A Measurement Investigation of ERC-4337 Smart Contracts on Ethereum Blockchain

Zibin Lin
College of Electronics and InformationEngineering
Shenzhen University
Shenzhen, China
Email: linaacc9595@gmail.com

Taotao Wang
College of Electronics and InformationEngineering
Shenzhen University
Shenzhen, China
Email: ttwang@szu.edu.cn

Chonghe Zhao
College of Electronics and InformationEngineering
Shenzhen University
Shenzhen, China
Email:zhaochonghe_szu@163.com

Shengli Zhang
College of Electronics and InformationEngineering
Shenzhen University
Shenzhen, China
Email:zsl@szu.edu.cn

Qing Yang
College of Electronics and InformationEngineering
Shenzhen University
Shenzhen, China
Email:yang.qing@szu.edu.cn

Long Shi
College of Electronics and InformationEngineering
Shenzhen University School of Electronic and Optical Engineering
Nanjing University of Science and Technology Jiangsu, China
Email:slong1007@gmail.com

***Abstract*—Account abstraction is a method that enhances the flexibility and extensibility of blockchain accounts. For the Ethereum blockchain, ERC-4337 is a proposal for implementing account abstraction without modifying the logic of the underlying consensus protocol. However, due to its complete implementation through smart contracts, the transaction costs associated with ERC-4337 remain expensive compared to regular Externally Owned Account (EOA) transactions. To evaluate the usage costs of ERC-4337, we introduce two innovative algorithms: the ERC-4337 Classification Algorithm and the ERC-4337 Gas Measurement Algorithm. The Classification Algorithm categorizes historical ERC-4337 transactions and logs, providing valuable insights into their nature and characteristics. The Gas Measurement Algorithm calculates the actual gas consumption for users and the incentives paid to bundlers that package the transactions of ERC-4337 (UserOperation) into an Ethereum standard transaction. We have implemented these algorithms within the official ERC-4337 deployment on the Ethereum network. Our findings indicate that creating an ERC-4337 account costs 381,489 gas, allowing only 78 accounts per block. Furthermore, a basic ERC-4337 transfer consumes 92,901 gas, which is four times the gas cost of an EOA transfer. These results confirm that high gas fees continue to pose a significant obstacle to the widespread adoption of ERC-4337. Moreover, our proposed algorithms can serve as a valuable toolset for evaluating the usage costs associated with different account abstraction proposals on Ethereum to contribute to the assessment and improvement of account abstraction mechanisms.**

***Index Terms*—Ethereum, Account Abstraction, ERC-4337, Smart Contract, Gas**

I. Introduction

With the rapid development of blockchain technology, Ethereum has become one of the most active public blockchains [1]. Its built-in account system, consisting of Externally Owned Accounts (EOAs) and Contract Accounts (CAs), provides basic functionality for users to transfer funds and interact with smart contracts [2]. However, the Ethereum account system has limitations in private key recovery, compatibility with signature schemes, and supporting advanced account features [3].

To overcome these limitations, the concept of account abstraction has been proposed [4]. Account abstraction aims to enhance the flexibility and extensibility of blockchain accounts by decoupling core components like signatures, permission control, and gas payment from accounts. This allows accounts to be customized for particular use cases while still being interoperable with the underlying blockchain.

Over the years, numerous account abstraction proposals have emerged [5]; however, many of them necessitate modifications to the logic of the Ethereum consensus protocol, making their implementation challenging. In 2021, a notable contribution to this field was made by ERC-4337 [4], introducing a novel architectural framework that achieves account abstraction solely through smart contracts at the application layer of the Ethereum platform. This distinctive approach allows ERC-4337 to seamlessly integrate with pre-existing Ethereum networks, offering a practical solution for account abstraction without necessitating alterations to the logic of the underlying consensus protocol.

While ERC-4337 has been successfully deployed on Ethereum networks, it is important to note that its transaction costs continue to present a significant disparity compared to direct Externally Owned Account (EOA) transactions. Addressing and mitigating the usage costs associated with ERC-4337 represent key research directions for future investigations. In this paper, we propose two innovative algorithms, namely

The research is supported in part by the National Natural Science Foundation of China under grant 62171291, and in part by the Shenzhen Key Research Project under grants JSGG20220831095603007, JCYJ20220818100810023, JCYJ20220818101609021.

the ERC-4337 Classification Algorithm and the ERC-4337 Gas Measurement Algorithm, to quantitatively assess these usage costs and identify potential areas for enhancement. Specifically, the ERC-4337 Classification Algorithm enables the classification of historical ERC-4337 transactions, while the ERC-4337 Gas Measurement Algorithm facilitates the analysis of gas consumption on Ethereum networks.

To validate the effectiveness of the proposed algorithms, we conducted a series of experiments on diverse Ethereum networks spanning the duration from February 5 to June 2, 2023. The experimental findings unequivocally demonstrate the pronounced nature of the usage costs associated with ERC-4337. For instance, the creation of an ERC-4337 account incurs a substantial gas cost of 381,489 units, thereby limiting the creation of only 78 accounts per block. Furthermore, a basic ERC-4337 transfer necessitates a gas expenditure of 92,901 units, which is approximately four times the cost of an EOA transfer. These results underscore the considerable disparity in usage costs between ERC-4337 and traditional EOA transactions.

The rest of the paper is organized as follows. Section II provides background on Ethereum accounts and account abstraction. Section III gives an overview of ERC-4337. Section IV presents our proposed measurement algorithms. Section V evaluates the algorithms on the Ethereum networks. Section VI concludes the paper.

II. BACKGROUND

This section presents the background related to ERC-4337, including the Ethereum account system, blockchain wallets, and the concept and history of account abstraction in Ethereum.

A. Ethereum Account System

In Ethereum, an account system is adapted to support the functionalities of transferring crypto assets (Ethers) and invoking smart contracts for users [6]. There are two main types of accounts in Ethereum: Externally Owned Account (EOA) and Contract Account (CA) [7]. The features and functions of EOA and CA are described as follows.

EOA: EOAs can initiate transactions. An EOA can send transactions to 1) transfer Ethers to another EOA/CA; 2) trigger the logical computations of the smart contract codes which are stored under a CA. Each EOA consists of a public and private key pair and an account address. The address of the account is like a bank number to receive Ethers, the private key serves as a password to control account ownership to sign transactions, and the public key is used to verify the signatures by the private key.

CA: CAs cannot initiate transactions because each CA is not associated with a private key. But CAs store smart contract codes and data to support logical computations, and they can be invoked by EOAs to execute and update the code and data of the smart contract. Also, CAs support the message storage and transfer of Ethers or other tokens.

Although both EOAs and CAs enable users to participate in the Ethereum network, they still have certain limitations. First, EOA key management poses challenges for users. As EOAs rely on private keys to initiate transactions, the loss of the private key irrevocably prevents further usage of the account's assets [8]. Second, the Signature Algorithm of EOA is restricted to ECDSA. However, some decentralized applications may desire to utilize alternative signature schemes beyond ECDSA [9].Finally, EOAs lack support for advanced transaction functionalities such as batching transactions. Similarly, the logic of consensus protocol constraints prevents CAs from autonomously initiating token transfers or smart contract executions. Collectively, these factors motivate the development of account abstraction techniques to enhance the flexibility and extensibility of blockchain accounts on Ethereum.

B. Account Abstraction

To address the inherent constraints of Ethereum's account system, account abstraction has emerged as an effective technique to enhance the flexibility and extensibility of the account system. Account abstraction entails consolidating the two primary account types in Ethereum, Externally Owned Accounts (EOAs) and Contract Accounts (CAs), into unified Contract Accounts with the ability to initiate transactions. This innovation can transform the integrated process of transaction verification and execution into modular components that each can be adjusted according to user needs. Account abstraction is typically characterized by three key properties:

1. Cryptographic Abstraction – The property of supporting multiple other signature schemes besides the Elliptic Curve Digital Signature Algorithm implemented by the current EVM for EOAs.

2. Functional Abstraction - The property of encouraging users to achieve diverse and complex transaction functionalities according to their customized requirements.

3. Gas Abstraction – The property to enable alternative gas fee payment methods, such as to pay it by other accounts or to pay it in ERC-20 tokens.

Driven by the demand for these properties, several account abstraction proposals emerged shortly after the official launch of Ethereum's Mainnet in 2015 [10]. These proposals fall into three primary categories according to their technical features:

1. Transaction initiator abstraction (EIP-101, EIP-2938, EIP-3074, EIP-5003) - Account abstraction via transaction initiation protocol design.

2. Transaction structure abstraction (EIP-86, EIP-2718) - Account abstraction through novel transaction data structuring.

3. Transaction process abstraction (EIP-1271, EIP-4337, EIP-5189) - Account abstraction through the design of new processing in transaction packaging and execution.

We summarize these proposals of account abstraction in Table I. In the account abstraction proposals introduced from 2015 to 2021, EIP-101, EIP-86, EIP-2938, and EIP-3074 all required relatively substantial modifications to the Ethereum consensus layer, leading to some incompatibility and security

TABLE I
COLLECTION OF ACCOUNT ABSTRACTION PROPOSAL.

EIP	TITLE	ABSTRACTION	CLASSIFICATION	OVERVIEW	AUTHOR	STATE	CREATION
EIP-101	Serenity Currency and Crypto Abstraction	transaction initiator	consensus layer	The account system is abstracted, and EOA and CA are unified	Vitalik Buterin	Stagnant	2015/11/15
EIP-86	Abstraction of transaction origin and signature	transaction structure	consensus layer	Propose a new transaction type, allowing CAs to initiate transactions and pay gas as a top-level account	Vitalik Buterin	Stagnant	2017/02/10
ERC-1271	Standard Signature Validation Method for Contracts	transaction process	application layer	Provides a set of standards to verify whether the signature of the contract account is valid, enabling CAs to perform signature verification	Francisco Giordano, Matt Condon	Final	2018/07/25
EIP-2718	Typed Transaction Envelope	transaction structure	consensus layer	Proposed that when creating new transaction types, specific encodings only need to be backward compatible	Micah Zoltu	Final	2020/06/13
EIP-2938	Account Abstraction	transaction initiator	consensus layer	Upgrade the CAs to a top-level account, allowing the CAs to initiate transactions	Vitalik Buterin, Ansgar Dietrichs	Stagnant	2020/09/04
EIP-3074	AUTH and AUTHCALL opcodes	transaction initiator	consensus layer	Allow EOAs to delegate control to smart contracts so that EOAs have smart contract functions	Sam Wilson, Ansgar Dietrichs	Review	2020/10/15
ERC-4337	Account Abstraction Using Alt Mempool	transaction process	application layer	Replicate the functionality of the transaction mempool in a higher-level system	Vitalik Buterin, Yoav Weiss, Dror Tirosh	Draft	2021/09/29
EIP-5003	Insert Code into EOA with AUTHUSURP	transaction initiator	consensus layer	Based on EIP-3074, the ability to deploy code to EOA, thereby removing the security problem of private key control	Dan Finlay, Sam Wilson	Stagnant	2022/03/26
ERC-5189	Account Abstraction via Endorsed Operations	transaction process	application layer	In the mechanism of ERC-4337, the role of Endorsed is added to assume the query responsibility of Bundler to prevent DOS attacks	Agustin Aguilar, Philippe Castonguay	Stagnant	2022/06/29

issues. Therefore, they are currently stagnant or still under review. ERC-1271 and EIP-2718 have both been adopted and implemented. The standard interface of the smart contract signature verification scheme proposed in ERC-1271 was eventually adopted by the community. This signature verification standard interface also serves as an important constituting component of ERC-4337. EIP-2718 defines a standardized transaction envelope format that enables new transaction types to be added without breaking the backward compatibility. This mechanism establishes the foundation for the transaction-type extensions in the future to achieve the purpose of account abstraction.

In September 2021, ERC-4337 as a breakthrough for account abstraction was proposed by Yoav Weiss, Dror Tirosh, and V. Buterin [4]. Uniquely, ERC-4337 achieves full account abstraction solely through smart contracts, completely avoiding the modifications to the logic of the underlying consensus protocol. This advantage allows ERC-4337 to successfully deploy on Ethereum's Miannet and Testnet, becoming the first real-implemented abstracted account system for Ethereum.

After ERC-4337, EIP-5003 and ERC-5189 were proposed. EIP-5003 introduced a new opcode AUTHUSURP. Through this opcode, users can deploy smart contract code at their EOA, which will enable an EOA to hold a smart contract. ERC-5189 aimed to incorporate endorser smart contracts into ERC-4337 for verifying the validity of transactions and enhancing the security of ERC-4337.

III. DESCRIPTION OF ERC-4337

Among the various account abstraction proposals, ERC-4337 stands out as the compelete solution successfully deployed in the real-world Ethereum networks. We provide a comprehensive technical overview about ERC-4337 in this section.

ERC-4337 redefines the data structure, the validation and the packaging for the user-issued transactions on top of the underlying mechanism of Ethereum. First, users can construct a data structure called UserOperation and submit it to the UserOperation mempool. The UserOperation is a transaction data object defined by ERC-4337, containing transaction information including signature and calldata fields. The signature field can be filled with the validation data based on some predefined validation rules, and is no longer limited to using ECDSA . The calldata field can specify the Account Contract function that serves as the execution rule defined by the user. In this way, the Account Contract function included in the calldata can be invoked through the UserOperation. The

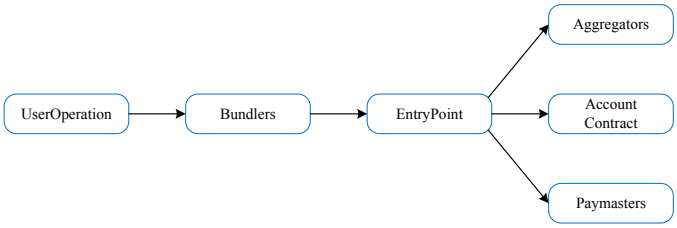

Fig. 1. All components of ERC-4337. EntryPoint acts as an intermediary to receive UserOperations submitted by Bundlers, and interact with Aggregators, Account Contracts, and Paymaser.

UserOperation mempool replicates the functionality of the Ethereum transaction mempool (where EOA transactions are stored and sorted) at a higher abstraction level to store and sort UserOperations. Once UserOperations arrive in the UserOperation mempool that is maintained by a node called bundler, the bundler will verify each UserOperation to make sure it can be successfully executed by EVM. Finally, the bundler packages a batch of valid UserOperations into a consolidated "bundle transaction", which will be signed using the EOA private key of the bundler and sent to the Ethereum network for including the bundle transaction onto the blockchain. The Gas fees of the bundle transaction are paid upfront by the bundler, who will subsequently receive compensation from the fees contained in the enclosed UserOperations.

As shown in Fig. 1, the ERC-4337 framework consists of the following core components: the UserOperation, Bundler, EntryPoint, Account Contract, Paymaster, and Aggregator.

UserOperation: This component is a structure that encapsulates transaction information such as "sender", "to", "calldata", "callGasLimit", "maxFeePerGas", "maxPriorityFee", "signature", "nonce", "payMasterAndData" and "initCode".

Bundler: These are entities that conduct off-chain verification of UserOperations from the mempool and package the valid UserOperations to construct an EOA transaction.

EntryPoint: This is a singleton smart contract serviced as an entrance on blockchain for ERC-4337. The core function within this contract is handleOps. This function allows the EntryPoint contract to monitor the verification and execution of UserOperations.

Account Contract: This is a user-owned smart contract that upholds the cryptocurrency assert, signature validation rule, contract data and execution rule of the account.

Paymaster: These are optional smart contracts employed to sponsor transaction gas fees or facilitate token payment of gas fees for the UserOperations.

Aggregator: These are auxiliary smart contracts that are trusted by accounts to validate aggregate signatures from multiple UserOperations.

As shown in Fig. 2, ERC-4337 coordinates the interaction of its core components through the following workflow:

1. The user generates and submits a UserOperation to the Bundler's mempool.

2. The Bundler performs off-chain verification on the UserOperation. If the UserOperation is valid, it will be packaged into an EOA transaction to invoke EntryPoint contract.

3. After the EOA transaction that contains the UserOperation is packaged on the blockchain by the miners to invoke the EntryPoint contract, the EntryPoint Contract will call the validateUserOp function of the Account Contract to verify the signature and pay the gas fee.

4. After the signature of the UserOperation has been verified, the EntryPoint contract calls the Account Contract to execute the function defined in callData.

5. Finally, the EntryPoint contract transfers gas fees and incentives to the Bundler's EOAs as compensation.

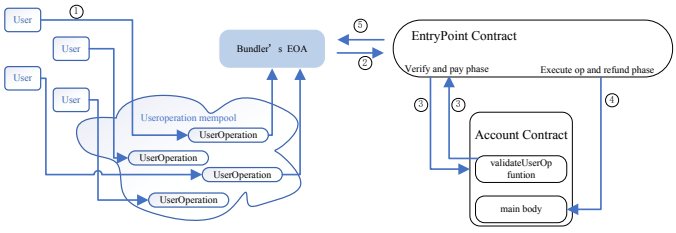

Fig. 2. The core workflow of ERC-4337. For clarity we do not show the interactions with PayMasters and Aggregators.

IV. MEASUREMENT INVESTIGATION

Quantifying the usage costs in its real-world deployments is pivotal for evaluating the viability of ERC-4337. Despite there being some ERC-4337 smart contacts deployed on Ethereum, the real-world costs associated with ERC-4337 remains unclear. To quantify these costs, we propose two novel measurement algorithms - the ERC-4337 Classification Algorithm and the ERC-4337 Gas Measurement Algorithm. The Classification Algorithm provides a detailed categorization of historical ERC-4337 transactions. The Gas Measurement Algorithm computes the actual gas expenditures for users and incentives paid to bundlers for packaging UserOperations into bundle transactions.

A. ERC-4337 Classification Algorithm

The Classification Algorithm extracts historical bundled transactions that invoked the handleOps function of the EntryPoint contract from Ethereum transactions. Furthermore, it categorizes these transactions based on their log topic.

In Ethereum, transactions will generate transaction logs after invoking a special type of functions called event [11] in smart contracts. Using the parsing function in web3.js [12], we can extract three fields from a transaction log: *log.txhash*, *log.topic0*, and *log.data* [11]: 1) *log.txhash* represents the hash of the transaction; 2) *log.topic0* records the identifier of the event; 3) *log.data* is the part of the output of the event, containing information of the parameters of the event.

Within the EntryPoint contract, various events are triggered upon the completion of tasks. When the *handleOps* function processes a UserOperation, it emits an event called *UserOperationEvent* after the UserOperation has been handled. For UserOperations that include instructions to create a new Account Contract, the EntryPoint contract emits an event named *AccountDeployed*. Additionally, when EntryPoint receives a transfer from an Account Contract, it emits an event named Deposited. The funds received are then utilized to cover the gas required for executing the UserOperation and incentivizing the bundler. The names of these events are recorded in the *log.topic0* field. The Classification Algorithm utilizes the information in *log.topic0* to categorize transactions that include UserOperations into multiple types.

The inputs of the algorithm consist of the address of the EntryPoint contract and historical transaction data from the Ethereum network. The algorithm produces two arrays that classify transactions and logs based on *log.topic0* identifier codes. These identifiers adhere to ERC-4337 definitions and categorize transactions and logs into various types such as *UserOperationEvent*, *AccountDeployed*, *Deposited*, and so on.

The Classification Algorithm calculates the selector of the *handleOps* function and assign it to variable *handleOps*, which is derived from the first four bytes of the hash of its function name and parameters. When executing smart contracts on the Ethereum network, designated function and its input parameters must be declared in the transaction's data field [13]. The function selector is stored in the first four bytes of the transaction's data field. By comparing the first four bytes of the data field in historical transactions with the computed selector of *handleOps*, the algorithm can filter out transactions that invoke the *handleOps* function of the EntryPoint contract within the collection of historical transactions.

In the subsequent processing stage, the transaction *tx* in T the inputed set of historical Ethereum transactions, needs to satisfy two conditions in order to be retained. Firstly, *tx.to* must be equal to the address of the EntryPoint contract. For Ethereum transactions that invoke smart contracts, *tx.to* field stores the address of the called smart contract. Secondly, the first four bytes of the transaction's data field, *tx.data*, must equal the variable *handleOps*. By applying these two filtering conditions, the algorithm ensures that *tx* invokes the handleOps function of the EntryPoint contract.

Based on the aforementioned conditions, the algorithm filters out the logs generated during the execution of the *tx* from the Ethereum transaction history logs. Ultimately, the algorithm organizes the extracted transactions and their corresponding logs into multiple arrays denoted as *Tx[Identifier]{t1, t2,...}* and *Log[Identifier]{l1, l2,...}*. The Identifier corresponds to the *log.topic0* value, enabling the categorical storage of transactions and logs into categories such as *UserOperationEvent*, *AccountDeployed*, *Deposited*,

and others. Algorithm 1 summarizes the ERC Classification Algorithm.

Algorithm 1 ERC-4337 Classification Algorithm

Require: $Transactions\ T = \{t_1, t_2, \ldots, t_n\},\ Logs\ L = \{l_1, l_2, \ldots, l_m\},\ EP\_address$
$handleOps \leftarrow$ hash($handleOps$(userOperation[], address))
for $i \leftarrow 1$ to m **do**
  if t_i.to $== EP\_address$ **then**
    txType $\leftarrow$ ti.data[1 : 4]
    if txType == handleOps[1 : 4] **then**
      Select l_j from LI where l_j.tx_hash $== t_i$.hash
      {
        eventTopic $\leftarrow l_j$.topic0
        Tx[eventTopic]{...} $\leftarrow t_i$
        Log[eventTopic]{...} $\leftarrow l_i$
      }
    end if
  end if
end for
return $Tx[Identifier]\{...\},\ Log[Identifier]\{...\}$

B. ERC-4337 Gas Measurement algorithm

Building upon the Classification Algorithm's output, the Gas Measurement Algorithm assesses usage costs for specific UserOperation types essential for task completion.

We can select the transactions *Tx[Identifier]{t1, t2,...}* and logs *Log[UserOperationEvent]{l1, l2,...}* from the arrays outputted by the Classification Algorithm, based on the target Identifier. These two arrays, along with a set of senders *S_addresses*, serve as inputs to the algorithm. By filtering the target set of senders, we can obtain the transactions that include UserOperation initiated by a certain type of Account Contract. This allows for a precise calculation of the costs associated with UserOperation for a certain type of Account Contract.

First, the Gas Measurement Algorithm calculates the minimum length of the data field [13]of all transactions in *Tx[Identifier]{t1, t2,...}*, and assigns the value of the length to variable *len*. This minimum length helps us identify transactions that only contain a single UserOperation and where the UserOperation triggers a single functionality, such as account creation or ETH transfers. We refer to these transactions as "transactions with a single and simple UserOperation." The reason behind this approach is that for transactions, the larger the data field is, the more UserOperations and tasks the transaction is expected to perform. *Tx[Identifier]{t1, t2,...}* records all the transactions that have accomplished the target tasks, and by filtering out the transaction with the minimum length of field, we can obtain a set of transactions with single and simple UserOperation.

For the transaction *tx* that belongs to the inputed set of transactions *Tx[Identifier]{t1, t2,...}*, the algorithm will extract bytes 166 to 185 from the and assign them to the parameter "Sender". Due to the nature of the *handleOps* function, the UserOperation serves as a parameter and has a fixed position within the . Therefore, the parameter "sender" within the UserOperation structure also has a fixed position within The position of "sender" is located in the byte segment 166 to 185 of the . Subsequently, the algorithm will check whether the parameter "sender" exists in the target set *S_addresses*. If the answer is not, the algorithm will choose the next transaction in *Tx[Identifier]{t1, t2,...}* and repeat the previous steps.

When the processed transaction meets the requirement for the sender of UserOperation, it must also satisfy another requirement, the length of should be equal to the minimum length, ensuring that *tx* is a transaction with single and simple UserOperation. Once these conditions are met, the algorithm will search for a log in *Log[UserOperationEvent]{l1, l2,...}* whose *log.txhash* is the same as *tx.hash*. This makes sure that the log is generated by *tx*. Because the log belonging to *Log[UserOperationEvent]{l1, l2,...}*, they are emitted by the event *UserOperationEvent* and have a fixed output data format. The segment of log.data from 251 to 257 bytes records the actual gas consumed by the UserOperation, including the incentive paid to the bundler, referred to as *actualGas*. Additionally, transaction's gas_used field records the gas consumed by the transaction excution in EVM. By subtracting *actualGas* from *tx.gas_used*, we can obtain the incentive paid to the bundler. Finally, to obtain a more accurate value, we sum up and average every *actualGas* and *tx.gas_used*, and then output the final result. Algorithm 2 summarizes the ERC-4337 Gas Measure Algorithm.

Algorithm 2 ERC-4337 Gas Measure Algorithm

Require: $Tx[Identifier]\{tx_1, tx_2, \ldots, tx_n\}$,
$Log[UserOperationEvent]\{log_1, log_2, \ldots, log_m\}$,
$S\_address = \{a_1, a_2, \ldots, a_k\}$
$len \leftarrow \min\{\text{length}(tx.\text{data})\}$
$actualGasSum \leftarrow 0$
$TxGasSum \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  Sender $\leftarrow$ tx$_i$.data[166 : 185]
  if $Sender \in S\_address$ **then**
    if length(tx$_i$.data) == len **then**
      Select log From Log
      {
        actualGas $\leftarrow log$.data[251 : 257]
        actualGasSum+ = actualGas
        TxGasSum]+ = tx.gas_used
      }
    end if
  end if
end for
$gasUsed \leftarrow \frac{\text{actualGasSum}}{n}$
$incentive \leftarrow \frac{\text{actualGasSum} - \text{TxGasSum}}{n}$
return $gasUsed,\ incentive$

V. EXPERIMENTAL RESULTS

With the key purposes of demonstrating the validity of the algorithms, and furnishing developers with empirical insights into real-world ERC-4337 costs, we conducted an experiment which exploited the Classification Algorithm and the Gas Measurement Algorithm to process the historical transactions of the Ethereum network. This experiment the with ERC-4337 network activity and gas consumption across the targeted Account Contract types.

A. Activity of ERC-4337 in different Ethereum networks

Firstly, this experiment utilized the ERC-4337 Classification Algorithm to extract and categorize the historical transactions and logs from the official ERC-4337 project that deployed by the Ethereum Foundation [4]. Notably, the historical transactions data entered is as of June 2, 2023, at 13:12. Activity metrics are showed in TABLE II across the Mainnet and Goerli Testnet which helps developers to test and validate the functionality and performance of Ethereum applications:

TABLE II
ACTIVE DATA OF ERC-4337 ON ETHEREUM NETWORKS.

Network	Total Account Deployed	Total UserOperation Handled	Gas Used Per Day	Account Active Per Day
Main Net	10	86	434,902	1
Goerli	1,255	6,519	21,125,791	21

As shown in TABLE II, during the specified timeframe, the official ERC-4337 project deployed a total of 10 Account Contracts on Mainnet and handled 86 UserOperations. On average, only one Account Contract remained active per day as well as consuming 434,902 gas. In contrast, on Goerli Testnet, 1,255 Account Contracts were deployed and 6,519 UserOperations were handled. On average, 21 Account Contracts were active per day, with a daily gas consumption of 21,125,791. This comparison demonstrates that the ERC-4337 project, officially deployed on Goerli Testnet, exhibits greater activity and generates a larger volume of data compared to the Mainnet. The accuracy of the ERC-4337 Gas Measurement algorithm's measurement results is directly proportional to the data size. Consequently, to ensure the reliability of the measurement results presented in this paper, only the measurement results for the project deployed on the Goerli Testnet are presented in the subsequent sections.

B. ERC-4337 in Goerli

The activity of the official ERC-4337 on Goerli Testnet is characterized through the following figures, which exhibit insights into inherent correlations between user behaviors and gas usage. Furthermore, we figure out the gas cost for kinds of UserOperation by the Gas Measurement Algorithm, which were showed in the TABLE III .

Remarkable synchrony emerges across key activity indicators. Fig. 3 illustrates the daily count of UserOperations included in the blocks on the Goerli network; Fig. 4 presents the daily count of newly deployed account contracts in the network; Fig. 5 displays the daily count of active account contracts in the network, which initiated UserOperations to the network; Fig. 6 depicts the daily gas consumption for processing UserOperations on the network. As depicted in Fig. 3, the amount of daily UserOperations handled reveals pronounced fluctuating patterns, with periodic peaks and troughs. The amount of daily Account deployment shown in Fig. 4, daily active accounts shown in Fig. 5, and daily gas spent shown in Fig. 6 similarly display synchronized periodic variations, with collective high activity from mid-March to mid-April.

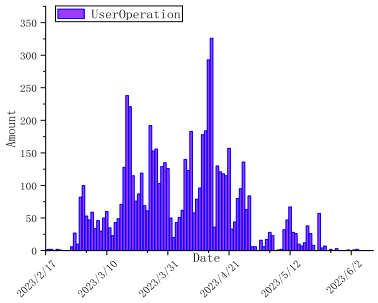

Fig. 3. The number of UserOperations handled on Goerli.

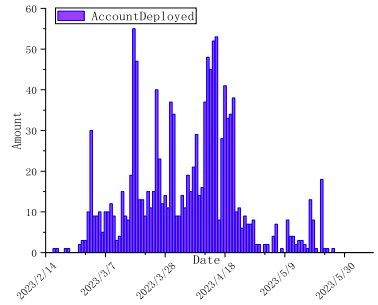

Fig. 4. The number of ERC-4337 Accounts deployed on Goerli.

By scrutinizing synchronized fluctuations across activity indicators, we can observe evident user disengagement following the completion of the Account Contract deployment. This is deduced from 1) As shown in Fig. 3 and Fig. 4, in the early stages, the number of Account Contracts deployment and the number of handled UserOperations showed synchronous fluctuations. However, in the later stage, as a substantial number of Account Contracts were already present on the network, the daily number of processed UserOperations did not exhibit a significant increase compared to the previous period; 2) As shown in Fig. 5, the number of active accounts per day is actually at a lower level after the peak period of account deployment; and 3) As shown in Fig. 6, gas spent perday remains at a relatively low level in the later stages.

Fundamentally, as shown at the TABLE III, the high UserOperation costs are likely the main reason for Account Contracts inactivity after deployment, as it hinders frequent usage.

The experiment further calculated the costs of kinds of UserOperation. There are various types of Account Contracts.

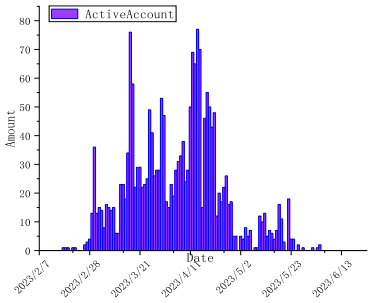

Fig. 5. The number of ERC-4337 Accounts active on Goerli.

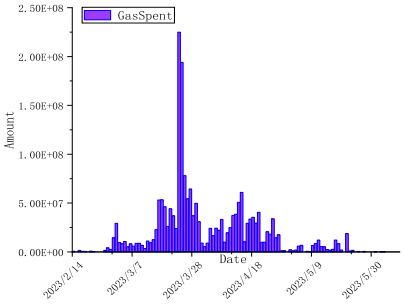

Fig. 6. The quantity of Gas spent for ERC-4337 on Goerli.

For the experiment, the chosen subject was the Account Contract type known as SimpleAccount. SimpleAccount exemplifies a minimal Account Contract implemented in ERC-4337 using ECDSA for signature verification. We collected a set of addresses for SimpleAccount Contracts and used this set as the input parameter *S_addresses* for the ERC-4337 Gas Measurement Algorithm. Consequently, TABLE III summarizes measured data from the ERC-4337 Gas Measurement Algorithm, providing gas costs data for most kinds of UserOperations.

TABLE III
GAS CONSUMPTION OF USEROPERATION.

Type	Actual Gas Used	Transaction Gas Use	Incentive (Gas)	Amount In One Block
Account Creation	381489	378204	3285	78.66
Transfer With Deposit	92901	89967	2934	322.92
Transfer Without Deposit	83984	81315	2669	357.24
EOA Account Creation	0	0		∞
EOA Simple Transfer	21000	21000		1428.60

The derived measurements demonstrate that the gas costs of the ERC-4337 UserOperations are quite high compared to the transactions of native EOAs. Creating an EOA is cost-free, only necessitating off-chain generation of a public-private key pair and deriving the account address from the public key. In contrast, creating an ERC-4337 SimpleAccount through UserOperation consumes a tremendous 381,489 gas units. Consequently, within the current per-block gas limitt [14], at most only 78 abstract accounts can be created in one block. Similarly, a simple transfer (only perform Ether transfer without calling any contract) UserOperation of SimpleAccount expends 92,901gas, approximately quadruple the cost of a simple transfer transaction of EOA. Evidently, the high gas fees present a significant barrier to widespread adoption of ERC-4337.

VI. CONCLUSION

ERC-4337 is the one of account abstraction proposals for Ethereum, which overcomes limitations of the native account system without modifications to the underlying consensus protocol logic. However, UserOperations costs remain a pivotal factor influencing the broader adoption of ERC-4337.

In this paper, we have introduced two novel algorithms providing a comprehensive toolset to compute the on-chain gas costs incurred by UserOperations. Our empirical results reveal that creating an ERC-4337 SimpleAccount by UserOperation consumes 381,489 gas units, permitting only $\tilde{7}8$ accounts per block given current limits. Moreover, a simple transfer UserOperation of SimpleAccount costs 92,901 gas, approximately four times the expense of a simple transfer of EOA.

These quantitative findings confirm that steep gas fees pose a significant impediment to widespread ERC-4337 adoption. Accordingly, future research should focus on designing strategies to optimize costs without compromising Ethereum's security and integrity.

REFERENCES

[1] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.
[2] P. Kasireddy, "How does ethereum work, anyway," *Medium*, 2017.
[3] P. Praitheeshan, Y. W. Xin, L. Pan, and R. Doss, "Attainable hacks on keystore files in ethereum wallets—a systematic analysis," in *Future Network Systems and Security: 5th International Conference, FNSS 2019, Melbourne, VIC, Australia, November 27–29, 2019, Proceedings 5*. Springer, 2019, pp. 99–117.
[4] V. Buterin, Y. Weiss, K. Gazso *et al.*, "Erc-4337 account abstraction using alt mempool," https://eips.ethereum.org/EIPS/eip-4337, accessed September, 2023.
[5] "Account abstraction," https://ethereum.org/en/roadmap/account-abstraction/, accessed September, 2023.
[6] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[7] @jmcook1186, "Ethereum accounts," https://ethereum.org/en/developers/docs/accounts/, accessed August, 2023.
[8] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, pp. 36–63, 2001.
[9] J. Na, H.-Y. Kim, N. Park, and B. Seo, "Comparative analysis of schnorr digital signature and ecdsa for efficiency using private ethereum network," *IEIE Transactions on Smart Processing & Computing*, vol. 11, no. 3, pp. 231–239, 2022.
[10] "Ethereum improvement proposals," https://eips.ethereum.org/, accessed September, 2023.
[11] "Anatomy of smart contracts," https://ethereum.org/en/developers/docs/smart-contracts/anatomy/#events-and-logs, accessed September, 2023.
[12] "how to decode log event of my transaction log?" https://ethereum.stackexchange.com/questions/87653/how-to-decode-log-event-of-my-transaction-log, accessed September, 2020.
[13] "The data field," https://ethereum.org/en/developers/docs/transactions/#the-data-field, accessed July, 2023.
[14] "The ethereum blockchain explorer," https://etherscan.io/, accessed September, 2023.