UNIVERSITÀ DI PISA

Ethereum dApp

TRY: a nfT lotteRY

Matteo Pinna

Introduction

The decentralized application (dApp) that has been developed implements the logic of a NFT lottery, in which users are able to win collectibles as prizes. The project has been developed with *Solidity* on the *Ethereum* blockchain, leveraging the *Remix IDE* as development, deployment and testing environment.

Implementation

The implementation consists of two contracts: *Lottery.sol* and *CryptoPepes.sol*, respectively the contract dedicated to the lottery and the one dedicated to the NFTs. The NFT contract is implemented according to the ERC721 standard, that is the de-facto standard for non-fungible tokens.
It's important to point out that, since *pragma solidity >= 0.8.0* is being used, there is no need to manually import *SafeMath* library for safe arithmetic operations since it is implemented by default by the compiler.

In the upcoming subsections the focus will be on some implementation choices that have been made, the contracts will be discussed in depth in the next dedicated section.

Code Style

The project is compliant with the official *Solidity Style Guide*[1] (i.e. naming, variable and function ordering, comments, etc.), to obtain a clean and elegant code base while complying with the main conventions of Solidity.

Gas optimization patterns

In order to make the dApp more efficient in terms of gas consumption, some of the most common Solidity patterns for gas optimization[2] have been used.
Here is a brief description of each one:

- **Packing variables:** since Solidity contracts have contiguous 32 byte (256 bit) slots used for storage, by ordering our variables by type it is possible to reduce the number of storage slots used in the contracts.
- **Memory vs Storage:** since storage is extremely expensive, temporary memory variables inside functions have been used whenever possible.
- **Internal calls:** leveraging Solidity inheritance and internal parameters/functions is more efficient w.r.t. public/external calls to other contracts.

[1] https://docs.soliditylang.org/en/v0.8.14/style-guide.html
[2] https://docs.soliditylang.org/en/v0.8.14/

- **Solidity compiler optimization**
- **Unchecked block:** the unchecked block allows to perform operations while skipping overflow/underflow checks, this is extremely useful when incrementing the index *i* inside a for loop, since such parameter is already bounded by the loop condition (for instance, $i < length$). Hence, there is no risk of overflow/underflow vulnerabilities and it is possible to save some gas.

Randomness source

As source of (pseudo)-randomness, data from the blockchain has been used (i.e. *block.difficulty* and *block.number*). To ensure having different pseudo-random numbers in case of consecutive calls also a seed has been added as input parameter to the computation of the *KECCAK256*.

Contracts

In Figure 1 a UML class diagram representing the structure of the dApp. Specifically, the inheritance between the contracts and which methods they use. Notice that trivial get/set methods, and ERC721 methods that are not used by *CryptoPepes.sol* have been omitted for simplicity and to ease the visualization.

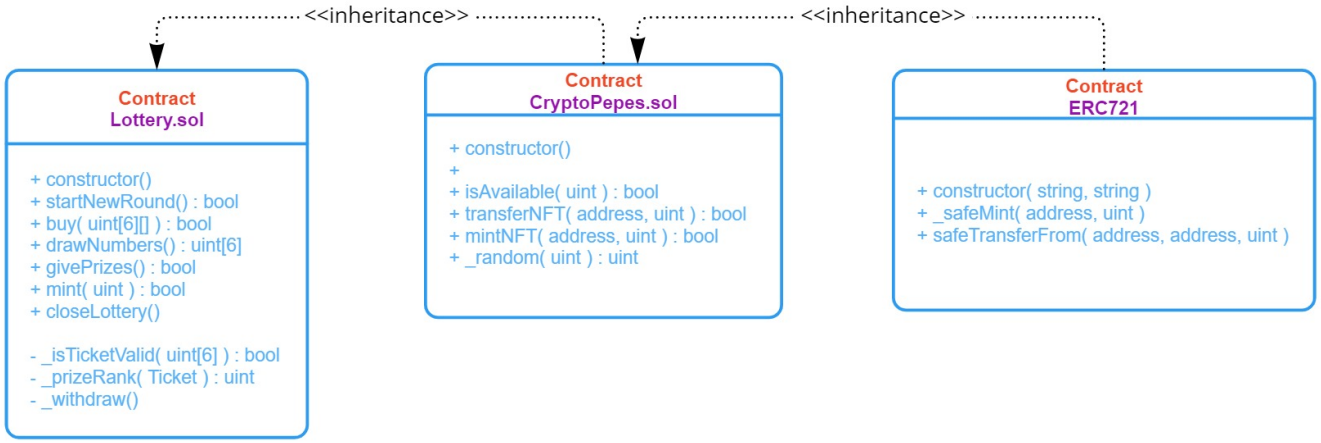

Figure 1: UML class diagram representing the dApp's contracts and their core methods.

Notice that it has been decided to make the *Lottery.sol* inherit from *CryptoPepes.sol* rather than creating an instance inside the lottery, since in this way it is possible to leverage Solidity inheritance and internal parameters and function of the NFT contract. This approach is extremely efficient w.r.t. implementing *CryptoPepes.sol* methods as public or external and making calls from inside the lottery, and is also compliant w.r.t. the Solidity gas optimization pattern previously mentioned.

Lottery.sol

Lottery.sol implements the lottery logic, most of its methods are callable only by the lottery operator (contract owner). We're going to give a short description of each method of the contract, categorized for visibility (i.e. public, private), as well as the requirements for their execution.

Below a brief description of each public method:

- *getBalance():* retrieve the contract balance.
- *getTicket(uint index):* retrieve a specific ticket of the current round.
- *getWinningNumbers():* retrieve the winning numbers of the current round.
- *setTicketPrice():* update the ticket price.

  - **Requirements**
    - operator only
    - ended round only

- *setDuration():* update the duration of a round.
  - **Requirements**
    - operator only
    - ended round only

- *setWithdrawal():* update the address for withdrawals.
  - **Requirements**
    - operator only

- *startNewRound():* start a new round by initializing some parameters or resetting them if it's not the first round (e.g. player list, winning numbers, etc.).
  - **Requirements**
    - operator only
    - endend round only

- *buy(uint[6][] _numbers):* buy a set of tickets, if duplicate standard numbers the ticket is invalid and not added to the lottery nor refunded. Also, if someone sends a *msg.value* which is greater than the total tickets price, the change is refunded.
  - **Requirements**
    - active round only
    - *msg.value* enough for tickets purchase
    - numbers in valid range

- *drawNumbers():* draw the pseudo-random winning numbers of current round and check that there are no duplicate standard numbers.
  - **Requirements**
    - operator only
    - inactive round only
    - winning numbers not already drawn

- *givePrizes():* assign a pseudo-random prize of a specific rank to each winner, the rank is computed according to the match between player ticket and winning numbers. If there is no NFT of such rank available, a new one is minted.
  - **Requirements**
    - operator only
    - inactive round only
    - winning numbers drawn

- *mint(uint rank):* mint a new NFT with a specific rank, used by the lottery operator to add new prizes at any moment.
  - **Requirements**
    - operator only

- *closeLottery():* deactivate the contract, refund all tickets if current round has not ended.
  - **Requirements**
    - operator only

Notice that the deactivation of the contract in *closeLottery()* has been implemented using the Solidity *selfdestruct()*, which is the standard approach for deactivating a contract. The *selfdestruct()* automatically withdraws all the contract balance to a specified address and resets to 0 all contract's parameters.

Below a brief description of each private method:

- *_isTicketValid(uint[6] _numbers):* check validity of a ticket (i.e. no duplicates in standard numbers).
- *_prizeRank(Ticket ticket):* get the rank of the prize to be assigned by computing the match between player ticket and winning numbers.
- *_withdraw():* withdraw contract balance to an address specified by the operator, called each time at the end of *givePrizes()*.

CryptoPepes.sol

For implementing the NFT contract, the standard OpenZeppelin ERC721 contract[3] has been used.
We're going to give a short description of each ERC721 method used inside our contracts, and of each additional method that that has been implemented in order to comply with the project requirements.

Below a brief description of each *internal* ERC721 method used inside *CryptoPepes.sol*:

- *_safeMint(address to, uint tokenId):* safely mints a new collectible with id *tokenId* and transfers it to *to*.
- *safeTransferFrom(address from, address to, uint tokenId):* safely transfers the collectible with id *tokenId* from *from* to *to*
- *ownerOf(uint tokenId):* retrieve owner address of token with id *tokenId*

Below a brief description of each public method of *CryptoPepes.sol*:

- *getAvailablePepes(uint rank):* retrieve the list of tokenIds of NFTS with a specific rank
- *descriptionOf(uint tokenId):* retrieve the description of a specific NFT
- *rankOf(uint tokenId):* retrieve the rank of a specific NFT
- *isAvailable(uint rank):* check whether there is at least one available NFT of a specific rank

Below a brief description of each *internal* method of *CryptoPepes.sol*:

- *mintNFT(address to, uint rank):* mint and transfer an NFT with a specific rank to a player that won the lottery, used when there are not enough prizes.
- *transferNFT(address to, uint rank):* to transfer a randomly picked NFT with a specific rank to a player that won the lottery, used when there are enough prizes.
- *_random(uint seed):* generates a pseudo-random number using blockchain data as source.

Below a brief description of each private method of *CryptoPepes.sol*:

- *_randomDescription(uint seed):* generates a pseudo-random string which will be used as NFT metadata.

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol

Events

In order to notify the outcome of the core functionalities of the dApp, a log is generated each time a major event occurs. This is achieved by leveraging Solidity *event-emit* pattern.
Here is a list of the events that are logged in the system:

- *LogDeployment(address owner):* log contract deployment by address *owner*.
- *LogNewRound(uint blockNUmber, uint roundNumber):* log start of a new round with block number *blockNumber* and current round id *roundNumber*.
- *LogTicketPurchase(address from, uint amount, uint[6][] numbers):* log ticket purchase from address *from* with msg.value *amount* and numbers chosen *numbers*.
- *LogTicketDuplicates(address from, uint[6] invalidNUmbers):* log rejection of an invalid ticket purchased from *from* with numbers *invalidNumbers*.
- *LogNumbersDrawing(uint[6] winningNumbers):* log drawing of the winning numbers *winningNumbers*
- *LogMint(address to, uint prizeRank, uint tokenId):* log mint of a new token when no prizes available, *to* is the address of the winner, *prizeRank* the rank of the new collectible being minted and *tokenId* its id.
- *LogWinner(address to, uint prizeRank, uint tokenId):* log the winning of a prize of rank *prizeRank*, with id *tokenId* and address of the winner *to*.
- *LogTicketRefund(address to, uint amount):* log the refund of a ticket price with amount *amount* and intended for player with address *to*. This event is logged both when a refund occurs because a player sent too much msg.value and when a refund occurs because the lottery is closed before the end of a round.
- *LogWithdrawal(address to, uint amount):* log the withdrawal of the contract balance at the end of a round, *to* is the address to which the contract balance is sent and *amount* is the value being transferred.
- *LogDeactivation(address to):* log the deactivation of a contract, *to* is the address to which the contract balance is sent before deactivation.

Gas estimate

In order to gain insights about the complexity of the system, the gas cost of some of the smart contract's main methods has been estimated.
The estimate is based on the official operation costs found in the *Ethereum Yellow Paper [pag. 27-29]*[4]. In *Appendix A* it is possible to view an extract of the mentioned paper which should be sufficient to understand the analyses that are going to be presented.

The analysis will be displayed in the form of comments on the existing code, so that it will be possible to visualize the cost of each instruction step by step, making it easier to understand how the estimate has been computed.

drawNumbers()

First, it has been decided to estimate the gas cost of *drawNumbers()*. In Listing 1 the estimate of its two modifiers, in Listing 2 the estimate related to the *_random()* method called inside, finally in Listing 3 the estimate of the main method.

Listing 1: Gas cost estimate of *drawNumbers()* modifiers.

```
1  modifier ownerOnly() {
2      require(_owner == msg.sender, "owner only");
3      // SLOAD(cold): 2.100 (_owner)
4      // CALLER: 2 (msg.sender)
5      // EQ: 3 (_owner == msg.sender)
```

[4] https://ethereum.github.io/yellowpaper/paper.pdf

```
    //
    // Total: 2.105 approx.
    _;
}

modifier inactiveRoundOnly() {
    require(_currentRound != 0 && block.number >= _startBlock + _duration && !_prizesAssigned, "inactive round only");
    // SLOAD(cold): 2.100 (_currentRound)
    // EQ: 3 (_currentRound != 0)
    // NUMBER: 2 (block-number)
    // SLOAD(cold): 2.100 (_startBlock)
    // SLOAD(cold): 2.100 (_duration)
    // ADD: 3
    // AND: 3
    // SLOAD(cold): 2.100 (_prizesAssigned)
    // EQ/NOT: 3 (!_prizesAssigned)
    // AND: 3
    //
    // Total: 8.417 approx.
    _;
}
```

Listing 2: Gas cost estimate of *_random()* method called inside *drawNumbers()*.

```
function _random(uint seed) internal view returns(uint) {
    return uint(keccak256(abi.encodePacked(block.difficulty, block.timestamp, msg.sender, seed)));
    // DIFFICULTY: 2 (block.difficulty)
    // TIMESTAMP: 2   (block.timestamp)
    // CALLER: 2 (msg.sender)
    // MLOAD: 2 (seed)
    // KECCAK256: 30 computation + (6 * 4) input params
    //
    // Total: 62 approx.
}
```

Listing 3: Gas cost estimate of *drawNumbers()* method.

```
function drawNumbers() public ownerOnly inactiveRoundOnly returns(uint[6] memory) {
    // TRANSACTION: 21.000 (default cost)
    // ownerOnly: 2.105
    // inactiveRoundOnly: 8.417

    require(_winningNumbers[0] == 0, "numbers already drawn");
    // SLOAD(cold): 2.100 (_winningNumbers)
    // EQ: 3

    uint[6] memory aux;
    // MSTORE: 3 (aux)

    // MSTORE: 3 (i = 0)
    for(uint i = 0; i < 5;) { // [5 times]
        // LT: 3

        uint rand = (_random(i) % 69) + 1;
        // MLOAD: 3 (i)
        // MOD: 5
```

```
20          // ADD: 3
21          // _random(): 62
22          // MSTORE: 3 (rand)
23
24          // MLOAD: 3 (rand)
25          // SLOAD(cold): 2.100 (_winningNumbersMapping)
26          // EQ: 3
27          while(_winningNumbersMapping[rand]) { // [1 time on average]
28
29              rand = (_random(rand*7) % 69) + 1;
30              // MLOAD: 3 (rand)
31              // MUL: 5
32              // MOD: 5
33              // ADD: 3
34              // _random(): 62
35              // MSTORE: 3 (rand)
36          }
37
38          aux[i] = rand;
39          // MLOAD: 3 (rand)
40          // MLOAD: 3 (i)
41          // MSTORE: 3 (aux)
42
43          _winningNumbersMapping[aux[i]] = true;
44          // SSTORE: 20.000 (_winningNumbersMapping)
45
46          unchecked { i++; }
47          // MLOAD: 3
48          // ADD: 3
49          // MSTORE: 3
50      }
51
52      aux[5] = (_random(5) % 26) + 1;
53      // _random(i): 62
54      // SLOAD(warm): 100/200 (POWERBALL_RANGE)
55      // MOD: 5
56      // ADD: 3
57      // MSTORE: 3 (aux)
58
59      _winningNumbers = aux;
60      // SSTORE: 20.000 * 6 (_winningNumbers)
61      // SSTORE: 20.000 (_winningNumbers.length)
62
63      emit LogNumbersDrawing(_winningNumbers);
64      // SLOAD(warm): 100/200 (_winningNumbers)
65      // LOG: 375 (emit LogNumbersDrawing)
66      // LOGDATA: 8 x 32 bytes (Event input params)
67
68      return aux; // RET: 0
69
70      // Total: 286.155
71  }
```

A short summary of the results:

- **TOTAL GAS COST**
  - *ownerOnly:* 2.105 approx.

- *inactiveRoundOnly:* 8.417 approx.
- *_random():* 62 approx.
- *drawNumbers():* 286.155 approx.

It is immediate to notice how the most impactful operations are the one related to the storage, this is especially true in the case of storage write operations (*SSTORE*). Notice that the storage read operations (*SLOAD*) are divided in cold access and warm access, where cold access refers to the first time a storage variable is accessed inside a function and warm access refers to all the following accesses.
Finally, it could be worth noticing that, aside from operations on the storage, the second most impactful instruction is the emit of an event.

closeLottery()

In addition, also the gas cost of *closeLottery()* has been estimated (Listing 4)..

Listing 4: Gas cost estimate of *closeLottery()* method.

```
function closeLottery() public ownerOnly {
    // TRANSACTION: 21.000 (default cost)
    // ownerOnly: 2.105


    // SLOAD(cold): 2.100
    // EQ/NOT: 3 (!_prizesAssigned)
    if(!_prizesAssigned) {

        // MSTORE: 3 (i = 0)
        // Loop executed only if closeLottery() called on not endend round
        for(uint i = 0; i < _players.length;) { // [N times]
            // SLOAD(cold): 2.100 (_players.length) the first time
            // SLOAD(warm): 100/200 (_players.length) after
            // LT: 3 (i < _players.length)

            emit LogTicketRefund(_players[i], _addressToAmount[_players[i]]);
            // SLOAD(cold): 2.100 (_players[i])
            // SLOAD(cold): 2.100 (_addressToAmount[_players[i]])
            // LOG: 375 (emit LogTicketRefund)
            // LOGDATA: 8 * 64 bytes (Event input params)

            payable(_players[i]).transfer(_addressToAmount[_players[i]]);
            // SLOAD(warm): 100/200 (_players[i])
            // SLOAD(cold): 2.100 (_addressToAmount)
            // TRANSFER: 9.000


            unchecked { i++; }
            // MLOAD: 3
            // ADD: 3
            // MSTORE: 3
        }
    }

    emit LogDeactivation(_withdrawal);
    // SLOAD(cold): 2.100 (_withdrawal)
    // LOG: 375 (emit LogTicketRefund)
    // LOGDATA: 8 * 32 bytes (Event input params)

    selfdestruct(payable(_withdrawal));
```

```
42      // SELFDESTRUCT: 5.000
43      // (24.000 gas are refunded when destructing a contract)
44
45      // Assuming no for loop execution, for simplicity
46      // Total: 58.147 approx.
47  }
```

A short summary of the results:

- **TOTAL GAS COST**
  - *ownerOnly:* 2.105 approx.
  - *closeLottery():* 58.147 approx.

Security

This section will briefly describe the main security vulnerabilities that have been discussed during the lectures of the course, for each one what should be avoided and a possible prevention approach. Then, we are going to discuss the security aspects of our project implementation.

- **Reentrancy:** avoid *.call.value*.
  - **Prevention**
    - use *.send* or *.transfer* (which is the most secure option).
- **Arithmetic overflow/underflow:** avoid Solidity arithmetic directly.
  - **Prevention**
    - use *SafeMath* library or *pragma solidity >= 0.8.0*, which implements the mentioned library by default.
- **Front Running:** avoid not having a gas price upper-bound.
  - **Prevention**
    - set an upper-bound on gas price at deployment (does not work if attackers are miners).
- **Miner attack:** avoid timestamp for entropy and time sensitive decisions on small differences.
- **Randomness:** avoid blockchain as randomness source.
  - **Prevention**
    - use an oracle (e.g. Chainlink VRF)
- **Pishing attack:** avoid using *tx.origin* for authentication.
  - **Prevention** use *msg.sender* instead.

In our specific case all the vulnerabilities are in some way prevented by following the prevention approaches that have been discussed, the only exceptions are miner attacks and vulnerabilities related to the generation of random numbers.

This is due to the fact that the lottery uses as randomness source data from the blockchain, which is public and visible to anyone. In fact, attackers could try to predict the input parameters of the KECCAK256 and possibly know the winning numbers of the lottery before the official drawing. This vulnerability is in some way prevented by concatenating the *block.difficulty* and the *block.timestamp* before computing the KECCAK256, since it would be very unlikely for someone who is not a miner to be able to predict the timestamp at which the block of the drawing is mined.
However, the problem of miner attack persists. A miner, if given enough incentive (e.g. rare NFT prizes in the lottery), could mine his block with incorrect timestamps to manipulate the outcome of the random function used inside the lottery. To solve this problem it would be enough to have as prizes of the lottery collectibles which values is less than a block reward, as it would make making an attack disadvantageous for the miner.

Finally, notice that using blockchain data as as randomness source was a mandatory requirement of this assignment. Hence it was not possible to use an oracle (e.g. Chainlink VRF), which would be the best approach to avoid such vulnerabilities.

Testing

In order to test and run a meaningful simulation of the system, it is enough to execute the following operations in the given order.

1. Compile the contract *Lottery.sol*
    - enable compiler optimization suggested
2. Deploy the contract
    - set gas price upper-bound to front running suggested
3. *startNewRound()*
    - make sure to use the same address used for deployment
4. *buy(uint[6][])*
    - call this function three times (three ticket purchases) to make the round become inactive
    - set value of payable function, for each ticket 0, 01 Ether (10.000.000.000.000.000 Wei), higher values are also valid
    - standard numbers range is 1-69
    - powerball number range is 1-26
    - the input format of the tickets is a list of lists, for instance,
        - $[[1, 2, 3, 4, 5, 6]]$ buys a ticket with powerball 6
        - $[[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]$ buys two tickets with powerball 6 and 12
5. *drawNumbers()*
6. *givePrizes()*

After, it is possible to start a new round and run a new simulation by repeating the steps 3-6. In addition, at any moment the contract can be deactivated by executing *closeLottery()* or a new NFT of rank *prizeRank* can be minted by executing *mint(uint prizeRank)*.

Notice that for each transaction it is possible to analyze the events that have been emitted, to gain additional information about what is happening in the system.

Appendix A

Name	Value	Description
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{jumpdest}	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
G_{verylow}	3	Amount of gas to pay for operations of the set W_{verylow}.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{\text{warmaccess}}$	100	Cost of a warm account or storage access.
$G_{\text{accesslistaddress}}$	2400	Cost of warming up an account with the access list.
$G_{\text{accessliststorage}}$	1900	Cost of warming up a storage with the access list.
$G_{\text{coldaccountaccess}}$	2600	Cost of a cold account access.
$G_{\text{coldsload}}$	2100	Cost of a cold storage access.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{\text{selfdestruct}}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{\text{selfdestruct}}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{\text{codedeposit}}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
$G_{\text{callvalue}}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{\text{callstipend}}$	2300	A stipend for the called contract subtracted from $G_{\text{callvalue}}$ for a non-zero value transfer.
$G_{\text{newaccount}}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
G_{expbyte}	50	Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
G_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead* transition.
$G_{\text{txdatazero}}$	4	Paid for every zero byte of data or code for a transaction.
$G_{\text{txdatanonzero}}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{\text{transaction}}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
G_{logdata}	8	Paid for each byte in a LOG operation's data.
G_{logtopic}	375	Paid for each topic of a LOG operation.
$G_{\text{keccak256}}$	30	Paid for each KECCAK256 operation.
$G_{\text{keccak256word}}$	6	Paid for each word (rounded up) for input data to a KECCAK256 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{\text{blockhash}}$	20	Payment for each BLOCKHASH operation.

Figure 2: Ethereum operations gas cost.

$W_{\text{zero}} = \{\text{STOP, RETURN, REVERT}\}$
$W_{\text{base}} = \{\text{ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE,}$
$\text{TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, CHAINID, RETURNDATASIZE, POP, PC, MSIZE, GAS}\}$
$W_{\text{verylow}} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, SHL, SHR, SAR,}$
$\text{CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}\}$
$W_{\text{low}} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND, SELFBALANCE}\}$
$W_{\text{mid}} = \{\text{ADDMOD, MULMOD, JUMP}\}$
$W_{\text{high}} = \{\text{JUMPI}\}$
$W_{\text{copy}} = \{\text{CALLDATACOPY, CODECOPY, RETURNDATACOPY}\}$
$W_{\text{call}} = \{\text{CALL, CALLCODE, DELEGATECALL, STATICCALL}\}$
$W_{\text{extaccount}} = \{\text{BALANCE, EXTCODESIZE, EXTCODEHASH}\}$

Figure 3: Ethereum subset of instructions.