Reducing the Gas Usage of Ethereum Smart Contracts without a Sidechain

Soroush Farokhnia, Amir Kafshdar Goharshady

Reducing the Gas Usage of Ethereum Smart Contracts without a Sidechain

Soroush Farokhnia
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
sfarokhnia@connect.ust.hk

Amir Kafshdar Goharshady
Departments of Computer Science and Mathematics
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
goharshady@cse.ust.hk

***Abstract*—To prevent DoS attacks, Ethereum assigns a fixed *gas cost* to every atomic operation in the EVM and the party who creates a transaction has to pay for its overall gas usage. While the gas model is successful in preventing DoS attacks, it causes significant costs in transaction fees. For example, in June-September 2022, the average daily gas usage of Ethereum was almost four million dollars. We propose a solution to minimize these fees by moving most of the execution of a contract off-chain and storing only the bare minimum on-chain. We then trigger an on-chain execution only if there is a disagreement between the parties to the contract, which is in turn only possible if at least one party is acting dishonestly. In such cases, our approach can identify and penalize the dishonest party by making them pay not only for the gas usage of their own function calls, but also calls made by other parties. Thus, it is game-theoretically irrational to behave dishonestly in this protocol. If all parties are rational, the total gas usage goes down significantly. Notably, our approach does not require a sidechain and works directly on the main Ethereum blockchain. We also provide extensive experiments over real-world Ethereum smart contracts, demonstrating that our protocol reduces their gas usage by 40.09%.**

***Index Terms*—Ethereum, Gas Optimization, Blockchain, Mechanism Design**

I. Introduction

Gas [1]–[3]. Since all function calls in a smart contract have to be executed by every node on the blockchain network, the system is vulnerable to DoS attacks by malicious actors who initiate the execution of a time-consuming or non-terminating function. To avoid this, each basic atomic operation is assigned a specific amount of *gas*, roughly proportionate to its real-world cost of execution for the nodes, and the caller of each function has to pay a transaction fee covering its total gas usage. This has the unfortunate unintended consequence of costing the blockchain users a huge amount of money in transaction fees [4]. For example, Ethereum users pay an average of 2,706.8 ETH or 3,938,749 USD per day in gas fees [5]. Thus, under-optimized smart contracts are a major source of unintended costs and many approaches are developed to reduce, optimize or bound the gas usage, either for particular contracts or in general [6]–[34].

Our Contribution. We present a secure and trustless solution that moves most of the computations in a smart contract off-chain and ensures constant gas usage for every function call. Our method has the following advantages:

- Our experiments demonstrate that our approach reduces the gas usage of real-world smart contracts on the Ethereum blockchain by a huge margin of more than forty percent.
- The parties to the contract are guaranteed to reach a consensus about the state of the contract at the end of its implicit off-chain execution. If no party is dishonest, then the protocol succeeds in avoiding unnecessary on-chain execution and gas costs. Otherwise, the dishonest party is identified and has to solely pay for the entire gas of the on-chain execution.
- Our protocol is entirely trustless. We do not assume that any party can be trusted to act honestly, even though dishonest actions are penalized. It is also decentralized and all users have the same privileges, putting no one at an advantage in comparison with others.

II. Our Protocol

Given a contract $\mathcal{C}$ in Solidity [35], [36], our protocol produces a "wrapper" contract $\mathcal{W}$, also written in Solidity, which includes a slightly modified version of the code of $\mathcal{C}$, as well as additional functionality. The developer should deploy $\mathcal{W}$ to the blockchain. The contract $\mathcal{W}$ has the same state variables as in $\mathcal{C}$, several new state variables for the new functionality, and for every function $\mathcal{C}.f$ there is a corresponding function $\mathcal{W}.f$ with minor changes. Additionally, $\mathcal{W}$ allows the developer to set values for the following state variables:

- A positive integer d which is supposed to be the deposit paid by each party;
- A time limit t whose use-case will become apparent in the following steps when we present the concept of challenges; and
- A limit on the amount of gas that each user might consume in calling functions of $\mathcal{C}$.

Using the values chosen by the developer, $\mathcal{W}$ provides the following functionality:

(1) *Joining.* Before interacting with $\mathcal{C}$'s functionality in $\mathcal{W}$, a party Paul $\in \mathcal{P}$ has to explicitly join $\mathcal{W}$ by calling $\mathcal{W}$.`join()` and providing a deposit of d ether.
(2) *Virtual Banking.* Given that $\mathcal{W}$ avoids running calls to $\mathcal{C}$'s functions on-chain and only keeps a record of the call requests in its ledger, any transfer of money caused by $\mathcal{C}$'s functions is also not executed on-chain. To enable the flow of currency between $\mathcal{C}$ and its parties, $\mathcal{W}$ takes on the role of a bank and lets every joined user deposit and later withdraw ether in $\mathcal{W}$. The balances used in $\mathcal{C}$'s functions will then refer to the users' account balances in $\mathcal{W}$'s internal bank, rather than their ether balance in the underlying blockchain. To avoid excessive gas usage, these $\mathcal{W}$-balances are not stored on-chain. Instead, each party in $\mathcal{P}$ keeps track of them on their own machine. So, every party to $\mathcal{C}$ knows about the $\mathcal{W}$-balances but other nodes of the network, who do not particularly care about $\mathcal{C}$

or $\mathcal{W}$, are not constantly keeping track of these values.

(3) *Ledger.* $\mathcal{W}$ keeps track of an on-chain internal ledger as a state variable. On this ledger, it stores a history of all actions on $\mathcal{C}$, i.e. the ledger is a sequence of deposits and withdrawals by the users as well as a history of the function call requests.

(4) *Depositing Ether.* Any party is able to deposit ether to $\mathcal{W}$ at any time. This adds an entry to $\mathcal{W}$'s ledger containing the amount that was deposited. Of course, this entry is added on-chain. As soon as this entry is added on the blockchain, all other participants in $\mathcal{P}$ update their off-chain version of the depositor's $\mathcal{W}$-balance.

(5) *Function Call Request.* $\mathcal{W}$ has a special function which is named `requestCall` and can be invoked by any party who wants to call any function in $\mathcal{C}$. To call $\mathcal{C}.f$, the caller creates a transaction that calls $\mathcal{W}$.`requestCall` and includes the following information:
- The name f of the function in $\mathcal{C}$ whose execution is being requested,
- A list of parameters that should be passed to f,
- The amount of money that should be paid from the caller's $\mathcal{W}$-balance to $\mathcal{C}$'s $\mathcal{W}$-balance.
- An upper-bound on the amount of gas that is allocated to be used by f,

Upon receiving the items above, instead of running $\mathcal{C}.f$ or $\mathcal{W}.f$ on-chain, `requestCall` adds an entry to the internal ledger. This entry includes all the parameters above, as well as a record of the values of all global variables [37] such as `block.number`. When a call request record is added to the ledger in $\mathcal{W}$, every party in $\mathcal{P}$ performs that function call off-chain on their own machine.

(6) *Withdrawing Ether.* The parties can decide to withdraw ether from their $\mathcal{W}$-balance to use elsewhere on the blockchain. We adopt a two-step process:
- Step 1: The party ALICE $\in \mathcal{P}$ calls $\mathcal{W}$.`requestWithdraw`(x). Here, x is the amount of the withdrawal. $\mathcal{W}$ adds a record of this request to its ledger on-chain.
- Step 2: If no other party challenges the withdrawal until more than t blocks after Step 1, then ALICE can receive her money from $\mathcal{W}$.

(7) *Challenging.* Suppose a party ALICE $\in \mathcal{P}$ requests a withdrawal of x ether from her $\mathcal{W}$-balance. The $\mathcal{W}$-balances are not explicitly stored in $\mathcal{W}$ on-chain and are instead implicit and depend on the whole sequence of operations that are recorded on the $\mathcal{W}$-ledger. These operations are all simulated by every other party BOB $\in \mathcal{P}$ off-chain on his own machine. Thus, BOB knows whether ALICE has a $\mathcal{W}$-balance of at least x. If BOB believes that ALICE's withdrawal exceeds her $\mathcal{W}$-balance, then he should challenge the withdrawal by calling $\mathcal{W}$.`challenge`(i) and providing the index i of ALICE's request in the $\mathcal{W}$-ledger.
At this point, either ALICE was dishonestly withdrawing more than her balance or BOB is maliciously trying to stop her from accessing her own money. To find out which case we are in, $\mathcal{W}$ starts an on-chain execution of all the function calls recorded on its ledger, which will inevitably identify the dishonest party, whose deposit is then used to pay for the gas usage of the entire process.

(8) *On-chain Evaluation.* On-chain evaluation is triggered only when there is a disagreement about the state of $\mathcal{C}$ and a withdrawal is challenged. Then, $\mathcal{W}$ first holds an auction to find a user who is willing to trigger the on-chain executions with the smallest gas price. This executioner then calls dedicated functions of $\mathcal{W}$ that simulate every entry in the $\mathcal{W}$-ledger and also keep track of the gas usage. Note that this simulation is performed in the environment that was present at the time of the `requestCall`. All the required parameters and global variables are already stored in the $\mathcal{W}$-ledger. In the end, the dishonest party is identified and their deposit is used to reimburse the executioner for gas fees.

III. Experimental Results

IMPLEMENTATION. We implemented our approach in Python 3 and used Slither [38], Web3Py [39] and Hardhat [40] for parsing smart contracts written in Solidity and simulating the gas usage on our machine.

BENCHMARKS AND EXPERIMENTAL SETTING. As benchmarks, we took all verified real-world smart contracts available in the Etherscan database [41] that were deployed between June 1, 2022, 00:00 UTC and October 1, 2022, 00:00 UTC. We then limited our analysis to **4,355** contracts. These were all the available contracts to which our parser and simulator were applicable. Of these, 1,025 benchmarks were extremely simple contracts, i.e. basic ERC 20 with no further functionality, for which our approach is not beneficial as they already have very low gas usage. Thus, we report results over the remaining **3,330** real-world contracts.

For each contract $\mathcal{C}$ in this list, we downloaded all the function calls to $\mathcal{C}$ that were registered on the Ethereum blockchain from the time it was deployed to October 1, 2022, 00:00 UTC. This led to a total of **327,132** transactions. We then applied our protocol and performed the exact same function call requests in the exact same order, environment, and gas price for all of our benchmarks. Finally, we compared the total gas usage incurred by our method with the actual total that was paid on the Ethereum blockchain.

OVERALL SAVINGS IN GAS USAGE. The total sum of the gas usage was initially 51,845,786,705 gas units $\approx$ 727.09 ETH $\approx$ **1,261,801 USD**. This was reduced by our approach to 31,058,348,542 gas units $\approx$ 469.86 ETH $\approx$ **819,149** USD. So, we provide a reduction of **40.09 percent**, which amounts to **442,651 USD** in just 121 days. This is a testament to the real-world utility and applicability of our method.

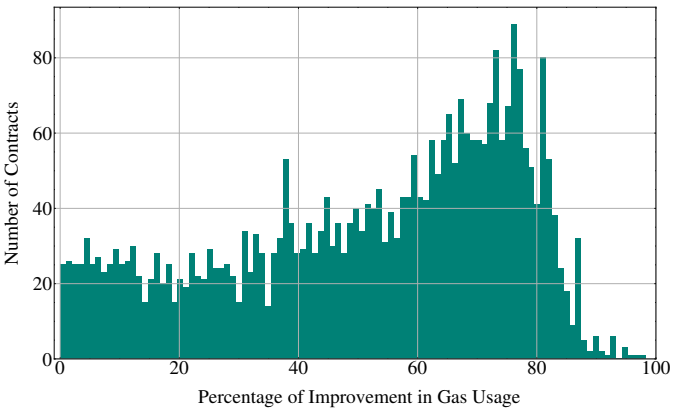

Fig. 1. Improvements obtained by our approach for each benchmark contract.

REFERENCES

[1] M. Dameron, "Beigepaper: an Ethereum technical specification," *Ethereum Project Beige Paper*, 2018.

[2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, pp. 1–32, 2014.

[3] Ethereum Foundation, "Gas and fees," 2022. [Online]. Available: https://ethereum.org/en/developers/docs/gas/

[4] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER*, 2017, pp. 442–446.

[5] CoinMarketCap, "Cryptocurrency prices, charts and market capitalizations," 2022. [Online]. Available: https://coinmarketcap.com/

[6] K. Chatterjee, A. K. Goharshady, and Y. Velner, "Quantitative analysis of smart contracts," in *ESOP*, 2018, pp. 739–767.

[7] T. Chen *et al.*, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *Trans. Emerg. Top. Comput.*, vol. 9, no. 3, pp. 1433–1448, 2021.

[8] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of super-optimized smart contracts using max-smt," in *CAV*, 2020, pp. 177–200.

[9] Q. Kong, Z. Wang, Y. Huang, X. Chen, X. Zhou, Z. Zheng, and G. Huang, "Characterizing and detecting gas-inefficient patterns in smart contracts," *J. Comput. Sci. Technol.*, vol. 37, no. 1, pp. 67–82, 2022.

[10] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, "Hybrid mining: exploiting blockchain's computational power for distributed problem solving," in *SAC*, 2019, pp. 374–381.

[11] K. Nelaturu, S. M. Beillahi, F. Long, and A. G. Veneris, "Smart contracts refinement for gas optimization," in *BRAINS*, 2021, pp. 229–236.

[12] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: gas analysis and optimization for ethereum smart contracts," in *TACAS*, 2020, pp. 118–125.

[13] Q. Nguyen, B. S. Do, T. T. Nguyen, and B. Do, "Gassaver: A tool for solidity smart contract optimization," in *BSCI*, 2022, pp. 125–134.

[14] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh, "Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities," *CoRR*, vol. abs/2112.14771, 2021.

[15] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and Y. Velner, "Ergodic mean-payoff games for the analysis of attacks in crypto-currencies," in *CONCUR*, vol. 118, 2018, pp. 11:1–11:17.

[16] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *S&P*, 2018, pp. 583–598.

[17] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan, "Gasfuzzer: Fuzzing Ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," *IEEE Access*, vol. 8, pp. 99 552–99 564, 2020.

[18] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Layer-two blockchain protocols," in *FC*, 2020, pp. 201–226.

[19] A. K. Goharshady, A. Behrouz, and K. Chatterjee, "Secure credit reporting on the blockchain," in *iThings/GreenCom/CPSCom/SmartData*, 2018, pp. 1343–1348.

[20] A. Gangwal, H. R. Gangavalli, and A. Thirupathi, "A survey of layer-two blockchain protocols," *CoRR*, vol. abs/2204.08032, 2022.

[21] A. Singh, K. Click, R. M. Parizi, Q. Zhang, A. Dehghantanha, and K. R. Choo, "Sidechain technologies in blockchain networks: An examination and state-of-the-art review," *J. Netw. Comput. Appl.*, vol. 149, 2020.

[22] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *ICSE NIER*, 2018, pp. 81–84.

[23] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *ASE*, 2018, pp. 259–269.

[24] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 116:1–116:27, 2018.

[25] C. Li, "Gas estimation and optimization for smart contracts on ethereum," in *ASE*, 2021, pp. 1082–1086.

[26] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *CCS*, 2018, pp. 949–966.

[27] Raiden Network Developers, "Raiden network documentation," 2022. [Online]. Available: https://raiden-network.readthedocs.io/en/stable/

[28] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *EUROCRYPT*, 2019, pp. 625–656.

[29] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *USENIX Security*, 2018, pp. 1353–1370.

[30] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020, pp. 9–15.

[31] A. D. Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora, "Profiling gas consumption in solidity smart contracts," *J. Syst. Softw.*, vol. 186, p. 111193, 2022.

[32] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, and A. Shakiba, "Optimal mining: Maximizing bitcoin miners' revenues from transaction fees," in *IEEE Blockchain*, 2022, pp. 266–273.

[33] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, "Probabilistic smart contracts: Secure randomness on the blockchain," in *IEEE ICBC*, 2019, pp. 403–412.

[34] P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi, "Cost analysis of nondeterministic probabilistic programs," in *PLDI*, 2019, pp. 204–220.

[35] Ethereum Foundation, "Solidity language documentation," 2022. [Online]. Available: https://docs.soliditylang.org

[36] ——, "Smart contract languages," 2022. [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/languages/

[37] ——, "Units and globally available variables," 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.15/units-and-global-variables.html

[38] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *WETSEB@ICSE*. IEEE / ACM, 2019, pp. 8–15.

[39] Python library for interacting with Ethereum, "Python library for interacting with ethereum," 2022. [Online]. Available: https://web3py.readthedocs.io/

[40] Ethereum development environment for professionals, "Ethereum development environment for professionals," 2022. [Online]. Available: https://hardhat.org/

[41] Etherscan, "Ethereum blockchain explorer," 2022. [Online]. Available: https://etherscan.io/

[42] S. Farokhnia and A. K. Goharshady, "Alleviating high gas costs by secure and trustless off-chain execution of smart contracts," in *SAC*, 2023.

The research was partially supported by the Hong Kong Research Grants Council ECS Project Number 26208122 and the HKUST Startup Grant R9272. A preliminary report of this project appeared as an extended abstract in [42].