Improved Gas Optimization of Smart Contracts

Togzhan Barakbayeva[1], Soroush Farokhnia[1], Amir Kafshdar Goharshady[2], Pingjiang Li[1], Zhaorun Lin[1]

[1]Hong Kong University of Science and Technology
Clear Water Bay, New Territories, Hong Kong
[2]University of Oxford, Oxford, United Kingdom

Abstract. Smart contracts are programs executed on top of a blockchain consensus protocol. Their compiled code (bytecode) is stored on the blockchain and is immutable after deployment. They are self-enforcing in the sense that any function call to a smart contract is executed by all nodes on the network, ensuring that they all reach consensus about the final state of the contract. To prevent denial-of-service attacks, such an execution is costly by design. A "gas" cost is assigned to each bytecode operation, roughly proportional to the resources required to execute it, and any user who initiates a function call to a smart contract has to pay the total gas cost of the resulting execution. On Ethereum alone, the users pay an astounding gas cost of more than 4 billion USD/year.

Smart contracts are often written in high-level programming languages such as Solidity and then compiled to bytecode before being deployed on the blockchain. Thus, a natural compiler optimization problem arising in this context is to produce efficient bytecode that minimizes the total gas usage. A leading approach in this direction is superoptimization, which considers every basic block of the smart contract separately and tries to rewrite it as an equivalent block that uses as little gas as possible. The current state-of-the-art tool is `syrup 2.0`, which encodes gas superoptimization as `Max-SMT` and then relies on SMT-solvers to synthesize an equivalent contract with optimized gas usage.

In this work, we make two observations: First, the performance of `Max-SMT` declines significantly as block sizes increase. Thus, although `syrup` is able to find an optimal rewriting for a small block with a dozen bytecode operations, its output on blocks with hundreds or thousands of operations, when given any realistic timeout, is far from optimal. Second, optimizations that can be applied to basic blocks are often local and compositional, i.e. they rewrite several small and disjoint parts of the block. Such locality is lost to `Max-SMT` solvers, mainly because it is unpredictable and there are no clear ways on how one should cut blocks of bytecode into smaller sub-blocks. To ameliorate these issues, we present a simple dynamic programming algorithm that tries every possible division of a block into sub-blocks, recursively calling `syrup` as a black box on each sub-block. Surprisingly, this simple idea leads to highly significant improvements in the gas usage, more than doubling the savings obtained by `syrup`, and reducing the gas usage of real-world smart contracts by 11.23 percent.

Keywords: Smart Contracts, Compiler Optimization, Blockchain

1 Introductory Blockchain Concepts

Blockchain. Blockchain is a family of distributed consensus protocols, first designed by Satoshi Nakomoto as the underlying protocol of Bitcoin [14]. In such protocols, our goal is to reach a consensus about an ordered sequence of *transactions*. In Bitcoin, a transaction encodes transfers of money. When a user creates a new transaction, they broadcast it to the whole network using a gossip protocol. Every node on the network keeps track of the transactions they have heard of (called the *mempool*) but does not consider them finalized until they are added to the *blockchain*. A blockchain, as its name suggests, is a chain (singly-linked list) of blocks, with each block B_i containing a sequence of transactions $\langle Tx_{i,0}, Tx_{i,1}, \ldots \rangle$ and a pointer to the previous block B_{i-1}. See Figure 1. Every node keeps track of a copy of the blockchain. To ensure consensus, appending new blocks to the end of the chain is a costly endeavor, called *mining*. Suppose the blockchain contains n blocks. A *miner* is a node that gathers unfinalized transactions, bundles them in a block B_{n+1} and attempts to append this block to the end of the consensus blockchain. The block B_{n+1} should also contain a proof π_{n+1} certifying that the miner is permitted by the protocol to add this block. In Bitcoin, one needs to solve a hard proof-of-work puzzle which is based on inverting a hash function. When the puzzle is solved successfully, the miner broadcasts their block B_{n+1}. The solution to the hash inversion puzzle serves as π_{n+1}. Every node then verifies the solution and adds the block to their copy of the blockchain. See [12] for a more detailed treatment. Proof-of-work is not the only consensus mechanism. There are many other well-established mechanisms [6, 4, 10], such as proof-of-stake [5] in which a miner's chance of being permitted to add the next block is proportional to their holdings in the currency.

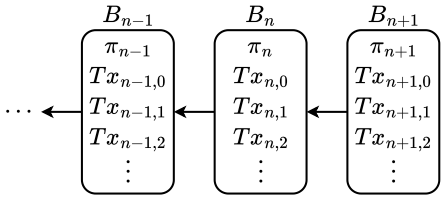

Fig. 1. A simplified view of a blockchain when the block B_{n+1} is appended.

Programmable Blockchains and Smart Contracts. While Bitcoin was the first cryptocurrency based on a blockchain protocol, Ethereum [18] pioneered the concept of smart contracts. A smart contract is a program that is stored on the blockchain. Every node on the network keeps track of the state of every contract. This is achieved by extending what a transaction can do. In programmable

blockchains, a transaction can (i) transfer money, (ii) deploy a new smart contract, providing its code – which will be saved on the blockchain as part of the transaction, or (iii) call a function in a previously-deployed smart contract, providing the arguments necessary for the function call.

Consensus. The blockchain protocol provides consensus on the history and order of transactions. Thus, every node on the network has the same view of the smart contracts, i.e. sees the same codes deployed by transactions on the blockchain and sees the same function calls to each contract in the same order. Therefore, each node can execute the transactions in the order they appear on the blockchain and reach consensus about the state of every contract, e.g. values of the variables internal to the contracts. This, together with the fact that smart contracts can receive and hold money in the form of the base cryptocurrency, allows one to implement real-world financial contracts as smart contracts. Of course, the underlying programming language should be unambiguous and deterministic, ensuring that different nodes executing the same sequence of function calls over the same contracts reach the same results. To achieve this, Ethereum designed a virtual machine (EVM) that supports a completely-specified low-level assembly-like bytecode format for writing smart contracts [18]. The EVM bytecode language is Turing-complete [18]. In practice, developers write their smart contracts in high-level languages, such as Solidity [7], and then a compiler, such as `solc`, compiles it to EVM bytecode.

Gas. Since our language is Turing-complete, there is nothing to stop programmers from writing long-executing or even non-terminating contracts or contracts that use a lot of storage. As the simplest example, one can write an infinite loop `while(true) {...}` in a smart contract, deploy it on the blockchain, and then create a transaction that invokes it. In such a scenario, when this invocation is added to the blockchain, every node on the network will have to execute it, causing a deadlock. To avoid situations like this, Ethereum introduced the concept of *gas*. Put simply, a gas cost is associated to every bytecode operation code (opcode). The gas cost is meant to be proportional to the actual cost of executing the operation. The costs have fixed formulas and provided as a table in the Ethereum Yellowpaper [18]. When a user creates a transaction that calls a function, they have to pay for the total gas usage of its execution, i.e. the sum of gas costs of all invoked opcodes. More specifically, the user has to specify the maximum amount g of gas that may be used in their function call and the amount of money p (in Ether) they are willing to pay per unit of gas. The transaction will first take a deposit of $g \cdot p$ from the user and then start executing the desired function call. If the transaction runs out of gas, i.e. the invocation requires more than g units of gas, it will be reverted without refunding the deposit. Otherwise, if it uses $\overline{g} \leq g$ units of gas, the user pays $\overline{g} \cdot p$ to the miners as a transaction fee and the rest is refunded [18].

Related Works. Analyzing and reducing gas costs are central research problems in the blockchain community. Out-of-gas errors are the source of many vulnerabilities and thus there are several tools focused on finding upper-bounds on the

gas usage of smart contracts [15, 1, 3]. In 2023, on Ethereum alone, gas costs were more than 4 billion USD [8]. Given this high cost, and the fact that gas usage is defined for low-level bytecode operations whereas programmers write their contracts in high-level languages such as Solidity, it is crucial that the compiler optimizes for gas usage. Indeed, the standard Solidity compiler `solc` has a flag `-optimize` which enables heuristics for optimizing the gas usage of the resulting bytecode. The Solidity language documentation [7] also talks about *gas-hungry* patterns and instructs programmers to avoid them in their code. There are many works on layer-two protocols which aim to minimize the amount of on-chain computation, i.e. gas-consuming calls to smart contracts, by moving most of the protocol off-chain [17] or delaying and avoiding the execution as far as possible [9].

Superoptimization. The current state-of-the-art in gas optimization by compilers is the work [2] which provides a tool called `syrup 2.0` that supports both Solidity source code and EVM bytecode as its input and outputs gas-optimized bytecode. Their approach is based on the concept of superoptimization [13]. Basically, the idea is to break the bytecode program down into its basic blocks, i.e. maximal straight-line subprograms that do not contain branching or jumps. Then, each basic block B is optimized separately by exhaustively trying all possible rewritings B' that are equivalent to B and taking the one with the smallest gas usage. Superoptimization is a well-known technique that has been implemented in mainstream tools and compilers such as LLVM [11, 16] usually with the goal of reducing runtime or memory usage. However, exhaustive search is far from scalable and can only be applied to toy programs with tiny basic blocks. For example, a `C++` basic block containing the single operation `x*=2` can easily be rewritten as `x«1`, reducing its execution time, but as the size of the block grows there will be a combinatorial explosion in the number of possible rewritings. Instead, [2] encodes the problem as `Max-SMT` and passes it to modern SMT-solvers. This encoding, and the rewrite rules for obtaining equivalent basic blocks, are far from trivial. Indeed, [2] provides several different encodings and experimentally finds the best combinations. The reduction to `Max-SMT` is the key to `syrup`'s scalability and enables huge savings in real-world gas costs of Ethereum smart contracts.

Example. Consider a simple basic block in Solidity that performs the operation $y = x \hat{} x$. Here, x and y are integers and $\hat{}$ is the bitwise exclusive or operation. Our goal is to compile this basic block to EVM bytecode. A naive compiler that applies no optimization, such as `solc` with all optimizations turned off, would provide the bytecode in Figure 2 (left). In EVM bytecode, `PUSH` adds a new item to the top of the stack, `POP` removes the top item, `SLOAD` loads a word from storage, `DUP` duplicates an item on the stack and `SWAP` swaps items in the stack. As expected, `XOR` performs the bitwise exclusive or operation. Each of these operations has a gas cost, which is fixed by the Ethereum Yellowpaper [18, Appendix H]. In this case, the naive compilation will lead to a total gas cost of 1,420 for this basic block. In contrast, a compiler that realizes $x \hat{} x = 0$ and thus rewrites $y = x \hat{} x$ as $y = 0$ would not even need to perform the `XOR` operation.

This leads to a much more gas-efficient bytecode such as the one in Figure 2 (center). This is the output of `syrup` and uses only 720 units of gas, almost halving the execution cost. Finally, it is possible to further optimize even this bytecode, obtaining the basic block in Figure 2 (right), which uses 706 units of gas. Note that all three basic blocks of Figure 2 are semantically equivalent and differ only in their gas usage.

```
PUSH1 0          PUSH1 0
SLOAD            SLOAD
PUSH1 0          PUSH1 1
SLOAD            PUSH1 0          PUSH1 0
XOR              SWAP2            SLOAD
PUSH1 1          POP              PUSH1 1
DUP2             SWAP1
SWAP1            SWAP1
```

Fig. 2. Three compilations of $y = x \hat{} x$ to EVM bytecode: naive (left), optimized (center), and further optimized (right).

Our Contribution. In this work, we use `syrup` as a black box and design a simple and elegant dynamic programming algorithm for optimizing the gas usage of Ethereum smart contracts. Our algorithm is also a flavor of superoptimization, i.e. it optimizes each basic block separately. We report significant improvements in the gas usage of the resulting smart contracts, not only in comparison with the unoptimized version, but also against `syrup` itself.

2 Our Algorithm

In this section, we present our simple dynamic programming algorithm. Our approach is based on the two intuitive observations below.

Observation 1: Scalability. We observe that `syrup` works really well on small basic blocks and often finds the optimal rewriting. However, this is no longer the case when the basic block increases in size. For basic blocks with more than a hundred bytecode operations, `syrup` rarely finds the optimal rewriting and often produces extremely suboptimal results, even when given generous time limits of several hours. This is not surprising since the problem is reduced to `Max-SMT` and SMT-solvers are simply not scalable enough to handle large basic blocks.

Observation 2: Locality and Compositionality. Our second observation is that gas-optimizing changes to basic blocks are often local and compositional. For example, a block with thousands of operations will probably be optimizable by hundreds of different local rewritings which are quite independent of each other.

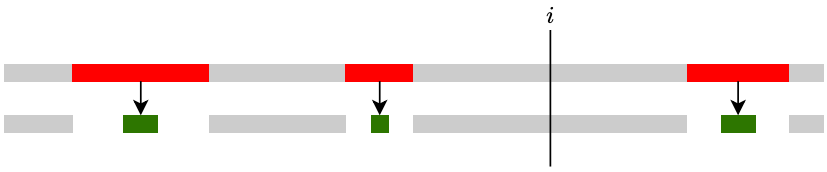

Fig. 3. A basic block B (top) in which some portions (red) can be optimized to use less gas (green).

More formally, let $\mathsf{B} = \langle \mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_n \rangle$ be a basic block consisting of n EVM operations, $g(\mathsf{B}) = \sum_{i=1}^{n} g(\mathsf{op}_i)$ be its gas usage, and $B^* = \mathsf{Optimized}(\mathsf{B})$ be the optimal rewriting of B, i.e. the equivalent basic block that uses minimal gas. Additionally, let $\mathsf{B}[i \ldots j]$ be the sub-block of B from op_i to op_j. We conjecture that in almost all cases, there is an index i such that

$$g(\mathsf{Optimized}(\mathsf{B})) = g(\mathsf{Optimized}(\mathsf{B}[1 \ldots i])) + g(\mathsf{Optimized}(\mathsf{B}[i+1 \ldots n])).$$

In other words, B can be divided in two parts and each part can be (recursively) optimized separately. This is shown in Figure 3. This intuition leads to two challenges: (i) how to identify when this kind of compositionality is present, and (ii) how to find the correct index i for dividing B in two parts. Our algorithm sidesteps both of these difficulties by simply brute-forcing all possibilities.

Our Algorithm. We use `syrup` as a black box in our algorithm. Let $\mathsf{B} = \langle \mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_n \rangle$ be a basic block and $\mathsf{syr}(\mathsf{B})$ be the gas-optimized block obtained by applying `syrup` to B. Instead of applying `syrup` directly to B, we can first divide B in two parts $\mathsf{B}[1 \ldots i]$ and $\mathsf{B}[i+1 \ldots n]$ and then optimize each part separately. We simply try this for all possible i, considering further sub-divisions recursively. Formally, let $\mathsf{leastGas}(\mathsf{B})$ be the minimum amount of gas usage that we can obtain by rewriting B to an equivalent basic block. We have:

$$\mathsf{leastGas}(\mathsf{B}) = \min \left\{ g(\mathsf{syr}(\mathsf{B})), \min_{i=1}^{n-1} \mathsf{leastGas}(\mathsf{B}[1 \ldots i]) + \mathsf{leastGas}(\mathsf{B}[i+1 \ldots n]) \right\}.$$

This formula leads itself to dynamic programming and tracing the dynamic programming steps can also help us find an equivalent block B^* with $g(\mathsf{B}^*) = \mathsf{leastGas}(B)$. More specifically, for every sub-block $\mathsf{B}[i \ldots j]$, we can find the best optimization. This is shown in Algorithm 1. Note that our algorithm does not guarantee that the resulting block will be globally optimal, but only that it will use no more gas than `syrup`'s output. As we will see in Section 3, the improvement is quite substantial in practice. Finally, we note that our algorithm can easily be parallelized at lines 5 and 9.

Algorithm 1 Our Algorithm: Dynamic Programming using `syrup` as a Blackbox

Input: A basic block $\mathsf{B} = \langle \mathsf{op}_1, \ldots, \mathsf{op}_n \rangle$ of n EVM bytecode operations
Output: A gas-optimized block B^* which is equivalent to B
int $\mathsf{leastGas}[n][n]$ ▷ $\mathsf{leastGas}[i][j]$ holds $\mathsf{leastGas}(\mathsf{B}[i \ldots j])$
block $\mathsf{bestBlock}[n][n]$ ▷ $\mathsf{bestBlock}[i][j]$ holds the best rewriting for $\mathsf{B}[i \ldots j]$
for all $i \leq j$ **do**
    $\mathsf{bestBlock}[i][j] \leftarrow \mathsf{syr}(\mathsf{B}[i \ldots j])$ ▷ Start with `syrup`'s output as the base case
    $\mathsf{leastGas}[i][j] \leftarrow g(\mathsf{bestBlock}[i][j])$
for $1 \leq l \leq n$ **do** ▷ l is the length of our sub-block
    for $1 \leq a \leq n - l + 1$ **do** ▷ a is the starting index of our sub-block
        $b \leftarrow l + a - 1$ ▷ b is the end index of our sub-block
        for $a \leq i \leq b$ **do** ▷ Try breaking the sub-block $\mathsf{B}[a \ldots b]$ at index i
            if $\mathsf{leastGas}[a][i] + \mathsf{leastGas}[i+1][b] < \mathsf{leastGas}[a][b]$ **then**
                $\mathsf{leastGas}[a][b] = \mathsf{leastGas}[a][i] + \mathsf{leastGas}[i+1][b]$
                $\mathsf{bestBlock}[a][b] = \mathsf{bestBlock}[a][i] \cdot \mathsf{bestBlock}[i+1][b]$
                ▷ The operator $\cdot$ represents concatenation of basic blocks.
return $B^* = \mathsf{bestBlock}[1][n]$

3 Experimental Results

In this section, we provide an experimental comparison between our algorithm and `syrup` over real-world Ethereum smart contracts. We implemented our algorithm in Python and integrated it with `syrup 2.0` [2]. All results were obtained on an Intel Xeon Gold 5317 machine (3.0 GHz, 12 cores, 18M cache) with 128 GB of RAM running Ubuntu 20.04.6 LTS. We performed gas optimization experiments on the benchmark smart contracts from [2], which contain some of the most widely-used real-world contracts on the Ethereum blockchain.

Benchmarks. We considered the 148 smart contracts of the benchmark suite of [2]. According to [2], this suite contains a random sampling of the most commonly-called smart contracts on the Ethereum blockchain. 128 of these contracts are provided in the EVM bytecode format and match the exact versions deployed on the blockchain. These deployed bytecodes were most likely obtained by the developers compiling high-level Solidity code using a version of `solc` that was available at the time of their deployment. The other 20 are provided as Solidity source code. We compiled them to bytecode using `solc`. Overall, this benchmark set consists of 22,136 basic blocks.

Gas Optimization Results. The total gas usage of unoptimized contracts over all basic blocks was 1,972,141 units of gas. This was reduced to 1,889,918 units of gas when applying `syrup 2.0`. It was further reduced to 1,750,576 units by our dynamic programming algorithm. Thus, `syrup` provides an overall gas saving of 4.17 percent over the baseline, whereas our approach obtains a saving of 11.23 percent (7.37 percent over `syrup`). More specifically, `syrup` succeeded in improving the gas usage in 4,678 basic blocks, with an average improvement of 21.40 percent over these 4,678 blocks. Our approach improved the gas usage

in 5,877 basic blocks, with an average improvement of 30.52 percent over these blocks. In comparison to `syrup`, our approach improved the gas usage in 4,328 basic blocks with the average improvement being 24.82 percent. Figures 4–5 visualize the data as histograms.

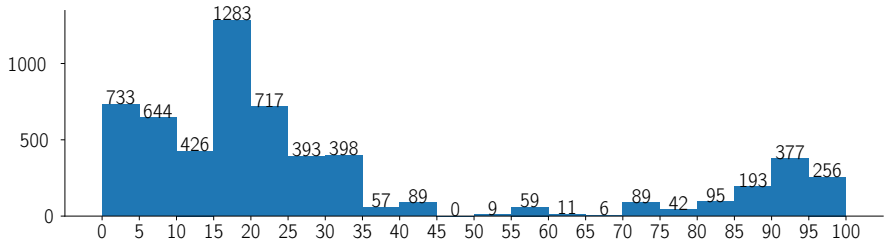

Fig. 4. Histogram of gas improvements obtained by our approach over the unoptimized smart contracts. The x axis is the percentage of improvement (bin size = 5%) and the y axis is number of basic blocks.

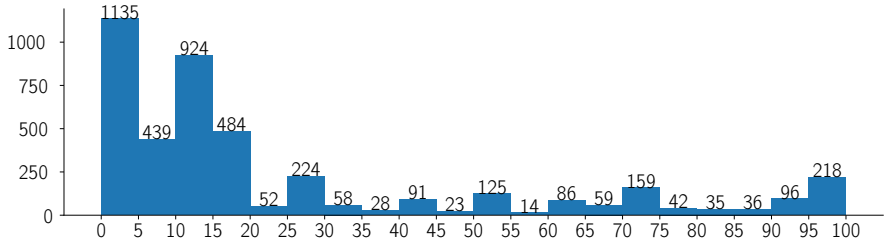

Fig. 5. Histogram of gas improvements obtained by our approach over `syrup 2.0`. The x axis is the percentage of improvement (bin size = 5%) and the y axis is number of basic blocks.

Summary. In summary, our simple dynamic programming more than doubles the benefits of `syrup`. We obtained a 11.23 percent improvement in gas usage over these standard benchmarks. We note that such an improvement is highly significant since the total annual gas cost on Ethereum is more than 4 billion USD.

4 Conclusion

Smart contract gas costs are a significant expense for Ethereum users, amounting to more than 4 billion USD last year. In this work, we provided a simple dynamic

programming approach to enhance superoptimization techniques for reducing the gas usage of smart contracts. We implemented our approach and integrated it with `syrup 2.0`, the current state-of-the-art gas optimizer for Ethereum smart contracts. Over a benchmark set consisting of 148 real-world and commonly-called smart contracts on the Ethereum blockchain, we observed that, amazingly, our approach more than doubles the benefits of `syrup`, increasing the gas savings from 4.17 percent to 11.23 percent.

Acknowledgments

The research was partially supported by an Ethereum Foundation Research Grant, as well as the Hong Kong Research Grants Council ECS Project Number 26208122. Togzhan Barakbayeva was supported by the Hong Kong PhD Fellowship Scheme (HKPFS).

References

1. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: *Don't run on fumes* - parametric gas bounds for smart contracts. J. Syst. Softw. **176**, 110923 (2021)
2. Albert, E., Gordillo, P., Hernández-Cerezo, A., Rubio, A., Schett, M.A.: Superoptimization of smart contracts. ACM Trans. Softw. Eng. Methodol. **31**(4), 70:1–70:29 (2022)
3. Cai, Z., Farokhnia, S., Goharshady, A.K., Hitarth, S.: Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts. Proc. ACM Program. Lang. **7**(OOPSLA2), 882–911 (2023)
4. Chatterjee, K., Goharshady, A.K., Pourdamghani, A.: Hybrid mining: exploiting blockchain's computational power for distributed problem solving. In: SAC. pp. 374–381 (2019)
5. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: EUROCRYPT (2018)
6. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: CRYPTO (2015)
7. Ethereum Foundation: Solidity: An object-oriented high-level language for implementing smart contracts (2024), https://docs.soliditylang.org/en/v0.8.28/
8. Etherscan: Ethereum gas tracker (2024), https://etherscan.io/gastracker
9. Farokhnia, S., Goharshady, A.K.: Reducing the gas usage of ethereum smart contracts without a sidechain. In: ICBC. pp. 1–3 (2023)
10. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: SOSP. pp. 51–68 (2017)
11. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. pp. 75–88 (2004)
12. Laurence, T.: Introduction to Blockchain technology. Van Haren (2019)
13. Massalin, H.: Superoptimizer - A look at the smallest program. In: ASPLOS. pp. 122–126 (1987)
14. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Satoshi Nakamoto (2008)

15. Nassirzadeh, B., Sun, H., Banescu, S., Ganesh, V.: Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. In: MARBLE (2022)
16. Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Taneja, J., Regehr, J.: Souper: A synthesizing superoptimizer. CoRR **abs/1711.04422** (2017)
17. Thibault, L.T., Sarry, T., Hafid, A.S.: Blockchain scaling using rollups: A comprehensive survey. IEEE Access **10**, 93039–93054 (2022)
18. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)