2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 978-1-6654-9425-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/TrustCom56396.2022.00127

Reducing Gas Consumption of Tornado Cash and Other Smart Contracts in Ethereum

Jingyan Yang[‖], Shang Gao[†], Guyue Li[‖*], Rui Song[†], Bin Xiao[†]
[‖]School of Cyber Science and Engineering, Southeast University, Nanjing, China
{jingyanyang, guyuelee}@seu.edu.cn
[†]Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
{shanggao, csrsong, csbxiao}@comp.polyu.edu.hk

***Abstract*—Ethereum, the largest blockchain for running smart contracts, has been widely used, especially in financial and cryptocurrency exchange applications. Among them, Tornado Cash is a typical financial application that protects the privacy of users with anonymous transactions. However, users need to pay prohibitively high gas (transaction fees for smart contract calls) for anonymous transactions, which hinders Tornado Cash from wide applications. To address this issue, we introduced a new approach that shifts the high gas-consuming operations on smart contracts to local users. Furthermore, we use zero-knowledge proofs to ensure the operations are properly executed. The smart contract only needs to verify and update the results, which significantly reduces the gas fees of Tornado Cash. To validate our approach, we implemented a prototype and showed that our proposed method could save more than 61% of gas consumption of current operations while maintaining the privacy feature of Tornado Cash. Finally, we discussed further applications and open problems of our approach.**

***Index Terms*—smart contract, gas, Ethereum, zk-SNARK, privacy protection**

I. Introduction

The blockchain technology provides an emergent decentralized architecture for many applications. Among all blockchain networks, Ethereum [1] is the most commonly used platform to provide high programmability for various applications such as Tornado Cash [2], Polymarket [3] and Uniswap [4]. For instance, Tornado Cash is a privacy-preserving finance application that protects users' privacy with anonymous transactions. In 2022, there are more than 36,000 users and the value of deposits in Tornado Cash reach more than 3.75 billion USD [5].

In Ethereum, a deployer deploys a smart contract that supports different functions on the blockchain. Users can run these functions with specific inputs with smart contract calls. To avoid dead loops in smart contracts and prevent Denial of Service (DoS) attacks, Ethereum charges transaction fees, i.e., gas, from users who call a smart contract and makes the fees a part of the rewards to miners for executing smart contracts [6]. However, as the price of ETH (Ethereum's token) has increased significantly in the past few years, gas consumption has become a heavy burden for Ethereum users. For instance, a deposit operation will consume 60 USD of gas in Tornado Cash based on the exchange rate of Eth in June 2022. Therefore, it is very important to reduce the gas consumption of smart contracts in Ethereum.

Related Work. Some work has been proposed to address the high gas consumption issue. Chen et al. proposed adjusting gas costs dynamically to defend against DoS attacks of Ethereum [7]. Their subsequent work, GasChecker, can automatically detect gas-inefficient code in the bytecode of smart contracts [8]. In [9], Albert et al. introduce the Gasol tool, which is able to analyze and optimize the gas cost of Ethereum smart contracts. Feist et al. present Slither, which can be used for extensive analysis of smart contracts [10]. Brandstätter et al. discuss 25 optimization strategies concerning potential gas savings when being applied to Solidity smart contracts [11]. Correas et al. present a novel static profiling technique for Ethereum smart contracts, that can be used to reduce the number of resources consumed by programs [12]. However, they cannot be applied directly to reduce gas consumption of smart contracts in privacy-preserving applications such as Tornado Cash. Besides, most of these approaches can only save a small amount of gas, which indicates a very limited application. A good scheme should support both public and private applications and could fix the maximum gas fee for all kinds of operations and various inputs.

In this paper, we address the high gas consumption issues in Tornado Cash by introducing a new approach to offload high gas consumption operations on smart contracts to the local users. The user directly submits the results of these operations together with zero-knowledge proofs for the correctness of the results. The smart contract only needs to verify the proof and make updates, thus reducing the gas consumption of Tornado Cash. To summarize, this paper makes the following contributions:

- We carry out a detailed analysis on Tornado Cash and pinpoint the bottleneck of high gas-consuming operations.
- We propose a new gas-saving method based on zero-knowledge proofs for Tornado Cash, which can achieve anonymity and reduce gas consumption at the same time.
- We implement a prototype of our approach in Ganache, which results in a 61% decrease in gas costs.
- We show further applications of our approach in more generic scenarios.

The rest of the paper is organized as follows. Section II briefly introduces relevant concepts of our methodology. A detailed analysis of Tornado Cash is presented in Section III.

2324-9013/22/$31.00 ©2022 IEEE
DOI 10.1109/TrustCom56396.2022.00127

Section IV describes a method to reduce the gas consumption of Tornado Cash. Section V presents the implementation of our approach on the Ganache test network and the results. Further applications and open problems of our methodology are discussed in section VI. Finally, Section VII summarizes the paper.

II. Preliminaries

A. Notation

If Z_p is a finite field, we denote by $Z_p[X]$ the set of all polynomials on Z_p with respect to an undefined element X. We denote by aux some auxiliary information. We denote by ℓ the number of public inputs in Z_p. We use $u_i(X)$, $v_i(X)$, and $w_i(X)$ to denote three sets of polynomials. We denote by $t(X) \in Z_p[X]$ a target polynomial.

We use $P(D, M) = g^D h^M$ to denote a Pederson hash function, where D is a personalization input, M is the message that needs to be hash, g and h are generating points on a specific elliptic curve.

Definition 2.1 (Deposit Transaction). A deposit transaction takes commitment cmt as inputs, where cmt is the parameter used for zero-knowledge proof.

B. Smart Contracts

Smart contracts are automatically executable programs containing rules and conditions for the execution of any blockchain transaction [13]. In Ethereum, a developer compiles smart contracts into bytecode that can be recognized by the Ethereum Virtual Machine (EVM), and deploy the compiled smart contracts on Ethereum. Once the deployment is completed, users can call the smart contract based on the generated contract address, as shown in Figure 1. When an

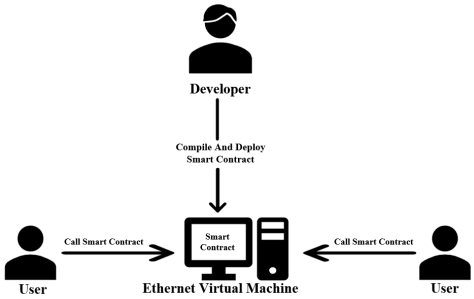

Fig. 1. The flow of smart contract operation

Ethereum node packages or verifies a transaction involving a smart contract call, it needs to call the EVM to execute the contract code to get the final result. To avoid the dead loops in smart contracts and prevent DoS attacks, the gas mechanism is designed to calculate fees for the execution of smart contracts on Ethereum. However, the gas mechanism imposes a significant additional overhead on normal users. For example, a large number of hash operations are required for deposit operations in Tornado Cash, resulting in 60 USD of gas. The detailed analysis of Tornado Cash will be presented in Section III.

C. Merkle Tree

Merkle tree is a hash binary tree structure, where the value of one node is the hash of its two leaf nodes. The root of Merkle tree is called Merkle root. Any change will result in a change in the value of all the nodes in the path from that node to the root. For example, the smart contract in Tornado Cash first constructs the Merkle tree with the default $zero\ value$ as the leaf node of a Merkle tree $leaf$. When a new commitment cmt is inserted, it replaces the original default value and becomes the new leaf. Then, the nodes on the path to the root are updated by computing hash, resulting in a new Merkle root. The Merkle tree update process is illustrated in Figure 2. Merkle tree can be used to prove that a transaction

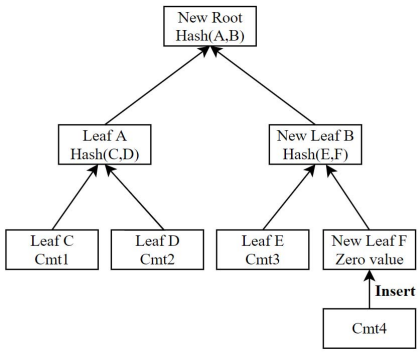

Fig. 2. Update Process in Merkle Tree

occurred but without providing transaction data. Merkle proofs need to prove whether the currently submitted data is in the tree by providing the current leaf and giving the data on the path and calculating the root. If the calculated root is the same as the given root, the proof is completed.

D. Groth16

Zero-knowledge succinct non-interactive knowledge argument (zk-SNARK) is a proof protocol that allows one party to prove a certain statement is true without revealing private information [14]. The current state-of-the-art zk-SNARK with the smallest proof size is Groth16 [15]. It allows proving a quadratic arithmetic program relation without revealing some secret information. In particular, it is a bilinear pairing-based approach with the proof size of only 3 group elements and verification with a total of 3 pairings to check. Groth16 has widely been used in today's privacy-preserving applications such as ZCash [16], Filecoin [17] and Tornado Cash.

More specifically, Groth16 works with quadratic arithmetic programs with the following relation R:

$$R = (Z_p, aux, \ell, \{u_i(X), v_i(X), w_i(X)\}_{i=0}^{m}, t(X)) \quad (1)$$

where $1 \leq \ell \leq m$ and $u_i(X)$, $v_i(X)$, $w_i(X)$ have strictly lower degree than the degree of $t(X)$ [15].

On the basis of public statements $(a_1, \ldots, a_\ell) \in Z_p^\ell$ and private witnesses $(a_{\ell+1}, \ldots, a_m) \in Z_p^{m-\ell}$ with $a_0 = 1$, a verifier needs to check the equation on group elements

$$\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) = \sum_{i=0}^{m} a_i w_i(X) + h(X)t(X) \quad (2)$$

for some degree $n-2$ quotient polynomial $h(X)$, where u_i, v_i, w_i are constants in Z_p and n is the degree of $t(X)$.

Groth16 argument consists of three parts: Setup, Prove and Verify. Initially, a trusted third party is required to call the setup algorithm to generate a common reference string consisting of an evaluation key e_k and a verification key v_k so that the prove and verify algorithms can run. The setup algorithm needs to be called only once for the same quadratic arithmetic relation. In the prove algorithm, the prover computes three elliptic curve points, A, B, and C, to represent $\sum_{i=0}^{m} a_i u_i(X)$, $\sum_{i=0}^{m} a_i v_i(X)$, and $\sum_{i=0}^{m} a_i w_i(X)$ in Equation (2) respectively as the proof. Finally, in the verify algorithm, the verifier checks whether equation (2) holds based on the proof, A, B, and C, and the common reference string.

III. Observation of Tornado Cash

In this section, we will first provide an overview of the Tornado Cash process. Afterward, we will analyze it in detail and discuss several technical challenges, as well as potential solutions to address them.

When deployed, Tornado Cash needs to be initialized first. After that, there are two major functions in Tornado Cash: deposit and withdrawal. The whole process is described as follows.

Init. Tornado Cash requires a series of arithmetic circuits, which is determined by the relation of Tornado Cash, to be built before deployment. The corresponding common reference string is generated based on the setup algorithm of Groth16, including p_k and v_k, which are used to generate proof and verify proof in the smart contract.

Deposit. Tornado Cash needs to guarantee user anonymity when calling the deposit function. To achieve this property, the Merkle tree data structure and Pedersen hash are used. Specifically, a user runs the following steps for a deposit request.

1) The user generates a nullifier $k \in Z_p$ and a randomness $r \in Z_p$ to compute $P(k, r) = g^k h^r$ as the cmt. The user initiates a deposit transaction call with the contract address left by the deployment with cmt as a parameter, after which the specified amount of Eth is transferred to the smart contract.
2) The main smart contract first maps a cmt to a Boolean variable. If the result is true, which means that the cmt has been submitted, stop the operation and return an error.
3) If the cmt has not been submitted and the Merkle tree is not full, the smart contract replaces the zero leaf with cmt as a new non-zero leaf, and updates the Merkle root accordingly. Finally, the smart contract inserts the new root into the root array.
4) After the smart contract has successfully executed the deposit, the user gets a string of characters as a note from it. The note can be used as a voucher for withdrawal by the user.

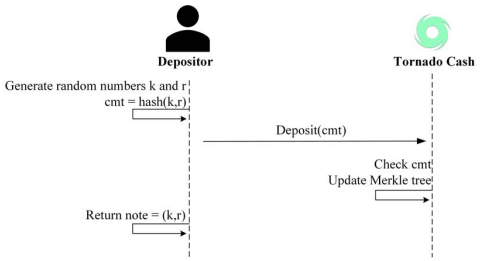

Fig. 3. The deposit workflow of Tornado Cash

Withdrawal. As for a withdrawal, the user submits a note. And after the smart contract completes the zero-knowledge proof based on the note, Eth (the amount left after deducting the fee) is transferred to the corresponding receiver.

Anonymity. Tornado Cash acts as a mixing service where all deposits transfer Eth into a smart contract. However, when withdrawals are made the user is allowed to withdraw funds from a different address and it is not required to be the funds originally deposited by the user, only proof that the user has made a deposit and has the corresponding amount of money. Once the proof is completed, the smart contract will transfer the corresponding amount of money directly to the user. With such a process, the link between the deposit and withdrawal is broken. This guarantees the anonymity of the user and the transaction cannot be traced by an external observer.

Gas Analysis. The entire process of deposit consumes 915,352 units of gas, which is about 60 USD. For the deposit, the Merkle tree needs to be updated in the smart contract. During the update process, 16 hash operations are required, which will cause a lot of gas consumption due to the complexity of hash operations and the number of operations (about 850,000 units of gas). As a result, each deposit operation will cost the user 60 USD. If we can reduce gas consumption, then we can significantly increase the number of users using Tornado Cash, thus improving the anonymity of Tornado Cash.

IV. Our Methodology

We now describe our methodology for smart contracts that result in reducing gas consumption. We show the two main parts that constitute our methodology for gas saving. The first part is to put the update operation of the Merkle tree on the local user instead of on the smart contract. The user simply submits the result (updated root) directly to the smart contract. Then, the second part is for the local user to use Groth16 proof to prove the correctness of the root. The smart contract only needs to verify the correctness of the proof and update the root. In the rest of the section, we describe the steps of our methodology in more detail and analyze the performance in comparison with previous work.

A. Shifting

To avoid the costly update operations of the Merkle tree on the smart contract, we transfer these update operations to the user and send the result to the smart contract. Accordingly, the smart contract only needs to update the root value and store it in the root array. In this way, we successfully avoid large amounts of gas consumption when updating the Merkle tree.

Before proceeding with all steps, it is necessary to initialize the process like the original process and generate the necessary parameter files to ensure that the subsequent steps are performed properly. The process works as follows.

Step1: As in the original process, the user generates cmt and passes it to the smart contract for checksumming to prevent cmt from being committed.

Step2: The user obtains the $cmts$ and the corresponding index by listening to the deposit event.

Step3: The user first constructs the Merkle tree with the obtained $cmts$. Then the user inserts the cmt and updates the Merkle tree to get the new root.

Using our method, it is possible to move the update of the Merkle tree to be performed by the local user, where the operation that generates a large amount of gas consumption in the smart contract is eliminated. Thus, we can perform gas optimization in this step of our methodology.

However, the user cannot be trusted by the smart contract as it can send an incorrect result. Specifically, the user can insert multiple deposits, d_1, and d_2 to the Merkle tree without making the corresponding transfer of d_2. If the smart contract accepts the updated root directly, the users can get d_2 with withdrawal requests. Therefore, we need the user to ensure that the updated result is correct.

B. Verification

After putting the update of the Merkle tree to the local user, then we need to prove that the update result is correct. Our solution is to adopt a Groth16 proof to verify the update result. Submitting an updated proof not only verifies the updated result and ensures the security of it. The zk-SNARK not only guarantees correct results and efficient verification, but also protects the privacy of the user. Specifically, we use the following steps to prove the root is correctly updated.

Step4: We determine the inputs and variables before generating the proof. We use the following parameters in our approach.

- $root_u$**:** The latest root from the local update.
- $root_c$**:** The root from the checksum calculation.
- $path\_e$**:** The elements of the path from the current node to the root node obtained by the user.
- $path\_e'$**:** The elements of the path calculated by the circuit.

Furthermore, given that the three polynomials with coefficients obtained by computing the curve points pi_a, pi_b, and pi_c can be publicly accessed, it generates a proof:

$$\Pi = (pi_a, pi_b, pi_c) \xleftarrow{R} Prove(p_k, (root_u, k, r, path_e), (root_c, leaf, path'_e)) \quad (3)$$

to prove the following relation:

$$root_u = root_c \wedge P(k, r) = leaf \wedge path_e = path'_e \quad (4)$$

is satisfied.

Step5: The user sends the tuple $(\Pi, root_u)$ to the smart contract.

Step6: After receiving the tuple $(\Pi, root_u)$ from the user, the smart contract takes v_k, and verifies whether Π is valid by computing $Verify(v_k, root_u, \Pi)$. If the verification is true, the smart contract updates the value of the root and stores it in the root array, which completes the deposit operation request.

Step7: The user gets a string of characters as a note, which can be used as a voucher for withdrawal.

With the verification step, we will be able to guarantee the behavioral integrity of the accomplished optimizations in the previous steps.

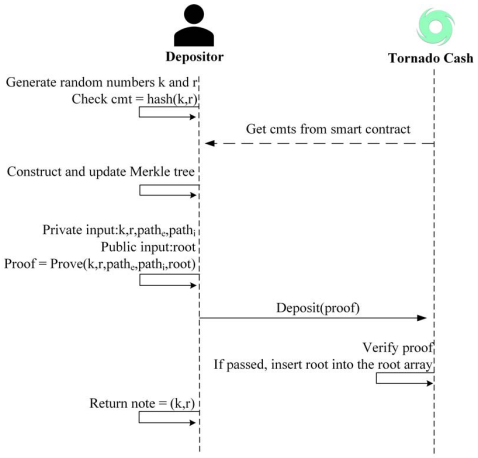

Fig. 4. Improved deposit of Tornado Cash

For the improvement of the deposit in Tornado Cash with less gas, we propose a step-by-step gas-reduction scheme, as shown in Figure 4. We move the operation that generates significant gas consumption in the smart contract to the user. The user only submits the updated result and the proof to the smart contract. After that, the smart contract verifies the proof and receives the updated result. Since our proposed method uses Groth16, the verification cost is constant regardless of the input size, therefore, the gas consumption is also constant.

Compared with the scheme proposed by Brandstatter [11], in which the saved gas cost differs significantly between different rules, our scheme can fix the maximum gas fee for all kinds of operations, which has obvious superiority in reducing gas consumption. Meanwhile, our scheme uses Groth16 to verify the update result, which also ensures the privacy of users as well as the speed of verification.

V. Experiment

In this section, we present the implementation of our methodology and the experimental evaluation results. The workstation we use for our experiments is equipped with an

Intel Core i7-12700H 2.3GHz CPU, 8GB of DDR5 RAM and 20GB SSD running Linux Ubuntu 18.04 operating system. We use the Truffle development testing framework and deploy the smart contracts to Ganache-CLI for testing.

We show that our solution can significantly reduce gas consumption and keep the time of the whole transaction process within an acceptable period. Specifically, we test the improved gas consumption, request initiation time, and request processing time. The detailed gas analysis and time analysis are presented below.

A. Gas Analysis

To clarify the relationship between gas consumption and level, we modified the height of the Merkle tree in the env file and compiled and deployed it again. Then we launch the deposit request again and analyze the transaction log in Ganache-CLI. After the improvement, we compile and deploy again after modifying the same parameter in the env file and performing the same test. The comparison of the gas consumption of the improved process and the gas consumption of the original process is shown in Figure 5.

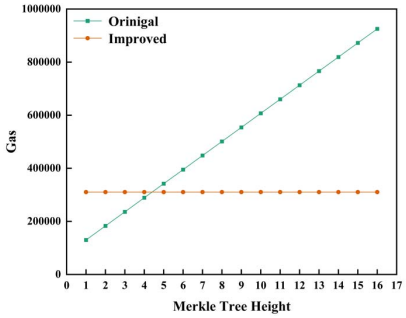

Fig. 5. Comparison of improved and original gas cost

The original process of deposit consumes up to 925,264 units of gas due to the level of hash operation in the Merkle tree update. Every time the level is reduced by 1, the gas consumption decreases by 53,102 units. There is a linear relationship between gas consumption and input size.

However, the gas cost remains around 310,000 units in the improved process regardless of changing the height of the Merkle tree. Our solution reduces 614,753 units, which are composed of three parts: the verification of the user's incoming proof (23,500 units), the update of the root array of the smart contract, and other consumption. All three components of gas consumption are fixed. We can conclude that even if we continue to increase the height of the Merkle tree or replace the hash operation with a more complex operation, the gas consumption will be fixed. Thus, our scheme successfully achieves an $O(1)$ improvement in gas cost.

B. Time Analysis

Request Initiation Time. A large part of the request initiation time is made up of the time it takes to generate the proof. The original process simply creates a deposited object and passes the commitment to the smart contract. This completed the request initiation between 350ms and 450ms.

The improved process involves creating the deposit, constructing the tree, and completing the proof. The proof generation time rises with the height of the tree, which in turn causes the improved request initiation time to rise with height. Figure 6 shows the time to generate the proof and initiate a request in the improved process.

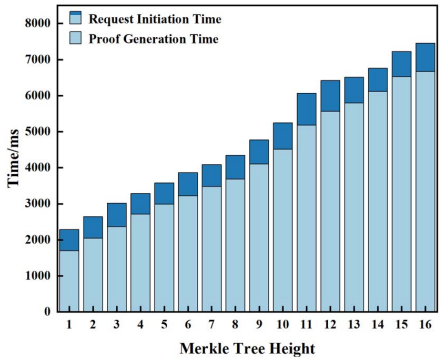

Fig. 6. The Improved Request Initiation Time and Proof Generation Time

Request Processing Time. There is a significant difference between the time it takes for a smart contract to process a request in the original process and in the improved process, as shown in Figure 7.

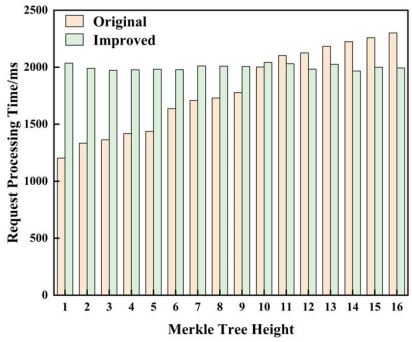

Fig. 7. Comparison of the Original and Improved Request Processing Time

The original process, after passing in the smart contract and inserting the cmt, gets the root. As the height increases, more hash is required, resulting in an increase in time. The improved process is to verify proof and store it directly into an array. The time to verify is independent of the height of the tree, and the total time is also independent of the height.

VI. Discussion

A. Extension

Our approach does not only target improvements in gas consumption in Tornado Cash but can also be extended to more generic applications. Figure 8 shows the extension of our scheme. Smart contracts that can be applied need to adhere to the following criteria.

1) The smart contract executes complex functions that cause large amounts of gas consumption. And the functions executed on the smart contract can be implemented by the user by other means.
2) After user implementation, reasonable constraints can be added. Then the operation is guaranteed to be correct by zero-knowledge proof.

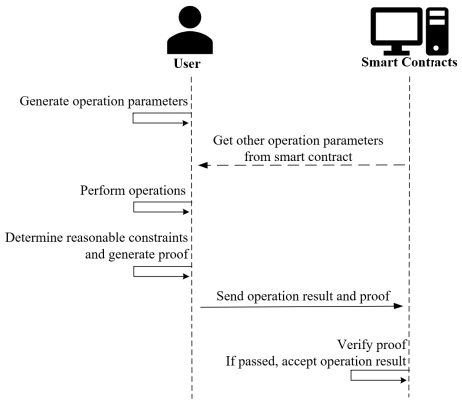

Fig. 8. Extension of our scheme

B. Open problems

Although our approach can effectively reduce gas consumption while ensuring anonymity, there are still some problems with it.

1) It is necessary for Groth16 to initialize each circuit once by a trusted setup, which means that different smart contracts need to be initialized once. Therefore, for smart contracts other than our specific one, the trusted setup needs to be re-executed.
2) Once the information in the trusted setup is compromised, the security of the entire proof system is no longer guaranteed.
3) Groth16 cannot support batched polynomial commitment. However, we can use Plonk, which needs only one trusted setup to verify multiple polynomials.
4) Our approach is targeted at smart contracts with high gas consumption. Smart contracts with gas consumption of less than 24w (gas consumption needed for the verification) do not apply to our scheme.

VII. Conclusion

In this work, we have presented a novel method for effectively reducing gas consumption by calling smart contracts on Ethereum. By moving high gas consumption operations on smart contracts to the local users, we optimize the calling of smart contracts. Furthermore, we employ zk-SNARK to solve the privacy issue of on-chain transactions. This method can save the gas cost of smart contracts while ensuring the anonymity of the calling process. We use the Truffle development testing framework to implement our approach and deploy smart contracts to Ganache-CLI for testing. After the actual deployment test, the experimental results show that the proposed method can produce significant gas cost savings for smart contracts on Ethereum. Compared to previously proposed schemes for reducing gas consumption, our scheme can fix the maximum gas fee for all kinds of operations, i.e., $O(1)$ cost, which has obvious superiority in reducing gas consumption. Meanwhile, our scheme uses zk-SNARK for verification, which takes into account the privacy of users.

Acknowledgment

This research is partially supported by HK RGC GRF PolyU 15216721, HK RGC GRF PolyU 15207522, the National Natural Science Foundation of China under Grant 62171121, the Natural Science Foundation of Jiangsu Province under Grant BK20211160 and Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201.

References

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[2] S. Roman *et al.*, "Tornado cash privacy solution," [Online], https://tornado.cash/Tornado.cash_whitepaper_v1.4.pdf/.
[3] C. Shayne *et al.*, "Polymarket," [Online], https://polymarket.com/.
[4] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson, "Uniswap v3 core," 2021.
[5] S. Roman *et al.*, "Tornado cash," [Online], https://tornado.cash/.
[6] J. Park, D. Lee, and H. P. In, "Saving deployment costs of smart contracts by eliminating gas-wasteful patterns," *International Journal of Grid and Distributed Computing*, vol. 10, no. 12, pp. 53–64, 2017.
[7] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against underpriced dos attacks," in *International conference on information security practice and experience*. Springer, 2017, pp. 3–24.
[8] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1433–1448, 2020.
[9] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: gas analysis and optimization for ethereum smart contracts," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 118–125.
[10] J. Feist, G. Grieco, and A. Groce, "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, 2019, pp. 8–15.
[11] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2020, pp. 281–290.
[12] J. Correas, P. Gordillo, and G. Román-Díez, "Static profiling and optimization of ethereum smart contracts using resource analysis," *IEEE Access*, vol. 9, pp. 25 495–25 507, 2021.
[13] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*. IEEE, 2018, pp. 1–4.
[14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct {Non-Interactive} zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 781–796.
[15] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.
[16] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.
[17] J. Benet and N. Greco, "Filecoin: A decentralized storage network," *Protoc. Labs*, pp. 1–36, 2018.