4.2). Under current low network conditions (3 gwei, ETH = $3,900), these savings translate to $185.6 – a modest amount. However, during high-congestion periods at 200 gwei – typical during peak network activity – the same optimizations yield $12,381.4 in savings, representing a 66× cost amplification. This volatility underscores why deployment costs constitute a critical consideration, making optimization essential for budget predictability and resource management.

Existing gas optimization approaches have primarily focused on reducing execution costs through bytecode transformations [24], source code modifications [64], and pattern-based strategies [25, 39, 46, 60]. However, these approaches exhibit a number of limitations: (a) they primarily target execution gas while neglecting deployment costs, and (b) they fail to adequately consider Solidity's storage internals and their impact on gas consumption patterns.

Beyond these general-purpose optimization tools, a well-established optimization paradigm in Solidity development involves *variable packing*, i.e., the systematic ordering of state variables by size to minimize storage slot utilization [12, 16, 70]. While this technique reduces storage footprint and can improve execution efficiency through consolidated access patterns, our analysis reveals a counterintuitive consequence: variable packing frequently *increases* deployment gas costs due to compiler-generated overhead for operations such as masking and shifting [3, 11, 12, 19], required when handling initialized variables within shared storage slots [31, 40].

Deployment gas optimization addresses critical gaps in current practice by reducing upfront costs, accelerating development cycles, decreasing testnet dependencies, and enabling complex contract deployment within budget constraints. For resource-constrained projects and decentralized organizations, these optimizations preserve funds for other initiatives while maintaining full contract functionality.

We present *State Variable Mutation*, a systematic approach for reducing smart contract deployment costs through guided reordering of state variable declarations. Our technique, implemented in DGRed (Deployment Gas Reducer), employs an optimization strategy inspired by search-based software engineering [27, 38, 48] and fuzzing techniques [28,

45, 52, 59, 61]. The approach performs targeted exchanges of declaration positions, evaluating each arrangement for gas reduction while ensuring semantic preservation and storage efficiency.[1] Note that, maintaining storage efficiency is essential in blockchain environments where storage represents a premium resource affecting both long-term gas costs and network scalability. The key insight of our mutation is that state variable declaration order significantly influences compiler storage layout decisions and initialization bytecode complexity. Through systematic exploration of variable orderings, our technique discovers arrangements that minimize deployment gas costs while maintaining optimal storage allocation.

We evaluate our approach on 300 real-world smart contracts, demonstrating deployment gas reductions of up to 32.72% (357,297 gas saved) with an average reduction of 15.65% (52,850 gas saved) across all contracts. The approach proves especially beneficial for complex production contracts, i.e., contracts with over 20 state variables, extensive state variable initialization, and external calls performed in the constructor achieve deployment gas reductions exceeding 10%. Furthermore, our evaluation shows that our mutation produces identical execution gas to the original contract in all 100 contracts evaluated for execution gas impact (Section 4.3). In contrast, traditional variable packing typically increases deployment costs, revealing the inadequacy of storage-focused optimizations for deployment efficiency.

We compare DGRed against state-of-the-art gas optimization tools GasSaver and GASOL. Our evaluation shows that DGRed achieves superior deployment gas reductions – 15.65% average compared to 4.91% for GasSaver and 2.83% for GASOL – while demonstrating substantially higher reliability. DGRed successfully produces valid, optimized contracts for all 300 evaluated contracts (100% success rate), whereas GasSaver generates compilation errors in 148 cases (49% failure rate) due to semantic alterations, and GASOL produces invalid bytecode in 234 cases (78% failure rate). Beyond effectiveness, DGRed maintains complete semantic equivalence, ensuring that optimized contracts preserve the exact functionality and behavior of their original versions – a critical requirement that existing tools frequently violate. We further show that DGRed's savings are complementary to the Solidity compiler's built-in optimization flags: the best compiler tuning (`--optimize-runs=50`) achieves only 1.86% deployment gas reduction, while DGRed achieves 15.65%—an 8× improvement (Section 4.5).

**Contributions.** The main contributions of this paper are:

- We analyze Solidity storage internals (Section 2.2) and identify the deployment gas optimization problem, demonstrating how variable packing can increase deployment costs through compiler-generated complexity (Section 2.3).
- We introduce a systematic approach for reducing deployment gas through guided reordering of state variable declarations, preserving contract semantics while optimizing storage layout decisions (Section 3).
- We conduct an extensive empirical study on 300 real-world smart contracts, demonstrating consistent gas savings that surpass variable packing in effectiveness while outperforming existing state-of-the-art tools in both effectiveness and reliability, and showing that DGRed complements the Solidity compiler's built-in optimization flags (Section 4).
- We provide DGRed as an open-source tool that developers can integrate into their deployment workflows to achieve immediate cost reductions without modifying contract logic or functionality (Section 3.4).

[1]The word "Tetris" in the title reflects the analogy between our method and the classic puzzle game: just as Tetris requires strategically rearranging falling blocks to minimize wasted space, our approach restructures state variable declarations to find layouts that minimize deployment gas costs.

```
contract Unixo {
    mapping(address => User) public users;
    uint256 public last_id;
    uint256 public value;
    bool    public op;
    address payable public root;
    mapping(uint256 => address) public
          users_ids;
    uint24  public LEVEL_TIME = 30 days;
    uint256[] public levels;
    constructor() public {
    ...
```

(a) `Unixo` contract. Deployment cost: 1,518,322.

```
contract Unixo {
    bool    public op;
    uint24  public LEVEL_TIME = 30 days;
    address payable public root;
    uint256 public last_id;
    uint256 public value;
    mapping(address => User) public users;
    mapping(uint256 => address) public
          users_ids;
    uint256[] public levels;
    constructor() public {
    ...
```

(b) State Variable Packing. Deployment cost: 1,524,274.

```
contract Unixo {
    bool    public op;
    uint256 public last_id;
    uint256 public value;
    mapping(address => User) public users;
    address payable public root;
    uint24  public LEVEL_TIME = 30 days;
    mapping(uint256 => address) public
          users_ids;
    uint256[] public levels;
    constructor() public {
    ...
```

(c) State Variable Mutation. Deployment cost: 1,512,831.

Fig. 1. Deployment gas comparison for different state variable orderings in `Unixo` contract.

## 2 Background and Motivation

To motivate our deployment gas optimization approach, we first provide essential background on Solidity's state variables and storage model (Section 2.1), then examine Solidity's storage internals and how the compiler's bytecode generation process affects deployment costs (Section 2.2). We then discuss the limitations of current optimization practices, demonstrating how conventional variable packing–despite being widely recommended–can increase deployment costs (Section 2.3). This analysis reveals a critical gap: existing approaches optimize for storage utilization or execution efficiency while neglecting deployment-specific considerations, motivating the need for a targeted deployment optimization strategy.

### 2.1 State Variables and Storage in Solidity

Solidity contracts define *state variables* [15] to persist data across function calls and transactions. Unlike local or memory variables, state variables are stored on-chain in the Ethereum Virtual Machine (EVM)'s storage space [4], and thus incur gas costs when written. State variables can be of both value and reference types [18], and may be initialized at declaration time or in a constructor.

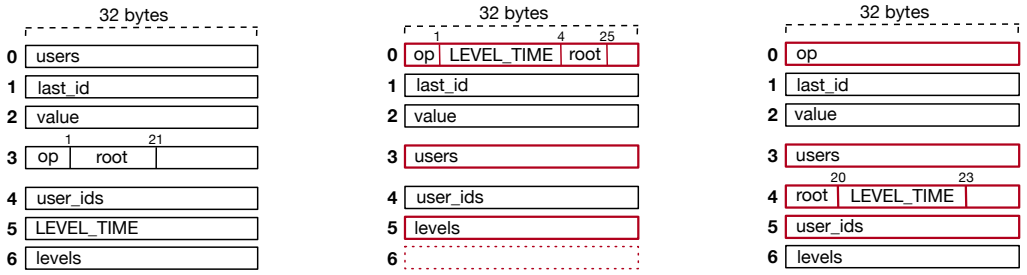
The Solidity compiler [14] transforms source code into EVM bytecode [8] for blockchain deployment. Its optimization decisions directly affect deployment size and runtime gas costs, especially in how it handles state variables, which are a primary contributor to storage usage and gas consumption.

Solidity uses a deterministic storage model in which each state variable is assigned to a fixed 32-byte (256-bit) *slot* [12]. These slots are allocated sequentially based on the order of declarations in the source code, starting at slot 0. Efficient slot usage is important, as the EVM charges significantly more gas for storage writes (`SSTORE`) than for reads [6].

Notably, when a state variable is initialized to a non-zero value, the compiler emits bytecode that writes this value to storage using the gas-expensive `SSTORE` opcode [21]. In contrast, variables with a default value of zero often avoid this cost, as the EVM assumes zero for all uninitialized slots.

### 2.2 Solidity Storage Internals

The relationship between state variable declaration order and deployment gas costs emerges from the Solidity compiler's bytecode generation process for storage initialization. Understanding this relationship requires examining three aspects: (1) the cost model for storage operations, (2) the compiler's storage layout decisions, and (3) how these decisions influence initialization bytecode complexity.




(a) Baseline slots. (b) Slots after Variable Packing. (c) Slots after State Variable Mutation.

Fig. 2. Unixo Storage Slot Variations.

*2.2.1* SSTORE *Operations and Gas Costs.* The EVM's SSTORE operation, which writes data to storage, operates under two distinct cost models [6, 22]:

- *Cold* SSTORE (22,100 gas): When writing to a storage slot for the first time in a transaction (changing from zero to non-zero), the EVM charges a premium cost representing the allocation of a new storage slot.
- *Warm* SSTORE (5,000 gas): Subsequent writes to a storage slot already accessed in the same transaction incur lower costs, as the slot is already loaded in memory.

During contract deployment, all state variables with non-zero initializers trigger SSTORE operations. The compiler's organization of these operations impacts total gas consumption, particularly when variables share storage slots.

*2.2.2 Storage Layout and Bytecode Complexity.* The Solidity compiler assigns state variables to 32-byte storage slots based on their declaration order and size compatibility. When multiple small variables can fit within a single slot, the compiler may pack them together to optimize storage utilization. However, this packing decision creates a trade-off between storage efficiency and initialization complexity.

For variables stored in separate slots, the compiler generates straightforward SSTORE operations for each initialized variable. When variables are packed together, the initialization process becomes more complex. If any packed variable requires initialization, the compiler must generate a sequence of bitwise operations to preserve existing values while updating the target variable:

(1) Load the entire slot using SLOAD
(2) Apply bitmasks (AND to isolate the target variable's byte range
(3) Shift the new value to the correct position within the slot
(4) Merge (OR) the new value with preserved bits from other variables
(5) Write the complete slot back using SSTORE

This sequence of operations increases both bytecode size and deployment gas consumption [31, 40]. Each additional operation—SLOAD, AND, shift, OR—adds to the initialization bytecode that executes once during deployment, directly increasing deployment costs. When multiple packed variables require initialization, the compiler generates even more complex sequences, each involving multiple SLOAD and SSTORE operations.

*2.2.3 Declaration Order Impact.* The order in which state variables are declared in source code directly influences which variables the compiler considers for packing and the resulting storage layout. Different orderings can lead to

different packing arrangements, each with distinct initialization costs. Some arrangements minimize the number of slots requiring initialization, while others reduce the complexity of required bitwise operations.

### 2.3 Limitations of Current Approaches

*2.3.1 Variable Packing.* Solidity proposes an optimization technique named variable packing [12, 51, 60] to (a) enhance the storage footprint of a contract, and (b) reduce its execution gas costs. In this context, developers order state variables by size (smallest to largest) to pack multiple small variables into a single 32-byte storage slot [16, 51, 70]. For example, a `bool` (1 byte), `uint24` (3 bytes), and `address` (20 bytes) can share one slot instead of occupying three separate slots. As a result, the EVM will perform only one `SLOAD` operation to read all variables and one `SSTORE` operation to write them back, rather than separate operations for each variable. However, despite reducing storage slots, variable packing can increase deployment gas due to the additional operations required for handling initialized variables in packed slots. A single non-zero initialized variable sharing a slot with other variables requires complex bytecode sequences that may outweigh the benefits of reduced slot count.

We demonstrate this effect through `Unixo`, a real-world smart contract deployed on Etherscan. Figure 1a presents the state variables of the original contract, while Figure 1b presents a version where variables are reordered based on variable packing.

In the original contract, slots 0–2 and 4–6 each store one variable. Slot 3 contains both `op` (1 `byte`) and `root` (20 `bytes`) which are packed together (see Figure 2a). However, since both variables are initialized to zero, the compiler skips emitting the gas-expensive `SSTORE` opcode. During deployment, one storage write will occur: during the initialization of `LEVEL_TIME` (slot 5), where the `SSTORE` opcode will be employed. The total deployment gas cost of the baseline contract is 1,518,322 gas.

When the state variables are reordered with variable packing method (Figure 1b), `LEVEL_TIME` now resides in slot 0 alongside `op` and `root` (see Figure 2b). This means that the compiler has to preserve the values of `op` and `root` while updating only the 3 bytes that correspond to `LEVEL_TIME`. This triggers the compiler to generate a sequence of instructions including masking, shifting, and storage operations [3, 11, 12, 19]. These extra instructions (as analyzed in Section 2.2) increase bytecode size and gas deployment cost, make it even bigger than the deployment cost of the baseline contract (1,524,274 gas).

*2.3.2 Existing Gas Optimization Tools.* GasSaver [64] is a source code analyzer designed to reduce gas costs and improve runtime efficiency. To do so, it modifies the contract's code through data type transformations (e.g. converting state variables to constants) and removes redundant code. However, GasSaver can introduce semantic changes and change the overall behavior and functionality of the contract under test (leading to compilation errors as we discuss in Section 4).

GASOL [24] is a bytecode optimizer that operates on EVM instructions directly. Specifically, GASOL offers cost models for analyzing gas consumption and rewrites bytecode when it identifies costly patterns. Nevertheless, these low-level optimizations usually produce invalid bytecode, as we show in Section 4.

## 3 Approach

Building upon the limitations identified in Section 2.3 and the storage internals analyzed in Section 2.2, we present State Variable Mutation–a systematic technique for reducing deployment gas costs through guided reordering of state variable declarations. Our approach leverages the key observation that declaration order significantly influences compiler-generated bytecode complexity during contract initialization, while variable semantics remain preserved

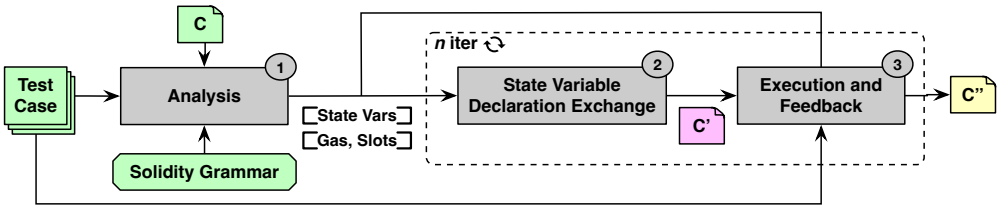



Fig. 3. Approach overview for decreasing deployment gas.

through identifier-based access. We formalize the fundamental properties enabling semantic-preserving optimization (Section 3.2), present our iterative optimization algorithm (Section 3.3), and describe its implementation in the DGRed tool (Section 3.4). We start with an architectural overview that illustrates how these components integrate to achieve deployment gas reduction.

### 3.1 Overview

Figure 3 illustrates the basic steps of our approach for reducing deployment gas costs in Solidity smart contracts. We begin with an *Analysis* (①) step that takes a contract ($C$) and its corresponding test suite as input. During this step, we extract information about the contract's structure, including the complete set of its state variables and their assigned storage slots. Our analysis establishes baseline measurements for both deployment gas cost and storage slot utilization, which serve as feedback for evaluating potential improvements. As described in Section 2.2, Solidity storage internals present specific challenges for deployment gas optimization.

Subsequently, our *State Variable Declaration Exchange* (②) rearranges state variable declarations based on specific properties. We describe these properties in Section 3.2. Unlike traditional approaches that recommend ordering variables strictly by size, our strategy employs random exchanges that are informed by gas and slot feedback rather than relying on predefined heuristics. For each candidate arrangement, our *Execution and Feedback* (③) step compiles the modified contract ($C'$) and executes the corresponding test suite to measure deployment gas costs. If the new arrangement reduces the deployment gas of the contract while maintaining or reducing the total number of storage slots, it is accepted as the better choice. Otherwise, we retain the previous arrangement.

The result of this step is provided as input to steps ② and ③ for a number of iterations that enable our approach to converge toward an optimal solution, as we explain in Section 3.3. The final contract $C''$ maintains semantic equivalence with $C$ as we discuss in Section 3.2.1.

### 3.2 Design Properties

Following our observations on storage internals (Section 2.2), the compiler's operation costs and the complexity of masking and shifting operations, we present the basic design properties of our approach.

*3.2.1 Semantic Preservation.* An important design principle of our approach is to maintain the semantic equivalence of contracts throughout the deployment gas optimization process. To do so, we focus on reordering state variable declarations.

Definition 1 (Semantic Preservation). *Let $\Sigma$ denote the set of all possible EVM states, and let $\mathcal{T}$ represent the set of all possible transactions. For a contract $C$ with state transition function $\delta_C : \Sigma \times \mathcal{T}^* \to \Sigma$, where $\mathcal{T}^*$ denotes sequences of transactions, and its transformed variant $C'$ with transition function $\delta_{C'}$, we say that $C'$* preserves the semantics *of $C$ if*

*and only if:*

$$\forall \sigma_0 \in \Sigma, \forall T = \langle t_1, t_2, \ldots, t_n \rangle \in \mathcal{T}^* : \delta_C(\sigma_0, T) \equiv \delta_{C'}(\sigma_0, T) \tag{1}$$

*where $\equiv$ denotes observational equivalence, meaning both contracts produce identical: (1) final storage states, (2) return values, (3) event emissions, (4) external interactions, and (5) revert conditions for all possible transaction sequences and initial states.*

The state variable declaration order in Solidity affects storage layout but not contract logic. This is because variables are accessed through their identifiers rather than their declaration positions [12, 23]. This enables us to safely reorder declarations while maintaining semantic preservation.

Property 1 (Declaration Order Semantic Independence). *Let $C$ be a contract with state variables $V = \{v_1, v_2, \ldots, v_n\}$ declared at source positions $\{p_1, p_2, \ldots, p_n\}$, where each variable $v_i$ is characterized by the tuple $\langle id_i, \tau_i, init_i \rangle$ representing its identifier, type, and initializer respectively. Let $C'$ be a contract derived from $C$ through permutation $\pi : [n] \to [n]$ of declaration positions, such that variable $v_i$ in $C$ appears at position $p_{\pi(i)}$ in $C'$. Then $C'$ preserves the semantic equivalence of $C$ according to Definition 1.*

Justification. Semantic preservation follows from three key observations about Solidity's compilation model:

*(1) Identifier-based access:* The Solidity compiler resolves variable references through identifiers rather than declaration positions. For any expression $e$ containing variable reference $v_i$, the compiler generates bytecode accessing the storage location associated with identifier $id_i$. Since reordering preserves all identifiers $\{id_1, \ldots, id_n\}$, all references resolve identically in both $C$ and $C'$.

*(2) Storage mapping consistency:* While the compiler may assign different storage slots in $C'$ compared to $C$ due to reordering, this mapping is bijective and consistent throughout the contract. Each variable maintains its type $\tau_i$ and initializer $init_i$, ensuring equivalent storage semantics under the new layout.

*(3) Control flow preservation:* Declaration reordering affects only the storage layout computation phase, not the control flow graph or function body bytecode generation. Since function bodies reference variables by identifier, and identifiers map deterministically to storage locations in both contracts, all execution paths remain equivalent. Therefore, for any transaction sequence $T$ and initial state $\sigma_0$, the execution trace follows identical control flow with equivalent storage accesses, establishing $\delta_C(\sigma_0, T) \equiv \delta_{C'}(\sigma_0, T)$. □ □

*3.2.2 State Variable Declaration Exchange.* We propose a simple yet effective transformation that exchanges the declaration positions of state variable declarations in a contract.

Definition 2 (State Variable Declaration Exchange). *Let $C$ be a contract with state variables $V = \{v_1, v_2, \ldots, v_n\}$ whose declarations appear at source code positions $\{p_1, p_2, \ldots, p_n\}$. We define the operator:*

$$\text{exchange} : C \times V \times V \to C$$

*as the function that, given a contract $C$ and two distinct variables $v_i, v_j \in V$, returns the contract obtained by swapping the source positions of the declarations of $v_i$ and $v_j$ while leaving every other declaration in place. That is,* $\text{exchange}(C, v_i, v_j) = C'$ *is the unique contract satisfying:*

- *The declaration of $v_i$ now appears at position $p_j$*
- *The declaration of $v_j$ now appears at position $p_i$*
- *All other variable declarations $v_k$ (where $k \neq i, j$) remain at their original positions $p_k$*

*A* declaration exchange *operation randomly selects two distinct variables $v_i, v_j \in V$ (where $i \neq j$) and produces $C' = \text{exchange}(C, v_i, v_j)$. The variables themselves—their identifiers, types, and initializers—remain unchanged; only their declaration positions are swapped.*

While a declaration exchange preserves program semantics (by Property 1), not all exchanges are beneficial for deployment gas optimization. We consider an exchange as valid if it reduces deployment gas without increasing the number of storage slots.

Definition 3 (Valid Declaration Exchange). *Let $\mathcal{G} : \mathcal{C} \to \mathbb{N}$ denote the deployment gas cost function and $\mathcal{S} : \mathcal{C} \to \mathbb{N}$ denote the storage slot count function for contracts in space $\mathcal{C}$. A declaration exchange transformation $\tau : C \mapsto C'$ via exchange$(C, v_i, v_j)$ is* valid *if and only if:*

*(1) Syntactic Validity: $C'$ compiles successfully with identical contract interface (same public functions and state variable accessors)*
*(2) Semantic Preservation: $C'$ preserves the semantics of $C$ according to Definition 1*
*(3) Gas Improvement: $\mathcal{G}(C') < \mathcal{G}(C)$, representing strict reduction in deployment gas cost with improvement magnitude $\Delta\mathcal{G} = \mathcal{G}(C) - \mathcal{G}(C') > 0$*
*(4) Storage Constraint: $\mathcal{S}(C') \leq \mathcal{S}(C)$, ensuring non-increasing storage slot count*

*We denote the set of all valid transformations from $C$ as:*

$$\Phi(C) = \{\tau : C \mapsto C' \mid \tau \text{ satisfies conditions 1–4}\} \tag{2}$$

These four validity conditions work together to ensure optimization quality: syntactic validity and semantic preservation (conditions 1–2) guarantee correctness, while gas improvement and storage constraint (conditions 3–4) ensure efficiency. The storage constraint is particularly important because increased storage slots would raise both deployment and runtime gas costs for contract interactions, potentially negating deployment benefits over the contract's lifetime [6]. Moreover, in blockchain environments, storage represents an expensive resource that impacts the overall network state size [5, 6].

*3.2.3 Iterative Optimization Strategy.* Our approach operates within a search space consisting of all possible orderings of state variable declarations. For a contract with $n$ state variables, this space contains $n!$ possible orderings, making exhaustive exploration computationally infeasible for contracts with many variables.

As previously mentioned, the effect of any individual declaration exchange on deployment gas cost depends on several interacting factors (see Section 2.3) that interact in non-trivial ways, making it difficult to predict a priori which exchanges will constitute valid declaration exchanges.

We therefore use guided random sampling to discover beneficial state variable declaration arrangements.

Property 2 (Declaration Reordering Gas Impact). *For a contract $C$ with state variables $v_i, v_j \in V$, the deployment gas impact of exchanging their declarations is:*

$$\Delta\mathcal{G}_{i,j} = \mathcal{G}(\text{exchange}(C, v_i, v_j)) - \mathcal{G}(C) \tag{3}$$

*where $\Delta\mathcal{G}_{i,j} < 0$ indicates beneficial gas reduction. This impact arises from changes in the compiler's storage layout decisions and initialization bytecode generation, which affect:*

*(1) Storage Slot Assignment: How variables are packed into 32-byte slots*

(2) *Initialization Bytecode: The complexity of masking and shifting operations required for handling initialized variables in shared slots*
(3) *SSTORE Operation Patterns: Whether SSTORE operations during initialization are cold (22,100 gas) or warm (5,000 gas). Every storage slot is zero at contract creation time, so the* first *write to any slot during deployment is always cold. A warm* `SSTORE` *(5,000 gas) can therefore arise during deployment only when the constructor (or the inline initializers that the compiler folds into the constructor) first reads and then writes the* same *slot—the typical case being a packed slot in which several initialized variables coexist and the compiler emits a sequence of read-modify-write updates against that slot.*

*The magnitude of $\Delta\mathcal{G}_{i,j}$ depends on the interplay between these factors, which cannot be predicted analytically due to compiler optimization complexity. Our empirical observations (Section 4) show that exchanges involving initialized variables sharing slots with other variables tend to produce the largest gas impacts.*

Having characterized the factors influencing deployment gas costs, we now formalize our optimization objective.

Definition 4 (Optimization Objective). *Given a contract $C_0$ with initial deployment gas cost $g_0 = \mathcal{G}(C_0)$ and storage slot count $s_0 = \mathcal{S}(C_0)$, let $\mathcal{R}(C_0)$ denote the set of all contracts obtainable by permuting the state variable declarations of $C_0$. Our optimization objective is to find a contract $C^* \in \mathcal{R}(C_0)$ such that:*

$$C^* = \arg\min_{C \in \mathcal{R}(C_0)} \mathcal{G}(C) \quad \textit{subject to} \quad \mathcal{S}(C) \leq s_0 \tag{4}$$

*The optimization gap achieved by solution $C^*$ is quantified as:*

$$\eta = \frac{g_0 - \mathcal{G}(C^*)}{g_0} \times 100\% \tag{5}$$

*representing the percentage reduction in deployment gas cost.*

To solve the optimization problem defined above, our approach employs an iterative method inspired by fuzzing techniques [26, 28–30, 45, 52, 59]. Given a fixed number of iterations $N$, the algorithm performs sequences of random state variable declaration exchanges, evaluating each for gas reduction and retaining only those that are valid. In each iteration, the algorithm forms disjoint pairs of variables whose declarations to exchange, applies these exchanges simultaneously, and measures the resulting gas impact. If the new arrangement reduces deployment gas without increasing storage slots, it becomes the new baseline for subsequent iterations. The monotonic non-increase of this iterative process can be formally characterized.

Property 3 (Monotonically Non-Increasing). *Our iterative optimization approach generates a sequence of contracts $\{C_0, C_1, \ldots, C_N\}$ with corresponding gas costs $g_i = \mathcal{G}(C_i)$, where $C_0$ is the initial contract. Since only improvements are accepted, the gas sequence is monotonically non-increasing: $g_{i+1} \leq g_i$ for all $i$. We measure convergence using the relative improvement rate:*

$$r_i = \frac{g_{i-1} - g_i}{g_0 - g_i} \tag{6}$$

*where decreasing $r_i$ indicates diminishing returns. Empirically, $r_i$ decreases over iterations (Section 4.1), demonstrating empirical stabilization within finite iterations.*

The optimal number of iterations $N$ involves a trade-off between optimization effectiveness and computational efficiency. While theoretically the algorithm could run until no further improvements are found ($r_i \approx 0$), in practice we determine

an empirically optimal value for $N$ that captures most potential gas savings while maintaining reasonable execution time. Our analysis in Section 4.1 demonstrates that approximately 30 iterations provides an effective balance.

### 3.3 State Variable Mutation

Algorithm 1 presents how we incorporate the properties established in Section 3.2 to solve the optimization problem defined in Definition 4. First, `Analyze` (line 1) performs a number of calculations to retrieve two key elements. Specifically, it records the initial deployment gas cost $g_0 = \mathcal{G}(C)$ of a given contract $C$ using the test suite $\mathcal{T}$. Then, it determines the set of storage slots used by the contract's state variables and retrieves the total slot count $s_0$. `StateVariables` extracts the ordered set of state variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ from $C$ (line 2). After that, we initialize tracking variables for the best solution found so far: $C^*$ for the optimized contract and $g^*$ for its gas cost (line 3).

We then repeat a specific set of steps $N$ times. Initially, we select a set of disjoint pairs of state variables to mutate. If the number of state variables ($n$) is odd (line 5), one variable is randomly excluded (line 6), and the remaining $n - 1$ variables are shuffled and grouped into $\lfloor (n-1)/2 \rfloor$ disjoint pairs (lines 7-9). If $n$ is even, all variables are shuffled and formed into $n/2$ disjoint pairs (lines 11-13). This strategy of random pairings enables efficient exploration of the variable ordering space while ensuring that each variable participates in at most one mutation per iteration. Once the exchange pairs are determined, we create a working copy of the best solution found so far, $C'$ (line 15), and apply state variable declaration exchanges for each pair (lines 17). Each exchange operation reorders the source code by swapping the positions where two state variable declarations appear, without modifying the variables' properties or the contract's logic. We then measure the deployment gas cost $g'$ of the modified contract and calculate its storage slot count $s'$ (lines 19). After that, we apply the validity conditions from Definition 3: the new layout is accepted only if it reduces the deployment gas cost ($g' < g^*$) while not increasing the number of storage slots ($s' \leq s_0$) (line 20). If these conditions are met, we update the best-known solution $C^*$ and its gas cost $g^*$ (lines 21). Otherwise, the current iteration's changes are discarded. After completing all iterations, we return $C^*$ (line 24), representing the most gas-efficient layout discovered during the optimization process. The convergence behavior of this iterative process follows Property 3, with the gas cost decreasing monotonically across iterations.

By applying our mutation to the `Unixo` contract discussed in Section 2.3, we discover the optimized arrangement of state variables illustrated in Figure 1c. Specifically, our algorithm produced an updated version of the `Unixo` contract with two state variable declaration pairs exchanged: (1) `op` with `users` and (2) `LEVEL_TIME` with `users_ids`. In this optimized layout, `LEVEL_TIME` is packed only with `root` in slot 4, while `op` occupies its own slot (see Figure 2c).

This arrangement reduces deployment gas in two ways. First, the compiler needs to preserve only one zero-initialized variable in slot 4, resulting in fewer masking and shifting operations than those of the variable-packed layout in Figure 2b. This happens because three state variables share the same slot in Figure 2b. Moreover, our layout improves upon the baseline (Figure 2a) as well. In the baseline, `LEVEL_TIME` is stored alone in its own slot and is initialized with a non-zero value, causing the compiler to emit a "cold" `SSTORE` – a costly 32-byte storage write [12, 22]. In contrast, in the layout generated by our approach, the compiler avoids emitting such a "cold" store. Instead, it emits a compact sequence of bitwise operations to insert `LEVEL_TIME` into its byte range within a zeroed slot [11, 70] and then performs a "warm" `SSTORE` operation. This optimization achieves an improvement of $\eta = 0.36\%$ compared to the baseline (5,491 gas units saved) and $\eta = 0.75\%$ compared to variable packing (11,443 gas units saved), using the optimization gap metric from Definition 4.

**Algorithm 1** State Variable Mutation

**Require:** Solidity contract $C$, test suite $\mathcal{T}$ for contract $C$, iteration bound $N \in \mathbb{N}^+$
**Ensure:** Optimized contract $C^*$ with minimized deployment gas cost

$(g_0, s_0) \leftarrow \texttt{Analyze}(C, \mathcal{T})$
$\mathcal{V} = \{v_1, v_2, \ldots, v_n\} \leftarrow \texttt{StateVariables}(C)$
$C^* \leftarrow C$, $g^* \leftarrow g_0$
**for** $i \leftarrow 1$ to $N$ **do**
  **if** $n \bmod 2 = 1$ **then**
    $\mathcal{V}' \leftarrow \mathcal{V} \setminus \{v_{random}\}$
    $\sigma \leftarrow \texttt{RandomPermutation}(n-1)$
    $\mathcal{V}'_\sigma \leftarrow (v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n-1)})$
    $\mathcal{P} \leftarrow \{(v_{\sigma(2j-1)}, v_{\sigma(2j)}) \mid j \in \{1, 2, \ldots, \lfloor \frac{n-1}{2} \rfloor\}\}$
  **else**
    $\sigma \leftarrow \texttt{RandomPermutation}(n)$
    $\mathcal{V}_\sigma \leftarrow (v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)})$
    $\mathcal{P} \leftarrow \{(v_{\sigma(2j-1)}, v_{\sigma(2j)}) \mid j \in \{1, 2, \ldots, \frac{n}{2}\}\}$
  **end if**
  $C' \leftarrow C^*$
  **for** $(v_a, v_b) \in \mathcal{P}$ **do**
    $C' \leftarrow \texttt{Exchange}(C', v_a, v_b)$
  **end for**
  $(g', s') \leftarrow \texttt{Analyze}(C', \mathcal{T})$
  **if** $g' < g^* \wedge s' \leq s_0$ **then**
    $C^* \leftarrow C'$, $g^* \leftarrow g'$
  **end if**
**end for**
**return** $C^*$

### 3.4 Implementation

We implement our state variable mutation algorithm as a Python tool called DGRed (Deployment Gas Reducer), consisting of ~500 lines of code. We develop our state variable mutation technique using `Tree-sitter` [17], a parsing system widely employed in code analysis tools [36, 37] and also employed by `GitHub`'s navigation system [41]. Specifically, we utilize `Tree-sitter` to analyze Solidity source code, identify state variable declarations, and perform reorderings that preserve the contract's semantics. Additionally, we use Foundry Forge [9] to obtain feedback about deployment costs and contract storage slots.

**Limitations**. DGRed requires executable test cases to measure deployment gas during mutation. Since deployment testing involves only constructor-oriented tests rather than complex runtime scenarios, creating these test cases is typically straightforward. Modern development frameworks such as Hardhat [65] make test creation increasingly accessible, and deployment testing is often prioritized since developers must verify deployment across different environments. The effort invested in creating these test cases also benefits overall contract reliability beyond gas optimization.

## 4 Evaluation

We evaluate DGRed based on the following research questions. Throughout all experiments (RQ1–RQ5), contracts are compiled with the Solidity compiler's `-optimize` flag enabled and `-optimize-runs = 200`, which is the recommended

production configuration [20]. Although the compiler ships with the optimizer *disabled* by default, enabling it is standard practice in virtually all production deployments and development frameworks (e.g. Foundry, Hardhat, Remix).

- **RQ1:** How many iterations are needed before DGRed's deployment gas savings converge?
- **RQ2:** What deployment gas savings can DGRed achieve?
- **RQ3:** How does DGRed's state variable mutation compare to variable packing?
- **RQ4:** How does DGRed compare to state-of-the-art gas optimization tools?
- **RQ5:** How does DGRed compare against different compiler optimization tunings?

### 4.1 RQ1: Iteration Convergence Analysis

*4.1.1 Experimental Setup.* To determine the optimal number of iterations for deployment gas optimization, we select 300 recently deployed contracts using Etherscan [7]. Our selection ensures representativeness across different deployment scenarios encountered in production environments. Specifically, we include:

- 100 Contracts with simple constructors and 10-20 state variables, representing simpler smart contract complexity.
- 100 Contracts with more than 20 state variables to ensure sufficient optimization potential.
- 100 Contracts with complex constructors and more than 20 state variables that require external contracts and libraries. For these contracts, we employ established testing methodologies using mocks [10] and stubs to handle dependency requirements.

To ensure that our results are not driven by structurally similar contracts (e.g., a large number of near-identical ERC-20 tokens with the same state-variable layout) we verify the diversity of the dataset. We classify the 300 contracts according to the OpenZeppelin Contracts taxonomy [67], plus a *Custom* category for contracts that match no OpenZeppelin module. The per-category breakdown is shown in Table 1. To perform our classification we use two features of Slither [43], a static-analysis framework for Solidity. First, Slither builds the *inheritance graph* of each contract, which we match against OpenZeppelin's named modules—the token bases (`ERC20`, `ERC721`, `ERC1155`, `ERC4626`) and their official extensions [68], or a non-token module (Access Control, Governance, etc.) [66]. Second, when no OpenZeppelin parent is found (typically in Etherscan-flattened sources that do not inherit a named module) we use `slither-check-erc`, a built-in Slither command that reports whether the contract implements a standard token interface (`IERC20`, `IERC721`, `IERC1155`, or `IERC4626`). When the inheritance graph matches more than one named OpenZeppelin module, the contract is assigned to the first matching category in the order listed above. Two contracts are said to share a *same layout* when their state-variable declarations agree in length and in the sequence of declared types. As Table 1 shows, the 300 contracts span 15 distinct categories. Only 8 contracts share a layout with another contract in the same category; manual inspection confirms that each declares fewer than 10 state variables and that DGRed reduces their deployment gas by less than 5%, making their effect on the dataset-wide reduction negligible.

For each contract, we construct deployment test cases to ensure accurate gas measurements. Since deployment testing only requires invoking the contract constructor, this process is straightforward compared to runtime testing scenarios.

We execute DGRed for up to 40 iterations on each contract to assess convergence patterns in gas savings, tracking deployment gas costs relative to baseline measurements. We run our experiments on a machine with an Intel Xeon CPU 2.30GHz processor with 6 logical cores and 64GB of RAM.

*4.1.2 Results.* To identify the optimal number of iterations for deployment gas optimization, we use the relative improvement rate formula introduced in Section 3.2.3.

Table 1. Distribution of the 300 contracts under the OpenZeppelin Contracts taxonomy [67]. The last column reports, per category, contracts whose state-variable declarations agree in length and type sequence with at least one other contract in the same category.

| ***Category*** | **Description** | **# Contracts** | **# Same layout** |
|---|---|---|---|
| *ERC-20* | Fungible-token contracts | 39 | 4 |
| *Access Control* | `Ownable` / `AccessControl` / `AccessManager` | 32 | 0 |
| *ERC-721* | Non-fungible (NFT) contracts | 27 | 2 |
| *Governance* | `Governor` / `TimelockController` / `Votes` | 24 | 0 |
| *Proxy / Upgradeability* | ERC-1967, Transparent, UUPS, Beacon, `Initializable` | 21 | 0 |
| *ERC-1155* | Multi-token contracts | 20 | 0 |
| *ERC-4626* | Tokenised vault (yield-bearing) contracts | 20 | 0 |
| *Crosschain* | Cross-chain messaging and bridge contracts (ERC-7786) | 18 | 0 |
| *Cryptography* | `ECDSA`, `MerkleProof`, `EIP712`, `SignatureChecker` | 17 | 0 |
| *Account Abstraction* | ERC-4337 helpers (`BaseAccount`, `validateUserOp`) | 17 | 0 |
| *Library* | Reusable utility code declared with the `library` keyword | 14 | 2 |
| *Utils* | `Pausable`, `ReentrancyGuard`, `Multicall` | 13 | 0 |
| *Metatx* | Meta-transaction helpers (`ERC2771Context`) | 12 | 0 |
| *Finance* | `VestingWallet`, `PaymentSplitter` | 9 | 0 |
| *Custom* | Contracts that do not match any OpenZeppelin category | 17 | 0 |
| ***Total*** | | **300** | **8** |

Figure 4 illustrates the progression of gas savings from 5 to 40 iterations across all 300 contracts. Median savings improve steadily from 5.98% (63,147 gas units) at 5 iterations to 8.95% (73,564 gas units) at 10 iterations, 11.79% (85,705 gas units) at 15 iterations, 13.03% (96,637 gas units) at 20 iterations, 15.06% (117,015 gas units) at 25 iterations. and 16.43% (131,528 gas units) at 30 iterations. Beyond 30 iterations, additional gains become marginal, with median savings only increasing slightly to 17.41% (148,929 gas units) at 40 iterations.

Applying our convergence measure $r_i = \frac{g_i - g_{i-1}}{g_0 - g_i}$ (see Section 3.2.3) to these results reveals the following pattern:

- $r_{10-5}$ = 1.411 (notable improvement)
- $r_{15-10}$ = 1.671 (stronger improvement)
- $r_{20-15}$ = 1.793 (sustained strong improvement)
- $r_{25-20}$ = 1.906 (peak improvement rate)
- $r_{30-25}$ = 1.132 (marked slowdown)
- $r_{35-30}$ = 0.837 (further decline)
- $r_{40-35}$ = 0.526 (marginal gains only)

The results demonstrate that significant improvements occur up to approximately 30 iterations, after which the rate of additional gas savings diminishes. Although the late-iteration relative rates remain non-trivial (e.g., $r_{40-35} = 0.526$ is still $\approx 28\%$ of the peak rate), $r_i$ is a *relative* measure: in absolute terms each of these later steps adds less than half a percentage point of additional median reduction, which is what the "marginal gains" label refers to. The difference between 30 and 40 iterations represents an additional 0.50% reduction in gas costs, suggesting that 30 iterations provides an optimal balance.

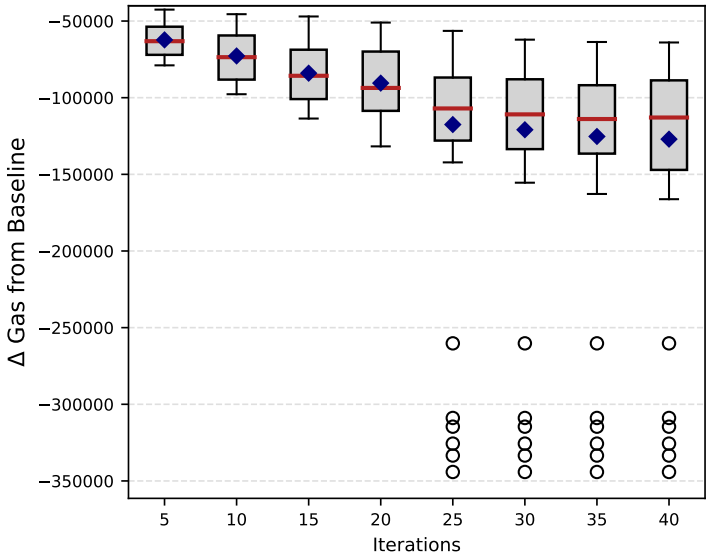



Fig. 4. **RQ1.** Deployment gas savings convergence across 40 iterations. Box plots show the distribution of gas savings for the 300 evaluated contracts at each iteration count. The horizontal line inside each box is the *median*; the box spans the *25th–75th percentile* (interquartile range); the whiskers extend to the most extreme data point within 1.5×IQR of the box; *circles* mark individual contracts that fall outside the whiskers (i.e., outliers); and the *blue diamond* on each box denotes the per-iteration *mean* savings across the 300 contracts.

Table 2. **RQ2.** Summary of DGRed's deployment gas savings across 300 evaluated contracts.

| Metric | Value |
|---|---|
| Contracts improved | 300 (100%) |
| Total gas saved | 15,854,883 gas |
| Average gas saved per contract | 52,850 gas |
| Median gas saved per contract | 37,620 gas |
| Maximum gas saved | 357,297 gas |
| Standard deviation | 56,632 gas |
| Mean reduction | 15.65% |
| Economic value (high congestion) | $12,381.4 |

### 4.2 RQ2. DGRed Deployment Gas Savings Assessment

*4.2.1 Experimental Setup.* To evaluate the practical impact of DGRed, we apply it to our 300-contract dataset for RQ1, to demonstrate real-world deployment gas savings. For each contract, we execute DGRed with 30 iterations (based on RQ1 findings) and measure the resulting deployment gas costs compared to the original contract layout.

*4.2.2 Results.* Table 2 summarizes the key statistics from our evaluation. DGRed achieves gas reduction for all analyzed contracts, with no contracts showing increased gas consumption after optimization, indicating that DGRed provides a safe and effective optimization approach.

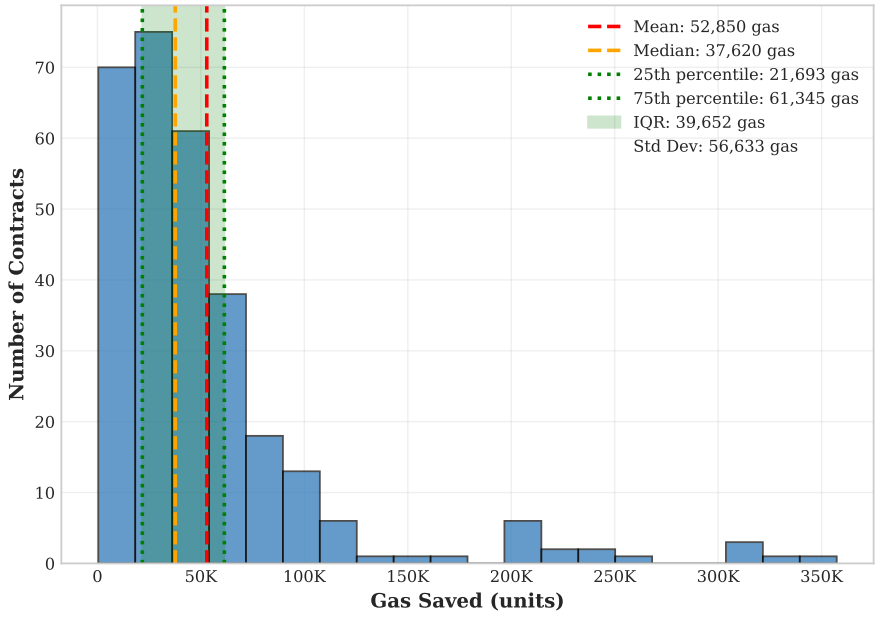



Fig. 5. **RQ2.** Distribution of deployment gas savings across 300 contracts.

Figure 5 shows the distribution of gas savings across all analyzed contracts. The distribution reveals a spread of optimization results with a mean of 52,850 gas and median of 37,620 gas per contract. Notably, DGRed achieves a mean reduction of 15.65% in deployment gas costs, demonstrating substantial optimization effectiveness across the entire dataset. The total gas saved—15,854,883 gas—translates to approximately $12,381.4 in savings during high congestion periods. The standard deviation of 56,632 gas quantifies the variability in optimization potential, indicating a moderate spread of savings around the mean. Our analysis reveals key percentiles: the 25th percentile at 21,693 gas, median (50th percentile) at 37,620 gas, 75th percentile at 61,345 gas, and maximum savings of 357,297 gas. The interquartile range (IQR), spanning from the 25th to 75th percentile as shown in the figure, represents the middle 50% of all contracts and highlights the typical range of savings achieved.

To better understand the relative impact of these savings, we categorized contracts based on the distribution's mean and standard deviation to categorize contracts by the percentage reduction in deployment gas cost, as shown in Figure 6. We define three categories:

- Low (<10.0%): 82 contracts (27.3%)
- Moderate (10.0-20.0%): 128 contracts (42.7%)
- High (20.0-32.7%): 90 contracts (30.0%)

Through qualitative analysis of the above categories, we observe clear patterns linking contract complexity to optimization effectiveness. DGRed achieves reductions exceeding 10% when contracts exhibit high complexity characteristics, i.e., more than 20 state variables, extensive initialization of state variables within the constructor (which aligns with our observations in Section 2.2), and external calls during constructor execution. These complex initialization patterns create more opportunities for our reordering technique to optimize the compiler-generated storage operations.

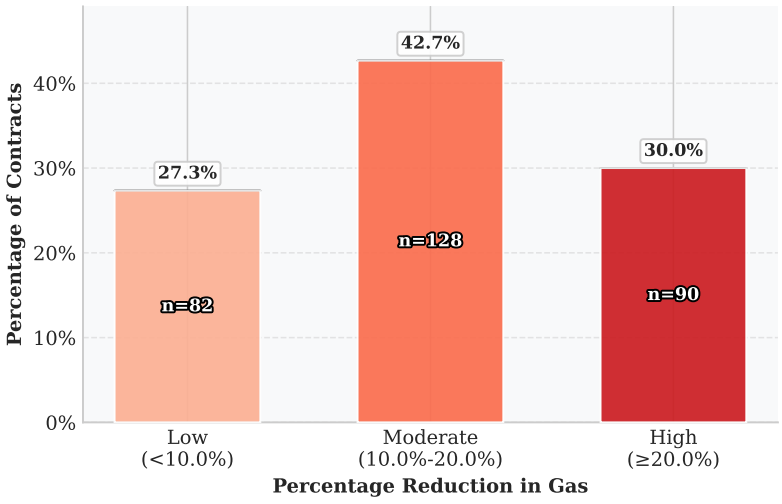



Fig. 6. **RQ2.** Percentage of contracts by improvement magnitude category.

Conversely, simpler contracts with 10-20 state variables and minimal constructor complexity yield lower reductions, typically below 10%, as they present fewer opportunities for variable declaration reordering. This correlation between contract complexity and optimization potential validates our approach's effectiveness on realistic production contracts while explaining the performance variation across our evaluation dataset.

*4.2.3 Deployment Gas Decomposition.* To clarify the scope of the reported savings, we decompose EVM deployment gas into its constituent components. When a smart contract is deployed, the total deployment gas comprises two parts:

$$G_{\text{deploy}} = G_{\text{constructor}} + G_{\text{code\_deposit}} \tag{7}$$

where $G_{\text{constructor}}$ is the gas consumed by executing the constructor bytecode (including a base creation cost of 32,000 gas, state variable initialization via `SSTORE` operations, and other setup logic), and $G_{\text{code\_deposit}} = |bytecode_{\text{runtime}}| \times 200$ is the fixed cost of storing the runtime bytecode on-chain (200 gas per byte, as specified in the Ethereum Yellow Paper [6]).

To quantify the relative weight of each component, we analyze all 300 contracts in our dataset (including multiple popular contracts with high transaction activity such as BetGPT and Cosmic Labs) using `forge inspect` to obtain both the deployment bytecode (which includes the constructor) and the runtime bytecode (which is stored on-chain). Table 3 reports the results.

Bytecode storage dominates deployment cost, accounting for 93.25% of total deployment gas, while constructor execution accounts for 6.75%. The average contract declares 16.1 state variables, of which only 2.1 (17.9%) are initialized inside the constructor; the remaining 14.0 (82.1%) are declared with default or inline-initialized values outside the constructor.

The 15.65% mean deployment gas reduction reported in Table 2 applies to the *total* deployment gas ($G_{\text{deploy}}$), not to any individual component. DGRED performs state variable reordering mutations across the *entire* contract body—not only inside the constructor. By optimizing the storage layout of all state variable declarations, DGRED affects the compiler's storage layout decisions and the resulting bytecode for both constructor execution and runtime code, thereby

Table 3. **RQ2.** Deployment gas decomposition across 300 Ethereum contracts. Constructor execution gas includes base creation cost, SSTORE operations, and initialization logic. Bytecode storage gas is the fixed cost of storing runtime code on-chain (200 gas/byte).

| **Component** | **Mean (gas)** | **%** |
|---|---|---|
| Constructor execution gas | 5,247,905 | 6.75% |
| Bytecode storage gas | 72,542,715 | 93.25% |
| **Total deployment gas** | **77,790,620** | **100%** |
| **Reduction by mutation scope** | **Mean** | **Contribution to 15.65%** |
| Constructor execution | 5.79% | 0.91% |
| Bytecode storage | 94.21% | 14.74% |
| **State variables** | **Mean (count)** | **%** |
| Initialized in constructor | 2.1 | 17.9% |
| Declared outside constructor | 14.0 | 82.1% |
| **Total per contract** | **16.1** | **100%** |

reducing the number of SSTORE operations and their associated gas costs during deployment. To qualify where these savings originate, we decompose the gas reduction using `forge inspect` across all 300 contracts. As shown in Table 3, the vast majority of the reduction (94.21%) stems from mutations in non-constructor state variables that affect the runtime bytecode (i.e., reduced $G_{\text{code\_deposit}}$), while only 5.79% comes from constructor state variable mutations (i.e., reduced $G_{\text{constructor}}$).

### 4.3 RQ3: Comparing State Variable Mutation with Variable Packing

*4.3.1 Experimental Setup.* Building on the illustrative example in Section 2.3.1, we evaluate DGRed's state variable mutation approach against variable packing. Our comparison involves both deployment and execution cost. To perform our measurements we utilize 100 contracts, specifically those with the highest transaction activity since their deployment. This reduced dataset is related to the fact that (a) we need to rearrange state variable definitions manually to achieve variable packing and (b) execution gas measurements require manual construction of test cases for state variable-intensive functions (functions that access the most state variables within a contract), which is a complex and time-intensive task. For each contract, we manually identified and tested the most state variable-intensive functions to assess whether deployment optimizations affect runtime efficiency.

*4.3.2 Results.* Figure 7 shows the deployment gas changes for both approaches relative to each contract's original layout. DGRed consistently reduces deployment gas across all 100 contracts, with average reductions of 111,620 gas units. In contrast, variable packing increases deployment gas costs, with an average increase of 4,260 gas units across the same contracts.

The results indicate that while variable packing may optimize storage slot usage, it often increases the complexity of deployment bytecode and thus the deployment gas cost [40]. This occurs because the compiler must generate additional masking and shifting operations to handle initialized variables packed in the same slot, as illustrated in our earlier `Unixo` example (Section 2.3).

To assess whether deployment gas optimizations affect runtime efficiency, we measured execution gas for each contract's most state variable-intensive function (see Table 4). Our analysis reveals that across all 100 contracts, DGRed's state variable mutation produces the same execution gas as the original contract for the most state variable-intensive

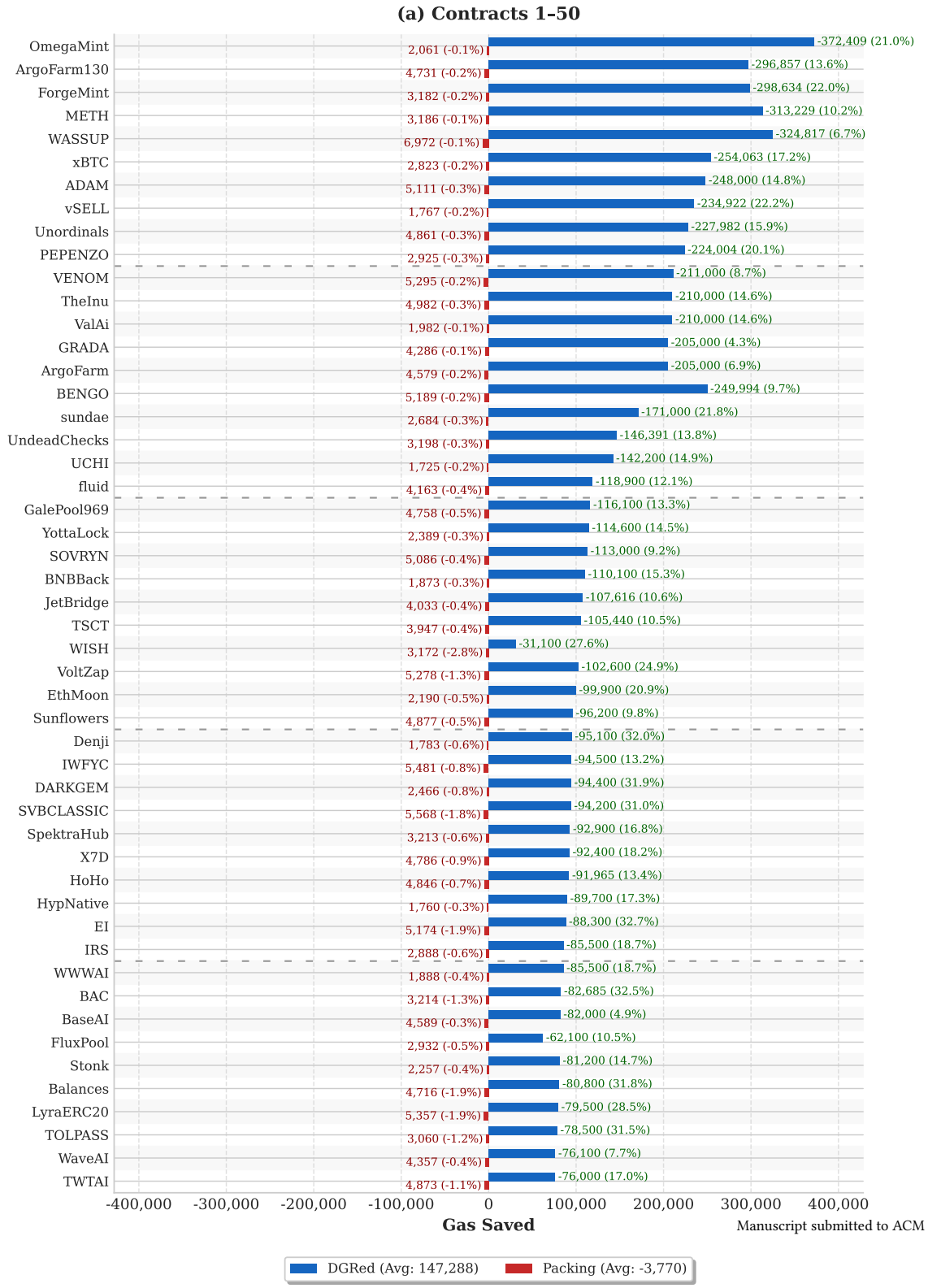





Fig. 7. **RQ3.** Deployment gas savings for DGRed vs. Variable Packing relative to baseline across 100 contracts. (a) Contracts 1–50.

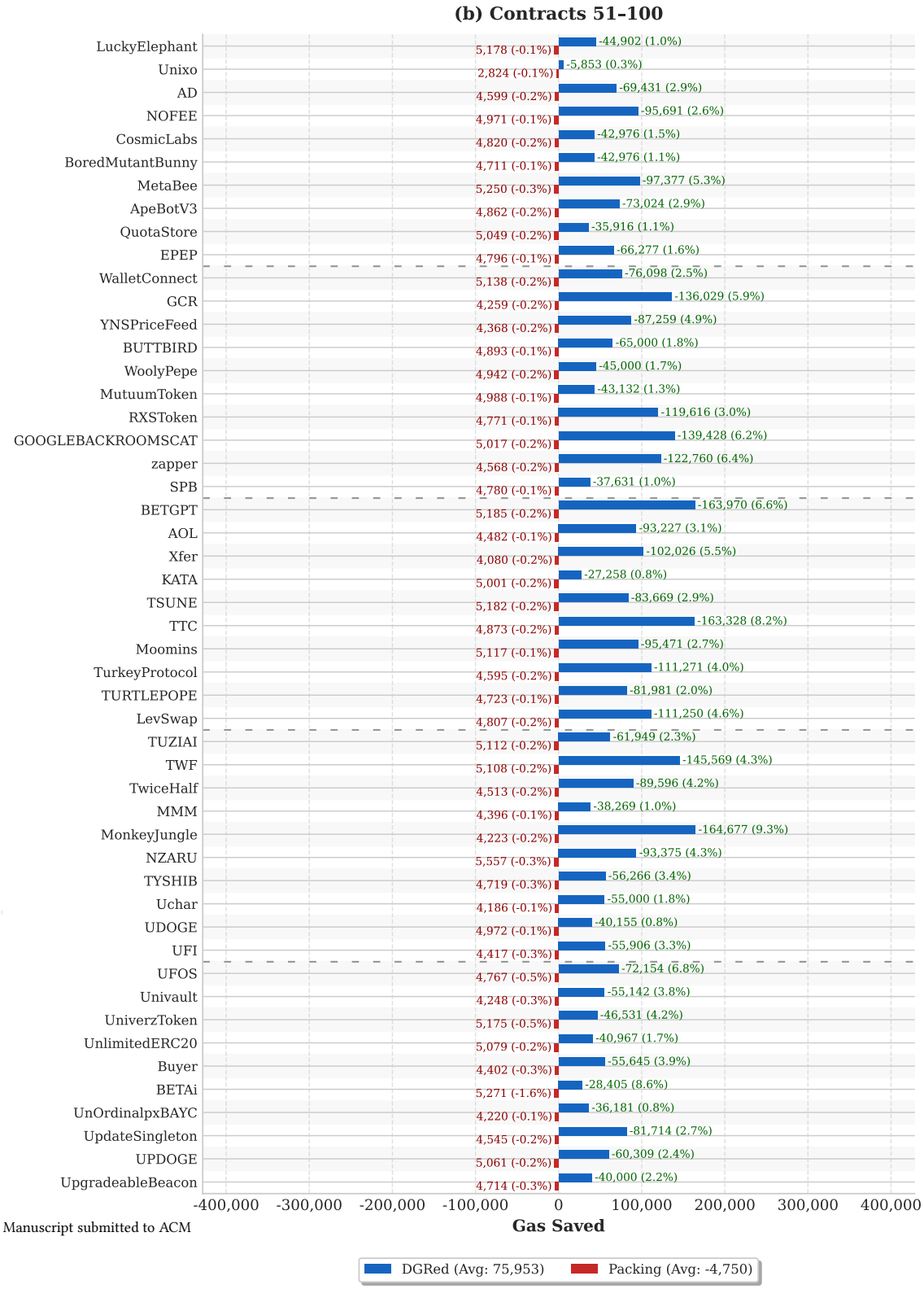



Fig. 7. Deployment gas savings for DGRed vs. Variable Packing relative to baseline across 100 contracts. (b) Contracts 51–100.

function, confirming that our approach does not affect runtime efficiency. Variable packing also produces the same execution gas as the original contract in 94 out of 100 contracts (94%). In the remaining 6 contracts, variable packing achieves lower execution gas for the most state variable-intensive function than the original contract (see Table 4). Each of these six functions accesses 3 state variables that are packed together in the variable packing approach, demonstrating the execution gas benefits described in Section 2.3. However, such cases are rare in practice, as we will discuss in Section 5.2.1.

Table 4. **RQ3.** Execution gas per state variable-intensive function call. The **# State Vars** column corresponds to the number of state variables accessed by the function. Rows in gray indicate contracts where variable packing reduces execution gas.

| Contract | Function | # State Vars | State Var. Mutation | Var. Packing | Original |
|---|---|---|---|---|---|
| OmegaMint | mint | 2 | 13,284 | 13,284 | 13,284 |
| ArgoFarm130 | transferFrom | 2 | 11,319 | 11,319 | 11,319 |
| ForgeMint | openTrading | 2 | 17,921 | 17,921 | 17,921 |
| METH | swapTokensForEth | 3 | 18,393 | 18,393 | 18,393 |
| WASSUP | increaseAllowance | 2 | 31,272 | 31,272 | 31,272 |
| xBTC | transfer | 3 | 22,763 | 22,763 | 22,763 |
| **JetBridge** | **transferFrom** | **3** | **15,385** | **14,811** | **15,385** |
| ADAM | transfer | 2 | 11,970 | 11,970 | 11,970 |
| vSELL | mint | 2 | 10,352 | 10,352 | 10,352 |
| Unordinals | burn | 3 | 12,588 | 12,588 | 12,588 |
| PEPENZO | exists | 2 | 9,285 | 9,285 | 9,285 |
| VENOM | transferFrom | 2 | 8,992 | 8,992 | 8,992 |
| **UCHI** | **transfer** | **3** | **10,139** | **9,985** | **10,139** |
| TheInu | mint | 3 | 8,284 | 8,284 | 8,284 |
| ValAi | transfer | 2 | 5,828 | 5,828 | 5,828 |
| GRADA | removeLimit | 2 | 62,768 | 62,768 | 62,768 |
| ArgoFarm | transferFrom | 2 | 7,817 | 7,817 | 7,817 |
| BENGO | exist | 1 | 3,228 | 3,228 | 3,228 |
| sundae | transfer | 3 | 4,271 | 4,271 | 4,271 |
| UndeadChecks | balances | 2 | 5,677 | 5,677 | 5,677 |
| fluid | increaseAllowance | 2 | 6,182 | 6,182 | 6,182 |
| Galepool969 | burn | 2 | 3,247 | 3,247 | 3,247 |
| YottaLock | remove | 2 | 7,184 | 7,184 | 7,184 |
| WaveAI | approve | 1 | 4,795 | 4,795 | 4,795 |
| BNBBack | transfer | 2 | 5,189 | 5,189 | 5,189 |
| TSCT | transferFrom | 2 | 4,677 | 4,677 | 4,677 |
| **SOVRYN** | **mint** | **3** | **5,162** | **5,002** | **5,162** |
| WISH | decreaseApproval | 1 | 2,116 | 2,116 | 2,116 |
| VoltZap | transfer | 2 | 3,490 | 3,490 | 3,490 |
| EthMoon | approve | 1 | 2,381 | 2,381 | 2,381 |
| Sunflowers | transfer | 2 | 7,157 | 7,157 | 7,157 |
| Denji | increaseAllowance | 2 | 5,512 | 5,512 | 5,512 |

Continued on next page

Table 4. **RQ3.** Execution gas per state variable-intensive function call (continued).

| Contract | Function | # State Vars | State Var. Mutation | Var. Packing | Original |
|---|---|---|---|---|---|
| IWFYC | isApprovedOrOwner | 2 | 3,916 | 3,916 | 3,916 |
| DARKGEM | transfer | 3 | 5,623 | 5,623 | 5,623 |
| SVBCLASSIC | tokenOfOwnerByIndex | 2 | 7,957 | 7,957 | 7,957 |
| SpektraHub | transferFrom | 3 | 6,819 | 6,819 | 6,819 |
| X7D | mint | 1 | 3,556 | 3,556 | 3,556 |
| Hoho | transfer | 3 | 6,167 | 6,167 | 6,167 |
| **VoltZap** | **mint** | **3** | **7,145** | **6,929** | **7,145** |
| HypNative | transfer | 2 | 3,178 | 3,178 | 3,178 |
| EI | revert | 1 | 7,895 | 7,895 | 7,895 |
| IRS | approve | 1 | 5,169 | 5,169 | 5,169 |
| WWWAI | burn | 2 | 3,558 | 3,558 | 3,558 |
| BAC | transfer | 3 | 4,668 | 4,668 | 4,668 |
| BaseAI | transfer | 3 | 7,162 | 7,162 | 7,162 |
| FluxPool | totalSupply | 1 | 3,578 | 3,578 | 3,578 |
| Stonk | mint | 2 | 5,571 | 5,571 | 5,571 |
| Balances | onlyOwner | 2 | 3,954 | 3,954 | 3,954 |
| LyraERC20 | increaseAllowance | 2 | 3,711 | 3,711 | 3,711 |
| TOLPASS | mint | 2 | 5,169 | 5,169 | 5,169 |
| TWTAI | transferFrom | 2 | 6,852 | 6,852 | 6,852 |
| zapper | constructor | 3 | 4,182 | 4,182 | 4,182 |
| SPB | transfer | 2 | 7,829 | 7,829 | 7,829 |
| BETGPT | transfer | 3 | 8,275 | 8,275 | 8,275 |
| AOL | burn | 2 | 6,261 | 6,261 | 6,261 |
| TSUNE | transfer | 2 | 3,272 | 3,272 | 3,272 |
| TTC | transferFrom | 2 | 3,264 | 3,264 | 3,264 |
| Moomins | mint | 1 | 3,215 | 3,215 | 3,215 |
| TURTLEPOPE | burn | 2 | 5,251 | 5,251 | 5,251 |
| **NZARU** | **transfer** | **3** | **4,162** | **3,972** | **4,162** |
| TYSHIB | increaseAllowance | 2 | 5,267 | 5,267 | 5,267 |
| UDOGE | mint | 1 | 9,252 | 9,252 | 9,252 |
| UFI | transferFrom | 2 | 4,258 | 4,258 | 4,258 |
| Univault | constructor | 1 | 4,162 | 4,162 | 4,162 |
| UniverzToken | buyToken | 3 | 6,155 | 6,155 | 6,155 |
| UnOrdinalpxBAYC | mint | 1 | 4,224 | 4,224 | 4,224 |
| UpdateSingleton | increaseAllowance | 3 | 10,178 | 10,178 | 10,178 |
| UPDOGE | transfer | 2 | 5,821 | 5,821 | 5,821 |
| LuckyElephant | transfer | 3 | 6,277 | 6,277 | 6,277 |
| Unixo | transferFrom | 1 | 4,612 | 4,612 | 4,612 |
| AD | transfer | 2 | 6,925 | 6,925 | 6,925 |
| NOFEE | transfer | 2 | 2,164 | 2,164 | 2,164 |

Continued on next page

Table 4. **RQ3.** Execution gas per state variable-intensive function call (continued).

| Contract | Function | # State Vars | State Var. Mutation | Var. Packing | Original |
|---|---|---|---|---|---|
| CosmicLabs | buy | 2 | 8,625 | 8,625 | 8,625 |
| BoredMutantBunny | mint | 1 | 3,331 | 3,331 | 3,331 |
| MetaBee | remove | 2 | 7,185 | 7,185 | 7,185 |
| ApeBotV3 | transfer | 2 | 7,881 | 7,881 | 7,881 |
| QuotaStore | opentrading | 3 | 9,271 | 9,271 | 9,271 |
| EPEP | transfer | 2 | 8,148 | 8,148 | 8,148 |
| WalletConnect | mint | 1 | 3,116 | 3,116 | 3,116 |
| GCR | remove | 2 | 4,722 | 4,722 | 4,722 |
| YNSPriceFeed | transfer | 2 | 7,185 | 7,185 | 7,185 |
| BUTTBIRD | approve | 3 | 7,974 | 7,974 | 7,974 |
| **WoolyPepe** | **mint** | **3** | **8,278** | **8,004** | **8,278** |
| MutuumToken | transfer | 2 | 5,714 | 5,714 | 5,714 |
| RXSToken | mint | 1 | 3,163 | 3,163 | 3,163 |
| GOOGLEBACKROOMSCAT | burn | 2 | 6,859 | 6,859 | 6,859 |
| Xfer | approve | 2 | 5,728 | 5,728 | 5,728 |
| KATA | transfer | 3 | 8,142 | 8,142 | 8,142 |
| TurkeyProtocol | openTrading | 3 | 10,016 | 10,016 | 10,016 |
| LevSwap | remove | 2 | 6,814 | 6,814 | 6,814 |
| TUZIAI | exist | 1 | 3,801 | 3,801 | 3,801 |
| TWF | transfer | 3 | 9,015 | 9,015 | 9,015 |
| TwiceHalf | mint | 1 | 2,794 | 2,794 | 2,794 |
| MMM | transferFrom | 3 | 6,050 | 6,050 | 6,050 |
| MonkeyJungle | approve | 2 | 5,995 | 5,995 | 5,995 |
| Uchar | transfer | 2 | 11,016 | 11,016 | 11,016 |
| UFOS | increaseAllowance | 3 | 9,178 | 9,178 | 9,178 |
| UnlimitedERC20 | mint | 1 | 2,905 | 2,905 | 2,905 |
| Buyer | totalSupply | 2 | 8,279 | 8,279 | 8,279 |
| BETAi | transfer | 2 | 6,170 | 6,170 | 6,170 |
| UpgradeableBeacon | approve | 2 | 4,874 | 4,874 | 4,874 |

### 4.4 RQ4: Comparison with State-of-the-Art Gas Optimizers

*4.4.1 Experimental Setup.* We compare DGRed against two state-of-the-art gas optimization tools: GasSaver [64] and GASOL [24], using the 300-contract dataset of RQ1 and RQ2.

*4.4.2 Results.* Figure 8 shows the deployment gas savings per contract across 50 contracts with the highest transaction activity after optimization by each tool, while Table 5 summarizes the overall performance comparison across all evaluated tools, in all 300 contracts.

Across all 50 contracts, DGRed consistently achieved greater deployment gas reductions than both GasSaver and GASOL (see Figure 8). On average, DGRed saved 147,786 gas, compared to 47,629 for GasSaver and 14,429 for GASOL. Regarding effectiveness, DGRed achieves an average deployment gas reduction of 15.65% across all contracts (see also

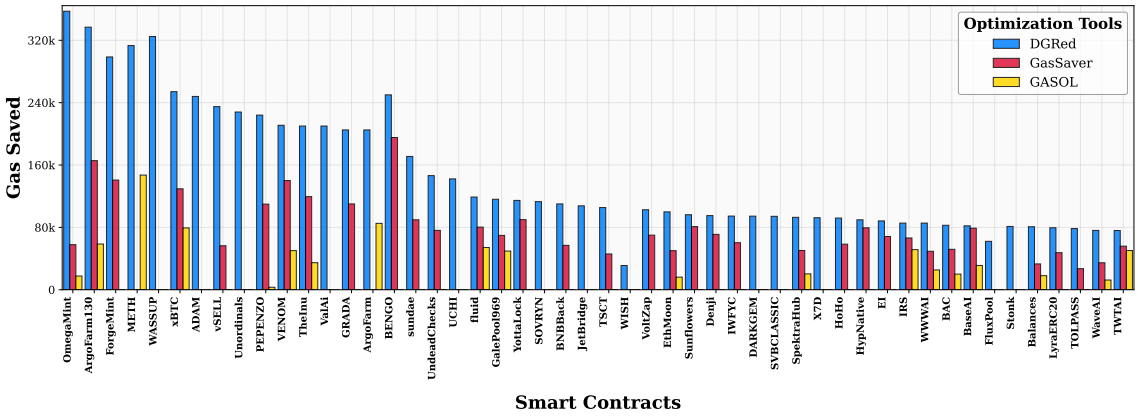



Fig. 8. **RQ4.** Deployment gas reduction comparison across 50 smart contracts with highest transaction activity for DGRed, GasSaver, and GASOL. DGRed outperforms both tools on every contract evaluated.

Section 4.2), compared to 4.91% for GasSaver and 2.83% for GASOL. Notably, DGRed outperforms both competing tools on every contract in our evaluation dataset. The maximum gas reduction further highlights DGRed's advantages, with peak savings of 32.72% for DGRed, compared to 13.25% for GasSaver and 7.39% for GASOL.

Beyond effectiveness, execution time analysis reveals significant differences in efficiency. As shown in Table 5, DGRed completes optimization for all 300 contracts in 239 minutes, while GasSaver requires 102 minutes but with limited support (discussed below), and GASOL takes 383 minutes – over 1.60× longer than DGRed.

DGRed successfully produces valid, optimized versions for all 300 contracts under test, while GASOL generates valid versions for only 66 contracts and GasSaver for 152. GASOL terminates execution during its transformation process in 234 cases, raising exceptions such as "*Error in computing vars*" when failing to compute specific variables or "*Error in Bytecode generation*" when producing problematic bytecode. GasSaver generates modified contract versions in all cases, but 148 of these result in compilation errors due to transformations that introduce semantic changes affecting contract behavior.

In terms of monetary impact, the savings achieved by each tool vary significantly depending on network conditions. At today's gas price of 3 gwei and ETH = $3900, DGRed's total deployment gas savings (see Section 4.2) translate into approximately $185, while GasSaver and GASOL yield only $62 and $38, respectively. Although these absolute numbers appear modest under current low network costs, the picture changes dramatically in periods of high congestion. At 200 gwei, the same optimizations correspond to more than $12.3k saved with DGRed, compared to $4.1k with GasSaver and $2.5k with GASOL. This illustrates that while savings may seem negligible in calm network conditions, they can scale to substantial economic benefits during busy periods, making DGRed especially valuable in real-world deployments.

### 4.5 RQ5: DGRed vs. Compiler Optimization Tunings

*4.5.1 Experimental Setup.* The Solidity compiler exposes two optimization-related flags that may influence deployment gas. The `-optimize` flag enables or disables the bytecode optimizer entirely; when enabled, the compiler applies constant folding, dead-code elimination, common-subexpression elimination, and peephole optimizations to the generated bytecode [13]. Notably, these transformations operate on the *instruction stream*, i.e., they simplify arithmetic expressions,

Table 5. **RQ4.** Tool support, deployment-gas reduction across 300 contracts, and equivalent monetary savings. USD values are computed with ETH = 3900 USD. "Current gwei" refers to today's gas price (3 gwei), while "High gwei" reflects a congested period (200 gwei).

| Tool | Contracts Supported | Mean Reduction | Max Reduction | Time | Total Gas Saved | Current Savings (USD) | High-Congestion Savings (USD) |
|---|---|---|---|---|---|---|---|
| DGRed | 300/300 | 15.65% | 32.72% | 239 min | 15,854,883 | $185.6 | $12,381.4 |
| GasSaver | 152/300 | 4.91% | 13.25% | 102 min | 5,251,915 | $61.5 | $4,098.3 |
| GASOL | 66/300 | 2.83% | 7.39% | 383 min | 3,264,983 | $38.2 | $2,544.8 |

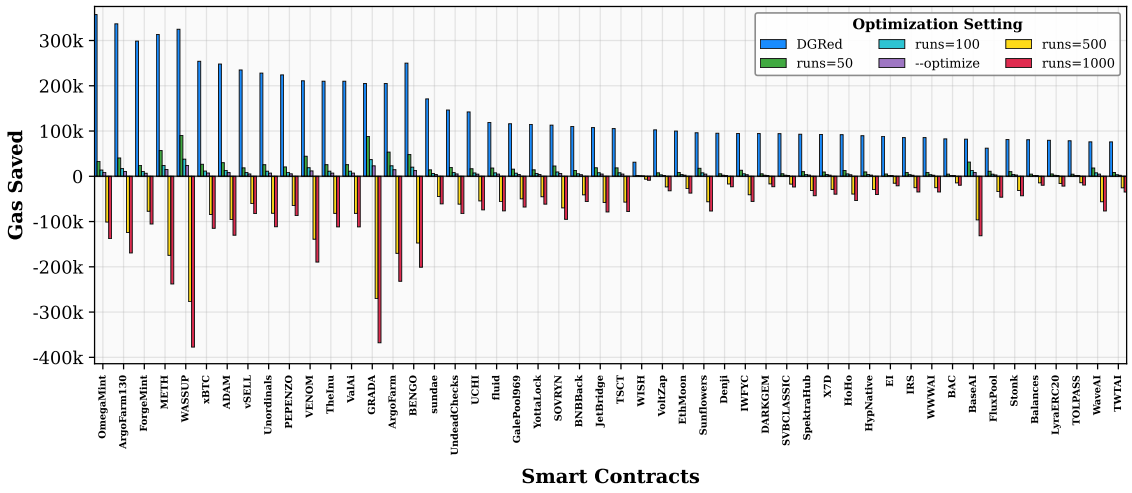



Fig. 9. **RQ5.** Per-contract deployment gas saved under DGRed, `-optimize`, and different `-optimize-runs` settings, relative to the unoptimized baseline (`optimizer = false`), for the 50 most active contracts. DGRed consistently achieves the greatest deployment gas savings. Higher `-optimize-runs` values *increase* deployment cost (negative bars), because the compiler trades smaller deployment bytecode for cheaper runtime execution.

remove unreachable code, and replace common sequences of opcodes with shorter equivalents. The `-optimize-runs` parameter further tunes the optimizer's trade-off between deployment size and runtime efficiency [20]: it indicates the expected number of times a deployed contract will be called. Low values (e.g., 50) cause the optimizer to favor smaller deployment bytecode, while high values (e.g., 1000) cause it to favor cheaper runtime execution at the expense of larger bytecode. When the optimizer is enabled, the default value of `-optimize-runs` is 200; hence, enabling the optimizer with `-optimize` is equivalent to setting `-optimize-runs = 200`. Note that the Solidity compiler ships with the optimizer *disabled* by default [20]; however, enabling it is the recommended production configuration and standard practice across all major development frameworks.

To investigate whether comparable deployment gas savings could be obtained by tuning these compiler flags without reordering state variables, we compile each of the 300 contracts in our dataset under the following configurations: (i) DGRed (with `-optimize`, `runs = 200`), (ii) `-optimize` (the recommended production configuration, `runs = 200`), and (iii–vi) `-optimize-runs` $\in \{50, 100, 500, 1{,}000\}$. As reference baseline we use compilation *without* the optimizer (`optimizer = false`). Between each setting we run `forge clean` to ensure no cached artifacts carry over.

*4.5.2 Results.* Figure 9 presents per-contract results for the 50 contracts with the highest transaction activity (also used in RQ4), while Table 6 summarizes the results across all 300 contracts.

Table 6. **RQ5.** Deployment gas reduction under DGRed, `-optimize`, and `-optimize-runs` tuning across all 300 contracts. All values relative to the unoptimized baseline (`optimizer = false`). Since `-optimize` uses `runs = 200` (the recommended production configuration), they are equivalent and reported as a single row. USD values are computed with ETH = 3 900 USD. Negative values indicate that the setting *increases* deployment gas.

| Setting | Mean Red. (%) | Max Red. (%) | Total Gas Saved | Current Savings (USD) | High-Congestion Savings (USD) |
|---|---|---|---|---|---|
| DGRed | 15.65 | 32.72 | 15,854,883 | $185.6 | $12,381.4 |
| `runs = 50` | 1.86 | 5.37 | 1,537,821 | $24.97 | $1,672.75 |
| `runs = 100` | 0.78 | 1.62 | 755,171 | $10.47 | $701.17 |
| `-optimize (runs = 200)` | 0.49 | 1.17 | 681,846 | $6.57 | $440.48 |
| `runs = 500` | −5.71 | −1.21 | −4,175,142 | −$76.85 | −$5,132.94 |
| `runs = 1 000` | −7.78 | −2.74 | −5,812,269 | −$104.46 | −$6,993.75 |

**The `-optimize` flag.** Enabling the compiler optimizer with `-optimize` reduces deployment gas by only 0.49% on average (up to 1.17%), saving a total of 681,846 gas across 300 contracts (Table 6). This negligible reduction is expected: as described in Section 2.2, these instruction-level transformations do not alter the storage layout or the sequence of `SSTORE` operations executed during constructor initialization, which is the dominant source of deployment gas consumption.

**Tuning `-optimize-runs`.** As shown in Table 6, even the most deployment-friendly setting, `runs = 50`, reduces deployment gas by only 1.86% on average (up to 5.37%), while `runs = 100` achieves a mere 0.78%. Conversely, *raising* the parameter above the default increases deployment gas: `runs = 500` increases deployment cost by 5.71% on average (adding 4,175,142 gas across 300 contracts), and `runs = 1 000` by 7.78% (adding 5,812,269 gas). Figure 9 visually confirms this pattern across all 50 evaluated contracts: DGRed consistently achieves the largest savings per contract, with a mean reduction of 17.32% (147,786 gas saved on average), while `runs = 500` and `runs = 1 000` produce negative bars (i.e. increased deployment cost) for all 50 contracts. DGRed outperforms every compiler tuning setting on every contract. This behavior is expected because `-optimize-runs` controls how aggressively the compiler inlines function bodies in the runtime bytecode [13, 20], but does not affect the constructor's `SSTORE` sequence, i.e., the dominant deployment cost factor.

**Comparison with DGRed.** DGRed achieves a mean deployment gas reduction of 15.65% (Table 6)—over 8× larger than the best `-optimize-runs` setting (`runs = 50`, 1.86%) and 32× larger than the recommended `-optimize` setting (0.49%). This gap arises because the compiler optimizer transforms the instruction stream (opcode simplification, dead-code elimination, inlining trade-offs), while DGRed optimizes the *storage layout* by reordering state variable declarations to reduce the number of `SSTORE` operations in the constructor. Since these two approaches target orthogonal aspects of the bytecode, their effects are fully complementary.

## 5 Discussion

### 5.1 Benefits Beyond Execution Efficiency

Targeting deployment gas reduction in smart contracts represents a valuable optimization strategy, even without emphasizing execution gas efficiency. Deployment costs constitute a significant one-time expense that can become prohibitive during high gas price periods or for projects with limited initial funding. This optimization particularly benefits experimental contracts and systems that require frequent redeployment. For projects with strict budget constraints, reducing deployment gas means more complex contracts can be deployed within available resources. Our approach addresses this deployment phase that other optimization techniques often overlook while pursuing execution

efficiency. Importantly, deployment gas reduction typically does not interfere with most execution optimization techniques, which focus on algorithmic improvements, function call patterns, and memory usage rather than storage layout decisions. Moreover, since DGRed only reorders state variable declarations without modifying contract logic or semantics (Property 1), it constitutes a strict improvement: developers can apply it unconditionally as a free optimization with no downside.

For development teams, reduced deployment costs accelerate testnet iteration cycles and decrease dependence on testnet faucets (e.g. ETH faucet [2], Alchemy Goerli faucet [1]). In Layer 2 solutions, where deployment economics differ significantly, targeted deployment optimization can yield greater proportional benefits. For decentralized organizations, minimizing deployment expenses directly preserves funds for other initiatives.

### 5.2 Threats to Validity

*5.2.1 Execution Gas Impact Analysis.* Our RQ3 evaluation indicates that DGRed's mutations have minimal execution gas impact, with only 6 out of 100 functions showing differences when multiple packed variables are accessed together. To understand the broader implications, we conduct a manual analysis of 335 randomly-selected open-source contracts, examining 1,509 functions and their state variable access patterns. Our analysis reveals that functions access an average of only 1.15 state variables per call. This low access density means that most functions do not benefit from variable packing's execution gas advantages, since the benefits only occur when multiple variables sharing the same storage slot are accessed within a single function call (the Solidity documentation indicates [12]). Without such simultaneous access, variable packing provides no execution gas savings and can even increase costs due to bit-masking operations. This finding explains why both DGRed and variable packing show similar execution gas costs in 94 out of 100 functions in our evaluation.

*5.2.2 Optimization Domains.* Although GASOL and GasSaver attempt to optimize both deployment and execution gas, we focus our comparison primarily on deployment gas for two reasons. First, our state variable mutation approach specifically targets deployment gas reduction, making deployment gas the most relevant metric for evaluating our contribution. Second, deployment and execution gas optimizations address fundamentally different aspects of smart contract efficiency: deployment costs are incurred once during contract creation, while execution costs occur during runtime operations. Since our approach targets variable declaration ordering without modifying execution logic, it remains orthogonal to execution-focused optimizations, allowing developers to combine both approaches when comprehensive optimization is desired.

*5.2.3 Upgradeable Contracts.* Our approach assumes that the optimized contract is deployed as a standalone, non-upgradeable contract. In upgradeable contract patterns such as the ERC-2535 Diamond Standard [63], storage slot assignments must remain consistent across contract upgrades to ensure continued access to previously stored data. Since DGRed reorders state variable declarations, it may alter storage slot assignments, potentially corrupting the storage layout expected by upgraded contract implementations. Therefore, developers using upgradeable contract patterns should not apply DGRed to contracts that will be upgraded with additional state variable definitions appended to the existing layout. This limitation does not affect the vast majority of deployed contracts, which are non-upgradeable.

## 6 Related Work

**Gas Analysis and Optimization Tools**. Different tools have been developed to optimize gas usage in smart contracts, as we discussed in Section 4.4. GASOL [24] and GasSaver [64] aim to reduce execution costs via bytecode and source

code transformations but struggle with deployment cost optimization and semantic preservation. GasChecker [39] detects gas-inefficient patterns but lacks automated optimization capabilities. In contrast, DGRed focuses on reducing deployment costs while preserving semantics, filling a key gap in current tools.

**Storage Layout Optimization**. As discussed in Sections 2.2.2 and 2.2, existing approaches focus on variable packing to reduce storage slots. The common strategy of ordering variables from smallest to largest (Section 2.3) neglects deployment-time impacts. While patterns from Marchesi et al. [60] and Khanzadeh et al. [51] offer useful storage guidance, our findings reveal a counterintuitive link between storage optimization and deployment costs, challenging established conventions.

**Smart Contract Reliability**. Ensuring smart contract correctness has motivated extensive research on automated issue detection. Static analysis tools examine code without execution: Securify [72] employs stratified Datalog for security pattern analysis, Slither [43] detects common anti-patterns, Ethainter [32] identifies composite vulnerabilities through information flow analysis to track how data moves and escalates across multiple transactions, Vandal [33] provides scalable bytecode decompilation for security analysis, and EthIR [25] offers a framework for high-level bytecode analysis. Zeus [50] combines abstract interpretation with symbolic model checking, while MadMax [46] detects vulnerabilities leading to out-of-gas conditions.

Dynamic testing complements static analysis through execution-based techniques. Fuzzing tools including ContractFuzzer [49], Harvey [73], and Echidna [47] automatically generate test inputs to explore execution paths and discover runtime issues. Symbolic execution tools reason about all possible paths: Oyente [57] pioneered constraint-based issue detection, Mythril [42] extends support to multiple blockchains, ETHBMC [44] applies bounded model checking, and Manticore [62] provides a unified framework for verification. Formal verification techniques like VeriSmart [71] and VerX [69] offer mathematical correctness proofs, though requiring user-provided specifications.

These tools focus on correctness and security, making them orthogonal to gas optimization. Contracts optimized by DGRed should undergo security analysis to ensure optimization preserves correctness properties.

**Search-Based Software Engineering and Fuzzing Optimization**. Our iterative approach is inspired by search-based software engineering and fuzzing techniques that explore large solution spaces via guided random search [48]. It aligns with modern fuzzing strategies—such as power scheduling [30, 58], coverage-guided prioritization [53, 55], distance-based search [27, 38], and adaptive mutation [30]—which use execution feedback to steer exploration. Similarly, our gas-cost feedback mechanism (Section 3.2.3) guides the reordering of state variables to minimize deployment costs and contract slots, treating them as fitness functions in the search process.

## 7 Conclusions

We introduce State Variable Mutation, a systematic approach for reducing smart contract deployment costs through guided reordering of state variable declarations. Our analysis of Solidity storage internals reveals that conventional variable packing—despite being widely recommended for storage optimization—often increases deployment gas costs due to compiler-generated masking and shifting operations. This counterintuitive finding challenges established optimization practices and demonstrates the need for deployment-specific strategies.

Our evaluation on 300 real-world smart contracts demonstrates substantial practical impact. DGRed achieves deployment gas reductions of up to 32.72% (357,297 gas units), with an average reduction of 15.65% (52,850 gas units per contract), translating to total savings of 15,854,883 gas units. During high network congestion periods (200 gwei), these savings correspond to $12,381.4 USD, demonstrating significant economic value. Importantly, our approach maintains

identical execution gas to the original contract in all 100 evaluated cases, confirming that state variable mutation does not affect runtime efficiency.

Comparative evaluation against state-of-the-art tools reveals DGRed's superior reliability and effectiveness. While GasSaver and GASOL achieve only 4.91% and 2.83% average reductions respectively, DGRed produces valid, optimized contracts for all 300 evaluated cases (100% success rate). In contrast, GasSaver generates compilation errors in 49% of contracts due to semantic alterations, and GASOL produces invalid bytecode in 78% of cases. Our approach maintains complete semantic preservation through identifier-based variable access, ensuring optimized contracts preserve exact functionality and behavior. Furthermore, comparison with Solidity's built-in compiler optimization flags shows that DGRed's 15.65% reduction is 8× larger than the best compiler tuning (1.86% with `runs = 50`), confirming that the two approaches are complementary and target different aspects of deployment gas.

From a gas optimization perspective, DGRed has no downside: it only reorders state variable declarations, provably preserves contract semantics, and can therefore be applied unconditionally as a free optimisation with no runtime trade-off and no risk of behavioural regression.

The availability of DGRed as an open-source tool enables immediate adoption by practitioners seeking cost-effective deployment optimization. By focusing specifically on deployment costs—an aspect neglected by existing tools—our approach fills a critical gap in smart contract optimization practice. The technique's orthogonal nature allows combination with execution-focused optimizations, providing developers with comprehensive gas reduction strategies without compromising contract functionality or correctness.



## References


[1] [n. d.]. Alchemy Goerli Faucet. https://www.alchemy.com/faucets/ethereum-sepolia. Accessed: 2025-03-04.
[2] [n. d.]. Ethereum Faucet. https://faucet.quicknode.com/ethereum. Accessed: 2025-03-04.
[3] n.d.. *Advance Solidity Assembly: Bit Shifting and Storage Offsets.* https://dev.to/web3_ruud/advance-solidity-assembly-bitshifting-and-storage-offsets-2mai [Last access 15/10/2023].
[4] n.d.. *Data Location.* https://docs.soliditylang.org/en/latest/types.html#data-location [Last access 12/5/2025].
[5] n.d.. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* https://ethereum.org/en/whitepaper/ [Last access 15/10/2023].
[6] n.d.. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* https://ethereum.github.io/yellowpaper/paper.pdf [Last access 15/10/2023].
[7] n.d.. *Etherscan.* https://etherscan.io/ [Last access 31/10/2023].
[8] n.d.. *EVM OPCODES.* https://www.evm.codes/ [Last access 12/5/2025].
[9] n.d.. *Forge Foundry.* https://book.getfoundry.sh/forge/ [Last access 12/5/2025].
[10] n.d.. *A Guide to Mocking and Stubbing in Software Development.* https://medium.com/@julius.prayoga/mock-stub-e7c0eaa801bf [Last access 12/5/2025].
[11] n.d.. *How to Write Gas Efficient Contracts in Solidity.* https://yos.io/2021/05/17/gas-efficient-solidity/ [Last access 15/10/2023].
[12] n.d.. *Layout of State Variables in Storage and Transient Storage.* https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html [Last access 12/5/2025].
[13] n.d.. *Optimizer — Solidity Documentation.* https://docs.soliditylang.org/en/v0.8.28/internals/optimizer.html [Last access 20/02/2026].
[14] n.d.. *Solidity Compiler.* https://github.com/ethereum/solidity/ [Last access 31/10/2023].
[15] n.d.. *State Variables in Solidity.* https://docs.soliditylang.org/en/latest/contracts.html#state-variables [Last access 12/5/2025].
[16] n.d.. *Tight Variable Packeting.* https://fravoll.github.io/solidity-patterns/tight_variable_packing.html [Last access 12/5/2025].
[17] n.d.. *Tree-sitter.* https://tree-sitter.github.io/tree-sitter/ [Last access 21/5/2025].
[18] n.d.. *Types.* https://docs.soliditylang.org/en/latest/types.html#mappings [Last access 12/5/2025].
[19] n.d.. *Understanding Gas Costs after Berlin.* https://hackmd.io/@fvictorio/gas-costs-after-berlin? [Last access 15/10/2023].
[20] n.d.. *Using the Compiler — Solidity Documentation.* https://docs.soliditylang.org/en/v0.8.28/using-the-compiler.html [Last access 20/02/2026].
[21] n.d.. *What is Smart Contract Storage Layout?* https://www.alchemy.com/docs/smart-contract-storage-layout [Last access 12/5/2025].
[22] n.d.. *Working with Contract Storage.* https://learnevm.com/chapters/evm/storage [Last access 15/10/2023].
[23] n.d.. *Writing Upgradeable Contracts.* https://docs.openzeppelin.com/upgrades-plugins/writing-upgradeable [Last access 15/10/2023].

[24] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, (TACAS '20)* (Dublin, Ireland). Springer-Verlag, Berlin, Heidelberg, 118–125.

[25] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 513–520.

[26] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 713–724.

[27] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344.

[28] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2329–2344.

[29] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344.

[30] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043.

[31] Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stephane Ducasse. 2018. SmartInspect: solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 9–18. doi:10.1109/IWBOSE.2018.8327566

[32] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 454–469. doi:10.1145/3385412.3385990

[33] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *arXiv preprint arXiv:1809.03981* (2018).

[34] Vitalik Buterin, Eric Conner O'Holleran, Rick Dudley, Matthew Slipper, Ian Borgers, and Abdelhamid Bakhta Nash. 2019. EIP-1559: Fee market change for ETH 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559. Accessed: 2025-05-28.

[35] Zhuo Cai, Soroush Farokhnia, Amir Kafshdar Goharshady, and S. Hitarth. 2023. Asparagus: Automated Synthesis of Parametric Gas Upper-Bounds for Smart Contracts. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 253 (Oct. 2023), 30 pages. doi:10.1145/3622829

[36] S. Chaliasos, M. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, Los Alamitos, CA, USA, 705–717.

[37] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. A Study of Inline Assembly in Solidity Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 165 (oct 2022), 27 pages.

[38] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2095–2108.

[39] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2021. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2021), 1433–1448. doi:10.1109/TETC.2020.2979019

[40] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[41] Timothy Clem and Patrick Thomson. 2021. Static Analysis at GitHub: An experience report. *Queue* 19, 4 (Sept. 2021), 42–67. doi:10.1145/3487019.3487022

[42] ConsenSys. 2017. Mythril: Security Analysis Tool for EVM Bytecode. https://github.com/ConsenSys/mythril GitHub repository.

[43] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[44] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2757–2774. https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[45] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (March 2012), 40–44.

[46] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 116:1–116:27.

[47] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 557–560. doi:10.1145/3395363.3404366

[48] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 45, 1, Article 11 (Dec. 2012), 61 pages.

[49] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[50] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*.

[51] Sourena Khanzadeh, Noama Samreen, and Manar H. Alalfi. 2023. Optimizing Gas Consumption in Ethereum Smart Contracts: Best Practices and Techniques. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. 300–309. doi:10.1109/QRS-C60940.2023.00056

[52] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 2123–2138.

[53] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 475–485.

[54] Chunmiao Li. 2022. Gas estimation and optimization for smart contracts on ethereum. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 1082–1086. doi:10.1109/ASE51524.2021.9678932

[55] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 533–544.

[56] Yunqi Liu and Wei Song. 2024. FunRedisp: Reordering Function Dispatch in Smart Contract to Reduce Invocation Gas Fees. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 516–527. doi:10.1145/3650212.3652146

[57] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/2976749.2978309

[58] Chenyang Lyu, Shouling Ji, Chao Zhang, Y. Li, Wei-Han Lee, Y. Song, and R. Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*.

[59] M. Zalewski. 2013. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Online accessed; 13-January-2022.

[60] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. 2020. Design Patterns for Gas Optimization in Ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 9–15. doi:10.1109/IWBOSE50093.2020.9050163

[61] Charalampos Mitropoulos, Thodoris Sotiropoulos, Sotiris Ioannidis, and Dimitris Mitropoulos. 2023. Syntax-Aware Mutation for Testing the Solidity Compiler. In *Proceedings of the 28th European Symposium on Research in Computer Security (ESORICS)*. Springer, 327–347.

[62] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. doi:10.1109/ASE.2019.00133

[63] Nick Mudge. 2020. EIP-2535: Diamonds, Multi-Facet Proxy. Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-2535 Accessed: 2026-03-05.

[64] Quang-Thang Nguyen, Bao Son Do, Thi Tam Nguyen, and Ba-Lam Do. 2022. GasSaver: A Tool for Solidity Smart Contract Optimization. In *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure* (Nagasaki, Japan) *(BSCI '22)*. Association for Computing Machinery, New York, NY, USA, 125–134.

[65] Nomic Foundation. 2023. Hardhat: Ethereum Development Environment. https://hardhat.org/ Accessed: 2025-03-04.

[66] OpenZeppelin. 2026. *OpenZeppelin Contracts – API Reference (Access Control, Governance, Proxy, Finance, Metatx, Utilities)*. https://docs.openzeppelin.com/contracts/5.x/api [Last access 11/05/2026].

[67] OpenZeppelin. 2026. *OpenZeppelin Contracts – Documentation*. https://docs.openzeppelin.com/contracts [Last access 07/05/2026].

[68] OpenZeppelin. 2026. *OpenZeppelin Contracts – Tokens (ERC-20/721/1155/4626 and Official Extensions)*. https://docs.openzeppelin.com/contracts/5.x/tokens [Last access 11/05/2026].

[69] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1661–1677. doi:10.1109/SP40000.2020.00024

[70] RareSkills. 2024. The RareSkills Book of Solidity Gas Optimization: 80+ Tips. https://www.rareskills.io/post/gas-optimization. Accessed: 2025-05-15.

[71] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2019. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. *arXiv preprint arXiv:1908.11227* (2019).

[72] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/3243734.3243780

[73] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3368089.3417064