Alleviating High Gas Costs by Secure and Trustless Off-chain Execution of Smart Contracts

Soroush Farokhnia, Amir Kafshdar Goharshady

Alleviating High Gas Costs by Secure and Trustless Off-chain Execution of Smart Contracts

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

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

ABSTRACT

Smart contracts are programs that are executed on the blockchain and can hold, manage and transfer assets in the form of cryptocurrencies. The contract's execution is then performed *on-chain* and is subject to consensus, i.e. every node on the blockchain network has to run the function calls and keep track of their side-effects including updates to the balances and contract's storage. The notion of *gas* is introduced in most programmable blockchains, which prevents DoS attacks from malicious parties who might try to slow down the network by performing time-consuming and resource-heavy computations. While the gas idea has largely succeeded in its goal of avoiding DoS attacks, the resulting fees are extremely high. For example, in June-September 2022, on Ethereum alone, there has been an average total gas usage of 2,706.8 ETH ≈ 3,938,749 USD per day. We propose a protocol for alleviating these costs by moving most of the computation off-chain while preserving enough data on-chain to guarantee an implicit consensus about the contract state and ownership of funds in case of dishonest parties. We perform extensive experiments over 3,330 real-world Solidity contracts that were involved in 327,132 transactions in June-September 2022 on Ethereum and show that our approach reduces their gas usage by 40.09 percent, which amounts to a whopping 442,651 USD.

CCS CONCEPTS

• **Computer systems organization** → *Peer-to-peer architectures.*

KEYWORDS

Blockchain, Smart Contracts, Gas Optimization, Ethereum

ACM Reference Format:
Soroush Farokhnia and Amir Kafshdar Goharshady. 2023. Alleviating High Gas Costs by Secure and Trustless Off-chain Execution of Smart Contracts . In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3555776.3577833

1 INTRODUCTION

Transactions [23]. To enable smart contract functionality, Ethereum-like blockchains support a wider notion of transactions than Bitcoin. More explicitly, a transaction is no longer limited to transferring money between people, but it can also: (i) Deploy a contract's code on the blockchain, so that everyone knows about the code and the code is then immutable; or (ii) Call a contract's function, providing its parameters. See [17] for a more detailed treatment and [18] for examples. Moreover, each contract has its own address that can be used for sending money, in the form of the base cryptocurrency, e.g. Ether, to it. One can also send money to a contract at the same time as calling one of its functions. All nodes in the network keep track of the blockchain, which includes a specific order of all transactions [10]. Therefore, it is easy to reach consensus about the state of variables in every contract. We say that a computation is performed *on-chain* if it has to be executed by all the nodes.

Gas [10]. Given that every function call has to be executed by every node of the network in order to reach a consensus about the state of the contracts and the balances of each person/contract, the whole system is vulnerable to a DoS attack such as calling a function with infinite runtime. To defend against such attacks, 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 originator of each function call has to pay a transaction fee covering the overall gas usage of the call. This fee is paid to the miner, not to every node. This is so that the miners are incentivized to solve the proof-of-work problem or its variants [6]. Indeed, the miners aim to maximize their transaction fee payoffs [21]. Although using gas has been successful in deterring DoS attacks, it has had the unfortunate unintended consequence of costing the blockchain users a huge amount of money in transaction fees [9] and has also been a source for many vulnerabilities [3, 5, 8]. The Ethereum Foundation admits the problem of high gas fees in its official documentation [12]. Due to these prohibitive costs, many smart contracts are constrained to simple programs with limited functionality.

Our Contribution. We present a secure and trustless solution that moves most of the computations in a smart contract off-chain and ensures only a small $O(1)$ gas cost for every function call. Moreover, it saves enough implicit data on the blockchain to be able to reconstruct the execution and final state of the contract. Our method has the following advantages:

- Our experimental results on 3,330 real-world smart contracts and 327,132 real-world transactions during June-September 2022 on Ethereum show that our approach significantly lowers gas usage by **40.09 percent**, which amounts to 469.86 ETH or **442,651 USD**.
- Assuming that all contract parties are honest, they are guaranteed to reach a consensus about the state of the contract at the end of its implicit off-chain execution, and the protocol will succeed without any extra costly on-chain computations. If a party or parties are

SAC '23, March 27-March 31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03.
https://doi.org/10.1145/3555776.3577833

dishonest, our protocol simulates the run of the contract on-chain. It ensures that the dishonest party is always identified and charged for the gas. Thus, our protocol possesses the following desirable properties: (a) all parties are strictly incentivized to be honest[1], and (b) honest participants have no risk of being penalized.
- Our protocol is in principle applicable to virtually all smart contracts on any programmable blockchain, regardless of the language used to program them.

2 PRELIMINARIES

Smart Contracts. Ethereum smart contracts are stored on the blockchain in a stack-based assembly-like format called EVM (Ethereum Virtual Machine) bytecode [10]. Various high-level languages are compiled to this bytecode, such as Solidity [14], which is a strongly-typed language loosely inspired by Javascript, and Vyper [15], which is similar to Python. Solidity is currently the most widely-used language for writing smart contracts in Ethereum [4, 13]. Each contract consists of a number of functions. A function can be public, meaning that it can be invoked by anyone who creates a transaction calling this function, as well as other smart contracts, or it can be private to the contract. There are two types of space for storing data in a contract: (i) a *memory* which is the working space of the contract and is erased after each transaction, and (ii) a persistent *storage* whose contents are not erased in between transactions and are permanently stored by all nodes on the network. Accordingly, there are three types of variables in a Solidity smart contract:
- *State variables* are the ones saved in the storage. The values of these variables collectively define the *state* of the contract. Every node on the blockchain keeps track of storage variables.
- *Local variables* are available only during the execution of a single transaction, then discarded when it is ended.
- *Global variables [10]* are special variables that provide information about the current block and the state of the blockchain.

Ethereum's Gas Model [23]. On Ethereum, executing each EVM bytecode operation costs a well-defined number of units of *gas*. See [10, Appendix A] for a complete table of gas costs. Generally, operations on storage are much costlier than memory. When a user initiates a transaction, she can set two values: (a) the maximum amount g_m of gas that she is willing to pay for, and (b) the price π, in ether, that she is willing to pay for each unit of gas. Based on these, a miner can choose which transactions to include in her block and in which order. When the transaction is included in a block and mined, every node on the blockchain executes it. This execution begins by taking a deposit of $g_m \cdot \pi$ from the initiator's account and then running the called function while keeping track of the total gas used until this point. There are three cases: (i) If the function terminates using g units of gas and $g \leq g_m$, then the miner is paid a transaction fee of $g \cdot \pi$ and the rest of the deposit, i.e. $(g_m - g) \cdot \pi$, is reimbursed to the initiating user. Moreover, any updates to the storage are saved by all nodes on the blockchain; (ii) If the function throws an exception/error, then all of its effects are reversed and the storage is returned to its status before this function call and the gas deposit is divided exactly as in the last case; (iii) Otherwise, an out-of-gas exception is triggered, causing the user to lose her deposit, which is paid to the miner in its entirety, and the storage state of all affected contracts to revert to right before the current transaction. Out-of-gas errors are a common and serious security vulnerability in smart contracts [1, 2]. However, avoiding them is an orthogonal issue and our approach ensures the exact same behavior in the optimized low-gas version of the contract as in the original. Blocks in Ethereum can use a maximum amount of gas called blocksize [10]. The limit can change based on the demand in the network but is at most 30 million units of gas per block.

[1]More formally, the only Quasi-strong Nash equilibrium [7] is when every participant is honest.

3 OUR PROTOCOL

Fundamental Idea. The main idea behind our approach is quite simple and elegant. Let $\mathcal{P}$ be the set of parties who intend to interact with the contract C. Then, it suffices to ensure that all members of $\mathcal{P}$ are in agreement about the current state of C and there is no need to force every other node in the network to constantly keep track of C's state, as well. Such an agreement in $\mathcal{P}$ can be obtained off-chain by storing only a small amount of information on-chain. More specifically, if we store C's code and a list of all function calls on-chain, without actually executing the function calls, then (i) we avoid paying gas fees for the execution of function calls, (ii) any member of $\mathcal{P}$ can execute all the function calls in the right order off-chain, i.e. on her own machine, and they will all reach the same final state, and (iii) this final state is uniquely determined by the information that is stored on the blockchain. So, if a party Alice $\in \mathcal{P}$ is dishonest, we can run all the function calls on-chain, identify the dishonest party Alice, and penalize her. Our idea above is similar to the concept of lazy evaluation [19] in programming languages theory [20]. However, in the context of smart contracts, we take lazy evaluation to the extreme and apply it (i) only to on-chain computations, and (ii) at the level of function calls. Hence, we opt for a lazy approach that triggers an on-chain execution only when two parties in Alice, Bob $\in \mathcal{P}$ have a disagreement about the current state.

Our Protocol. Our protocol is implemented as a "wrapper" contract $\mathcal{W}$, also written in Solidity, which includes a slightly modified version of the code of C, as well as additional functionality. The developer should deploy $\mathcal{W}$ to the blockchain, instead of directly deploying C. Obtaining $\mathcal{W}$ from C is a well-defined algorithmic process and we provide a free and open-source tool that performs this task (See Section 4). More specifically, for every function $C.f$ 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 upon its deployment on the blockchain:
- The deposit d that each user should put down to ensure the gas costs of on-chain execution can be billed to this user in case of dishonest behavior;
- A positive integer t used as a time limit for challenges, whose use-case will become apparent in the withdrawal and challenge process described below;
- The maximum amount of gas that each user is allowed to consume in executing functions of C;
- The maximum amount of gas that each single function call to a function in C is allowed to consume;
- The maximum number of allowed function calls of C.

By enforcing one or more of the maximum values above, the developer must ensure that all gas usage in the contract's lifetime is covered. Using the values chosen by the developer, $\mathcal{W}$ provides the following additional functionality:

(1) *Joining.* Before being able to interact with C's functionality in $\mathcal{W}$, a user/party Paul $\in \mathcal{P}$ has to explicitly join $\mathcal{W}$ by calling $\mathcal{W}$.`join()` and providing a deposit of d ether. This deposit will remain in the contract's custody as long as Paul is a party to the contract and will be used to compensate for gas usage if Paul's dishonest behavior triggers an on-chain execution.

(2) *Virtual Banking.* Since $\mathcal{W}$ is lazy in running functions call of C and avoids running them on-chain by default, any money transfers by these calls is also not executed on-chain. To enable the functionality of money transfers between C and the participants, $\mathcal{W}$ acts as a bank and allows each joined user to deposit and withdraw ether to $\mathcal{W}$. The balances used in C's functions will then refer to the user's balance in $\mathcal{W}$, rather than her ether balance on the blockchain. These $\mathcal{W}$-balances are not explicitly stored on-chain. Each party in $\mathcal{P}$ keeps track of them off-chain on their own machine. The contract C has a $\mathcal{W}$-balance as well.

(3) *Ledger.* $\mathcal{W}$ keeps track of an on-chain internal ledger (as a state variable) which is a sequence of deposits and withdrawals by the users to $\mathcal{W}$'s bank and also the function calls requested by the users to functions of C.

(4) *Depositing Ether.* A party Paul can deposit ether to $\mathcal{W}$ at any time. The deposited ether will be under the control of $\mathcal{W}$ and an entry will be added to $\mathcal{W}$'s ledger certifying the amount that Paul deposited. This entry is added on-chain. Upon seeing this entry, all participants in $\mathcal{P}$ update their off-chain version of Paul's $\mathcal{W}$-balance accordingly.

(5) *Lazy Function Call.* $\mathcal{W}$ has a dedicated function which is named `requestCall` and can be utilized by a party Paul who wants to call a function in C. To call $C.f$, Paul has to create a transaction that calls $\mathcal{W}$.`requestCall` containing following parameters:
- The name f of the function that should be called,
- The maximum amount g_m of gas that may be used by f,
- The parameters that should be passed to f,
- the amount of money that should be paid from Paul's $\mathcal{W}$-balance to C's $\mathcal{W}$-balance. This is only applicable if $C.f$ is payable, i.e. if it can accept payments.

Upon receiving the items above, $\mathcal{W}$.`requestCall` first checks that Paul is not exceeding the maximum gas usage allowed by the developer. If this limit is exceeded, then the function call is ignored. Otherwise, instead of running $C.f$ or $\mathcal{W}.f$ on-chain, `requestCall` adds an entry to the internal ledger. This entry includes all the parameters (a-d) above, as well as a record of the values of all global variables such as **`block.number`** and **`msg.sender`** = Paul. This is all the information that can possibly be needed for executing the call to f, but the call itself is not executed on-chain. 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 and updates their own copy of the state variables in C and the $\mathcal{W}$-balances accordingly.

(6) *Withdrawing Ether.* The parties can decide to withdraw ether from their $\mathcal{W}$-balance at any time in two steps. Although the $\mathcal{W}$-balances are not explicitly stored in $\mathcal{W}$, all parties have executed function calls off-chain and are aware of all $\mathcal{W}$-balances.
- Step 1: The party Alice $\in \mathcal{P}$ calls $\mathcal{W}$.`requestWithdraw`(x) and specifies the amount x she wishes to withdraw as a parameter. This request is added to the $\mathcal{W}$-ledger.
- Step 2: If no other party challenges the withdrawal until t blocks after Step 1, then Alice can call the dedicated function $\mathcal{W}$.`withdraw` and receive the desired amount of money.

(7) *Challenging.* Suppose a party Alice $\in \mathcal{P}$ requests a withdrawal of x ether from her $\mathcal{W}$-balance. Although $\mathcal{W}$-balances are not explicitly stored in $\mathcal{W}$ on-chain, Bob $\in \mathcal{P}$ knows whether Alice has a $\mathcal{W}$-balance of at least x because all operations (lazy function calls, deposits, withdrawals) are all simulated by every other party off-chain on his own machine. If Bob finds out that Alice is trying to withdraw more than her $\mathcal{W}$-balance, then Bob can challenge the withdrawal at index j by calling a function named $\mathcal{W}$.`challenge`(j). When a challenge occurs, then either Alice is dishonest and trying to withdraw more than her balance or Bob is dishonest and stopping Alice from withdrawing her money. In such a case, the wrapper contract $\mathcal{W}$ initiates an on-chain evaluation.

(8) *Leaving.* Any party can call $\mathcal{W}$.`leave()` and leave $\mathcal{W}$ at any time to get their deposit d back as long as they do not have an active withdrawal request. A withdraw request is active if (i) the current time is within t blocks after the request, or (ii) the request has been challenged and the ensuing on-chain evaluation has not concluded yet.

(9) *On-chain Evaluation.* On-chain evaluation is triggered only when there is a disagreement about the state of C and a withdrawal is challenged. Let i be the index of the last operation in the $\mathcal{W}$-ledger that is executed on-chain. Originally, we have $i = 0$, but i might have increased in case of previous challenges. Let j be the index of the withdrawal request by Alice in the $\mathcal{W}$-ledger and suppose the request is challenged by Bob. We have to simulate all the operations in the range $[i + 1, j]$ of the $\mathcal{W}$-ledger *on-chain* in order to find the $\mathcal{W}$-balance of Alice at the time of the request. The contract $\mathcal{W}$ maintains a state variable b[Paul] for every party Paul $\in \mathcal{P}$. This is the $\mathcal{W}$-balance of Paul immediately after the on-chain execution of the i-th entry in the $\mathcal{W}$-ledger, i.e. his $\mathcal{W}$-balance up until the last operation that is executed on-chain. It also has a state variable $b[C]$ which similarly tracks the $\mathcal{W}$-balance of C. If we execute everything until index $j - 1$ on-chain, then b[Alice] would be the $\mathcal{W}$-balance of Alice at the time she requested to withdraw x units. So, we simply have to figure out whether b[Alice] $\geq x$. Note that a smart contract cannot initiate its own execution and all function calls have to be initiated by a user. Moreover, the initiator of a function call has to pay for its gas usage. However, we would like to ensure that (i) the dishonest party ultimately bears the gas costs, and (ii) the incurred gas costs are as small as possible. Hence, the wrapper $\mathcal{W}$ holds an auction for the role of initiator in which anyone can enter a bid for the gas price π that they are willing to charge for the execution. More specifically, the on-chain simulation contains the following steps:
- *Bidding.* For t blocks after the challenge, any party Ingrid $\in \mathcal{P}$ can call the function $\mathcal{W}$.`bid`(j, π) signifying that she is willing to initiate the on-chain execution as long as she is paid π units of currency (ether) per unit of gas. Note that this is not necessarily the amount that she really pays the miners per unit

of gas. Indeed, π can be larger than the real unit gas cost, hence giving some profit to Ingrid who is volunteering to initiate the on-chain execution. However, this profit is unlikely to be high as parties can undercut each other. The potentially dishonest parties Alice and Bob cannot bid. After t blocks, the smallest bidder becomes Ingrid and triggers on-chain simulation.

- *Simulating Every Entry of the $\mathcal{W}$-ledger.* After Ingrid is chosen in the bidding process above, she has to call $\mathcal{W}$.simulate(k) for every $i+1 \leq k \leq j$ in order. This function simulates the execution of the k-th entry in the $\mathcal{W}$-ledger *on-chain*. This will of course incur gas costs which are paid by Ingrid. However, $\mathcal{W}$.simulate stores of how many units of gas Ingrid has paid in gb[Ingrid]. If Ingrid fails to call $\mathcal{W}$.simulate(k) within t blocks after she called $\mathcal{W}$.simulate($k-1$), she loses her deposit d. Moreover, another bidding process will begin as above to find a new initiator for the rest of the on-chain simulations. This ensures that the simulation will be successfully carried out until index j.

4 EXPERIMENTAL RESULTS

Implementation. We implemented our approach, i.e. an automated tool to obtain the wrapped contract $\mathcal{W}$ from any given contract C in Python 3. We used Slither [16], Web3Py [22], and Hardhat [11] for parsing smart contracts written in Solidity and simulating the gas usage on our machine.

Overall Savings in Gas Usage. As benchmarks, we took all the Ethereum smart contracts that were deployed in Etherscan between June and October 2022. Some of them were removed either because they were extremely simple contracts, e.g. basic ERC 20 with no additional functionality, or because they were not parsable using our libraries. Thus, we report results over the remaining **3,330** real-world contracts with **327,132** function calls. The total gas usage was reduced from 51,845,786,705 gas units ≈ 727.09 ETH ≈ **1,261,801 USD** to 31,058,348,542 gas units ≈ 469.86 ETH ≈ **819,149** USD. So, our approach provided a reduction of **40.09 percent**, corresponding to a whopping **442,651 USD** in a short period of only 121 days. Each contract's gas usage was, on average, reduced by **51.65 percent**. This shows the significant real-world utility of our method, as well as the fact that current Ethereum smart contracts are quite wasteful in terms of gas usage. The average improvement per transaction was 63,544.49 units of gas ≈ 0.0008 ETH ≈ 1.35 USD. Figure 1 provides more detailed results.

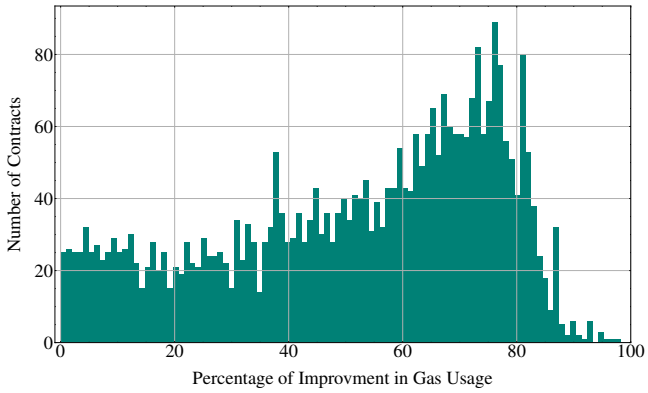

Figure 1: Improvements obtained by our approach in the gas usage of each benchmark contract.

ACKNOWLEDGMENTS

The research was partially supported by the Hong Kong Research Grants Council ECS Project Number 26208122, the HKUST-Kaisa Joint Research Institute Project Grant HKJRI3A-055 and the HKUST Startup Grant R9272. Authors are ordered alphabetically.

REFERENCES

[1] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *TACAS*. 118–125.
[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on Ethereum smart contracts. In *POST*. 164–186.
[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST*. 164–186.
[4] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2019. The treewidth of smart contracts. In *SAC*. 400–408.
[5] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Yaron Velner. 2018. Ergodic Mean-Payoff Games for the Analysis of Attacks in Crypto-Currencies. In *CONCUR*. 11:1–11:17.
[6] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. 2019. Hybrid mining: exploiting blockchain's computational power for distributed problem solving. In *SAC*. 374–381.
[7] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. 2019. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. In *IEEE ICBC*. 403–412.
[8] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *ESOP*. 739–767.
[9] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *SANER*. 442–446.
[10] Micah Dameron. 2018. Beigepaper: an Ethereum technical specification. *Ethereum Project Beige Paper* (2018).
[11] Ethereum development environment for professionals. 2022. Ethereum development environment for professionals. https://hardhat.org/
[12] Ethereum Foundation. 2022. Gas and Fees. https://ethereum.org/en/developers/docs/gas/
[13] Ethereum Foundation. 2022. Smart contract languages. https://ethereum.org/en/developers/docs/smart-contracts/languages/
[14] Ethereum Foundation. 2022. Solidity Language Documentation. https://docs.soliditylang.org
[15] Ethereum Foundation. 2022. Vyper: a contract-oriented pythonic programming language for the EVM. https://vyper.readthedocs.io/en/stable/
[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *WETSEB@ICSE*. IEEE / ACM, 8–15.
[17] Amir Kafshdar Goharshady. 2021. Irrationality, Extortion, or Trusted Third-parties: Why it is Impossible to Buy and Sell Physical Goods Securely on the Blockchain. In *Blockchain*. 73–81.
[18] Amir Kafshdar Goharshady, Ali Behrouz, and Krishnendu Chatterjee. 2018. Secure Credit Reporting on the Blockchain. In *iThings/GreenCom/CPSCom/SmartData*. 1343–1348.
[19] Paul Hudak. 1989. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.* 21, 3 (1989), 359–411.
[20] Jan-Willem Maessen. 2002. *Hybrid eager and lazy evaluation for efficient compilation of Haskell*. Ph.D. Dissertation. MIT.
[21] Mohsen Alambardar Meybodi, Amir Kafshdar Goharshady, Mohammad Reza Hooshmandasl, and Ali Shakiba. 2022. Optimal Mining: Maximizing Bitcoin Miners' Revenues from Transaction Fees. In *Blockchain*. 266–273.
[22] Python library for interacting with Ethereum. 2022. Python library for interacting with Ethereum. https://web3py.readthedocs.io/
[23] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014), 1–32.