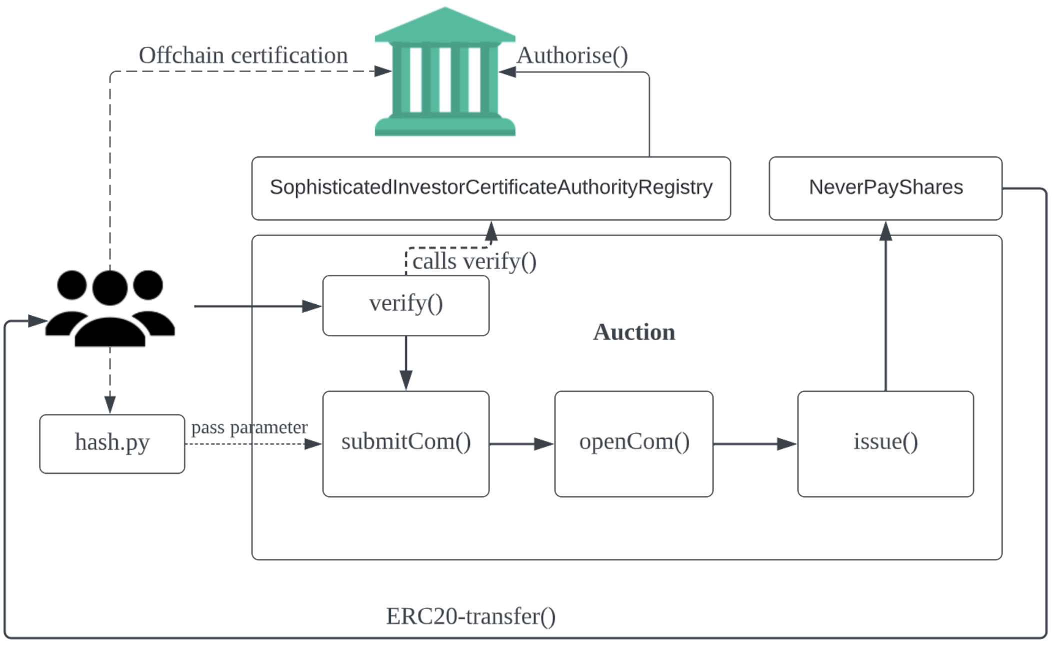
Design and implementation

Overview

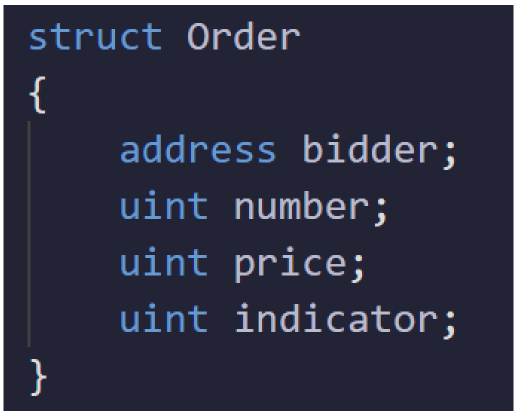


The overall structure of this application is depicted in the diagram above. Totally three contracts are defined for this application, which are registry, shares and Auction. There are four callable functions that are prepared to be called by the investors in turn defined in Auction contract. The investors only need to interact with functions defined in Auction in order to bid for shares. Functions in the other two contracts are called by the code in need.

For the authorized entities. They need to register their public key by letting ASIC to call authorise() function. Afterwards, they are able to issue certificates to sophisticated investors with the corresponding private key using certify.js.

Data model

There is only one core data field: order. So I created a struct called Order defined as below:

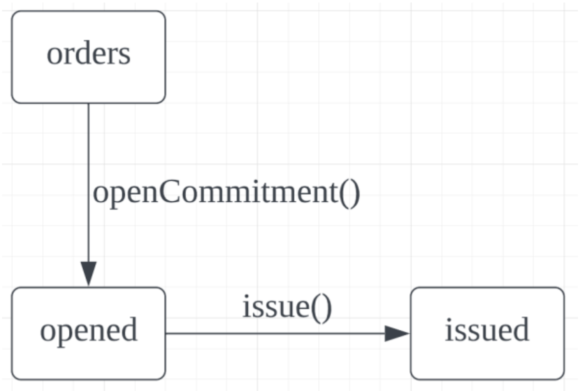


```
struct Order
{
    address bidder;
    uint number;
    uint price;
    uint indicator;
}
```

There are four factors associated with an order: address of bidder, unit of shares purchased, price per share and an indicator.

The indicator is a generated ordering indicator used for sorting the orders, this will be introduced in detail later.

Then three mapping variables are defined to keep track of order status which are: *orders*, *opened* and *issued*.



Orders is a mapping variable from byte32 to uint256 while *opened* and *issued* are mapping from byte32 to Boolean indicating whether or not a specific order is opened or issued.

The reason why the *orders* variable is mapped to uint256 is that we need to record the sequence that a order is submitted for sorting purpose. this will be discussed in detail in the section introducing sorting strategy.

The byte32 value used in these three mapping variables is the unique order hash, which is calculated from *number*, *price*, *address*, *nonce* associated with the order. The reason why the system records the order hash only in the first round is that block information is public, we need to ensure that the information of the orderbook is not visible from any one until the orders are revealed. But we also need to ensure that an order can be recorded reliably in round one. So, we encode the critical values of an order using keccak256 hash function to achieve that. This hash function hash a satisfying collision resistance and is hard to be decoded.

A array of struct Order called *openOrders* is defined to record the information of all opened orders. This array will be sorted for issuing.

Off-chain computation

Order hash

An investor must compute the order hash locally to ensure that others don't know the exact information about his order. So, we provide a hashing function called *hash.py*. Investors can run this script by simply calling python hash.py in terminal and enter the information following the prompt:

```
python hash.py
Number:5
Price:1000
identifier/nonce:1
Address:0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
Order hash is:  0xcd5398fa0105917c4aa0d0e62f9418d4c9f37343c624875add06a47981904446
Hash copied to paperclip.
```

Crypto.Hash, eth_abi, web3 and pyperclip are required as support package. Users can install these packages by the following commands:

pip install eth_abi

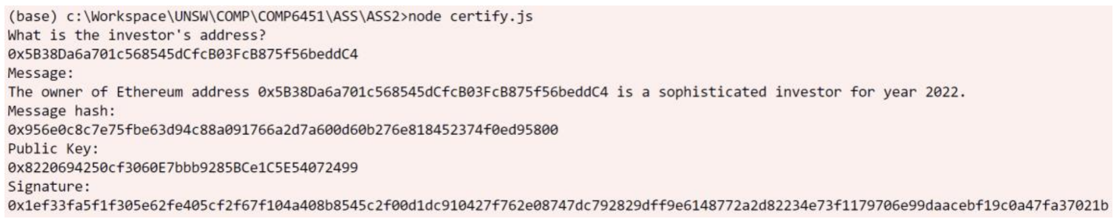
pip install web3

pip install pyperclip

pip install pycryptodome

Signature

An organization trusted by ASIC can run certify.js to issue certificates to an investor. They can run the script by typing node certify.js in terminal. And they should enter the address of the investor. After that, A series of relevant information would be prompted.



```
(base) c:\Workspace\UNSW\COMP\COMP6451\ASS\ASS2>node certify.js
What is the investor's address?
0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
Message:
The owner of Ethereum address 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 is a sophisticated investor for year 2022.
Message hash:
0x956e0c8c7e75fbe63d94c88a091766a2d7a600d60b276e818452374f0ed95800
Public Key:
0x8220694250cf3060E7bbb9285BCe1C5E54072499
Signature:
0x1ef33fa5f1f305e62fe405cf2f67f104a408b8545c2f00d1dc910427f762e08747dc792829dff9e6148772a2d82234e73f1179706e99daacebf19c0a47fa37021b
```

Packages used are: web3-eth-accounts, web3-eth-abi and web3-utils. Users can install them using *npm install <packet name>*.

Requirement analysis

1. A total of 10,000 shares are to be sold.

```
uint constant numberShares = 10000;        // Maximum number of shares
```

A constant numberShares is defined in the contract to explicitly limit the number of shares. This variable is used in constructing the NeverPayShares contract and issuing the shares. This is the pseudo code of issuing process.

```
uint remainingShares = numberShares;
for each order of the bidder:
{
    Issue shares to bidder;
    Settle and refund;
    decrement remainingShares;
}
Mark bidder issued;
```

2. The minimum price per share is 1 Ether, any bid to buy shares for less than this amount will not be issued shares.

Price is checked when opening the orders, if a price that is lower than 1 Ether is passed to the function openCommitment(), the transaction will revert and an error message will be returned. Relevant code fragment shown below:

```
uint minPrice = 1000000000000000000; // Valued by Wei: 1ETH = 10^18 Wei

require(
    price >= minPrice,
    "The minimum price per share is 1 Ether."
);
```

3. Investors should be identified by means of an anonymous Ethereum address.

Address is used to identify the bidders in the order struct.

4. First round

A function called *submitCommitment* is defined to implement the functionality of round 1.

```
function submitCommitment(bytes32 newOrder) public {
    require(
        //Timestamp for 20-04-2022-00:00:00-GMT
        block.timestamp < 1650412800,
        "The first round has ended."
    );
    if (orders[newOrder] != 0)
        return;
    numOrders++;
    orders[newOrder] = numOrders;
}
```

Firstly, we check that the order is submitted before April 20, 2022 by checking the timestamp of current block.

And then we check that whether this order has already been submitted. If it is a new order, *numOrders* is incremented and the sequence number of this order is set to the new value of *numOrders*.

blinded commitment

Notice that all investors are provided a script to generate the order hash, so the parameter newOrder has already incorporates the price and amount the bidder is willing to offer.

Because the keccak256 has good collision resistance, it is impossible to get the order information solely based on the information shown in the block.

An investor may submit multiple bids in the first round.

There is no limit on the number of orders submitted in the first round. In addition, to ensure that two orders with the same price and amount to offer can be submitted multiple times, a nonce is added in computing the *orderHash*.

Any bid may be withdrawn until the first round deadline.

The withdraw is implemented by a withdraw function that receive the orderHash that the message sender wants to withdraw. After checking that he is one who submitted the order, the system will set the order sequence to zero to indicate that the order is withdrawn.

5. Second round

A function called *openCommitment* is defined to implement the functionality of round 2.

```
function openCommitment(bytes32 orderHash, uint number, uint price,
uint nonce) payable public {
    require(
        block.timestamp >= 1650412800,
        "The second round hasn't started.");
    require(
        //Timestamp for 27-04-2022-00:00:00-GMT
        block.timestamp < 1651017600,
        "The second round has ended.");
        Other checks on order detail.
        openOrders.push(Order(msg.sender, number, price,
10000000000*price+orders[orderHash]));

        opened[orderHash] = true;
        quickSort(openOrders, 0, int(openOrders.length)-1);
    }
```

In this function, time stamp of current block is check at first to ensure that the orders can be opened in appropriate time interval. Then a series of value check of order information is checked. If they are all passed, struct *Order* will be constructed and pushed to *openOrders* array and set the status of current order *opened*. Then perform quick sort once to ensure that *openOrders* is correctly sorted.

No cheat, only reveal what you submit in the first round

The function checks the validity of information submitted and then compute the order hash using the information passed to it. If neither the computed value match the hash value provided by the bidder nor it is recorded in the *orders* variable, this request would be rejected.

In this round, the investor is also required to submit their payment in full

The function openCommitment is set payable so that it can accept payment. After validation the order, this function would check that *msg.value* is the same as the required payment (number of shares × price per share), if mismatch detected, this transaction would be reverted.

After the deadline for the end of the second, only paid orders become valid

Only paid orders are marked *opened* and pushed into *openOrders*. These variables will be used in issue() function that issues shares to the investors.

6. Share issuing

A function called issue() is defined to issue shares and settle the orders. This function firstly checks the timestamp of current block and whether the message senders has already called the issue function for themselves. After that, a issue process is performed, the pseudo code is shown below:

```
for each order in openOrders:
    if msg.sender is not order.bidder:
        Decrement remaining Shares;
    if (order.number <= remainingShares)
        Settle price;
        Transfer shares;
        remainingShares -= openOrders[i].number;
    else
        refund the not used Ethers.
        if (remainingShares != 0)
            Settle price with remaining shares;
            Transfer available shares;
        remainingShares = 0;
issued[msg.sender] = true;
```

The function inspects all the opened orders in defined order and check whether the message sender is the one who places the order. Then the function initiates settle(), refund() or transfer() functions to issue shares or refund to the bidders. And the value remaining shares is always decremented whether or not the message sender is the bidder of the order. This ensures that the shares are only issued to the highest prices. Once remaining shares reaches zero, no shares would be issued and only refund would be called.

Ordered issuing

To ensure that the orders are correctly ordered and sorted by two keys: price and submission date. A dedicated data field called indicator is used to be compared when sorting. Notice that, when creating a new Order instance, the indicator is set to

```
10000000000*price-orders[orderHash]
```

This means that the earlier the order is submitted, the higher the indicator would be. Thus, if two orders offer the same price, the index of the order that is submitted earlier would be smaller.

Multiplying price by 10000000000 permits 10^{10} orders to have the same price. It is not likely to have more than this number of orders placed with the same price. In addition, the number space of uint256 is approximately 10^{77} and the base price is 10^{18} Wei, multiplying price by 10^{10} cannot cause arithmetic overflow.

Refund

A function refund() is defined to implement the refund functionality. The address of bidder is recorded in each Order instance. If an order is opened but not fully issued, the contract would refund the unused Ethers back to the bidder.

7. Investors can transfer their shares to another investor

A contract called NeverPayShares is defined using standard ERC20 interface. This contract is instantiated and made public in the constructor function of Auction contract. Also, the issuing process is just done by transferring shares from the Auction contract to shareholders. Once a investor receives the shares, he can transfer their shares with the address of NeverPayShares using ERC20 operations.

8. Costs

The smart contract should be cost effective for the NeverPay to run

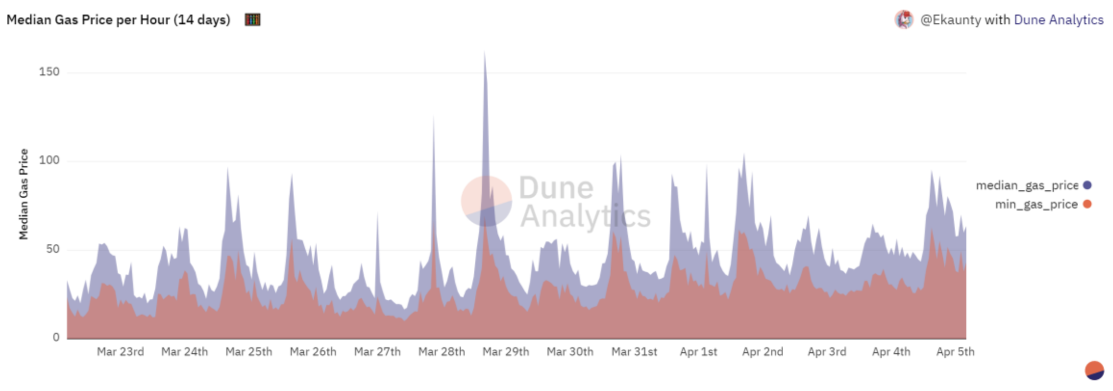
Almost every function is called by the investors. The investors must call submitCommitment(), openCommitment() and issue() by them selves so the related gas fee are borne by the investors.

One investor may only pay the transaction cost for himself

Each stage of the auction process is self-serviced. The investors call the functions on behalf of their orders only. Calling issue function will also issue the shares belong to the message sender only.

Costs analysis

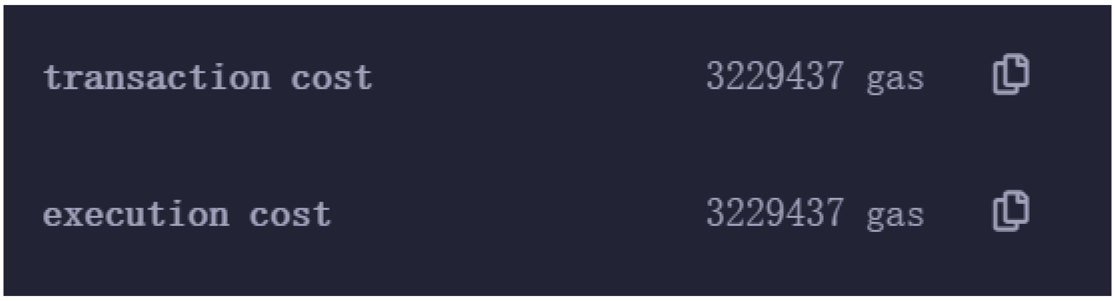
Current gas fee



According to the historical data of the past 14 days, Both Never Pay and investors can expect a gas fee lower than 50 Gwei/gas. For Never Pay and some investors, processing time is not critical factor. So, they can choose to transact in the “cheap hours”. Thus, the 50 Gwei is used to estimate the unit gas fee for them. But ordering of submission may matter in round 1 if ties between bids making the same price offer happens. For those investors, 75 is used conservatively.

Never Pay

Because each round is self-serviced, the major cost for Never Pay is an up-front deployment cost. According to the result of establishing the contract, the estimated gas cost is 3,229,437 units of gas:



```
transaction cost          3229437 gas

execution cost            3229437 gas
```

The estimated deployment cost is then (Exact number is used because deployment cost tends to be uniform).

$$3{,}229{,}437 * 50 = 161{,}471{,}850\ Gwei \approx 161.47\ PWei \approx 0.161\ Eth$$

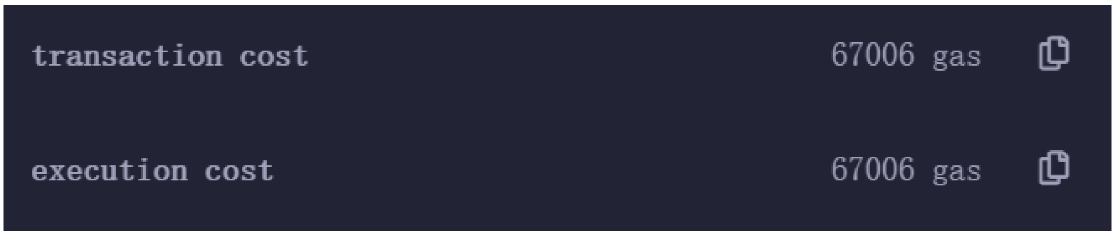
The cost is considerable small comparing to the fund that is going to be raised.

Investors

Here are 4 functions that are required by the investors to bid for shares, which are verify, submitCommitment, openCommitment and issue. Because not every investor can go through the whole process, these four functions will be analyzed separately. Each transaction is tested for multiple times to avoid extreme cases.

Verify

The related transaction cost of verifying a sophisticated investor is around 67,000 units of gas.



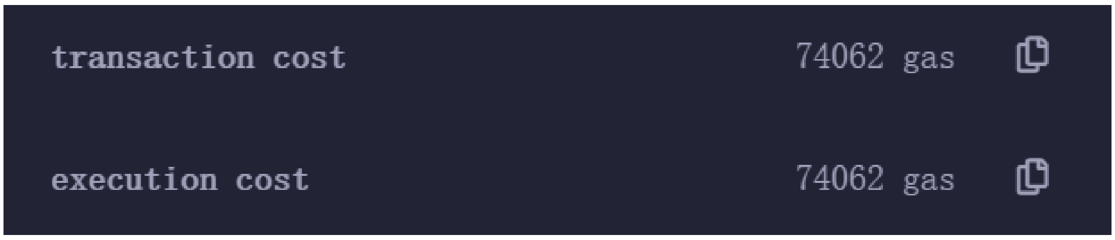
```
transaction cost          67006 gas
execution cost            67006 gas
```

The expected cost is then

$$67{,}000 * 50 = 3{,}350{,}000\ Gwei = 3.35\ Pwei = 0.00335\ Eth$$

submitCommitment

The transaction cost of submitting a commitment is around 74,000 units of gas.



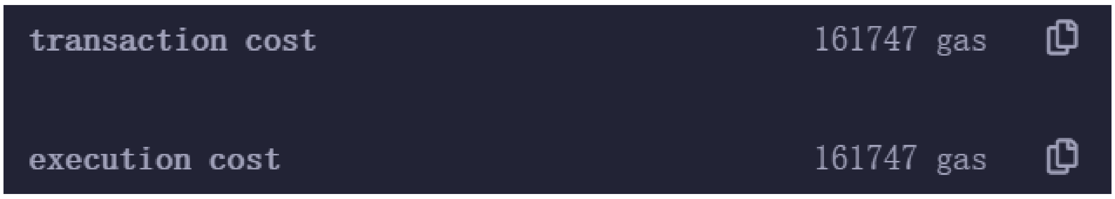
```
transaction cost          74062 gas
execution cost            74062 gas
```

The expected cost is then

$$74{,}000 * 50 = 3{,}700{,}000\ Gwei = 3.7\ Pwei = 0.0037\ Eth$$

openCommitment

The base transaction cost of opening the first commitment is around 160,000 units of gas.



```
transaction cost          161747 gas
execution cost            161747 gas
```

But the cost is not constant due to the function quickSort() called inside this function. The running cost of quicksort grows with number of orders opened, the more orders opened, the more gas is required to sort. Because the target that is going to be sorted openOrders is always sorted before each insertion. And the pivot is selected in the middle of the array. The complexity of each call quick sort is $O(logn)$. The constant associated with the complexity is on thousand level because arithmetic operations only cost single-digits. Even loading and storing cost a lot more, they are only called once when operating the stored array. Each time quick sort is called, the case is always inserting an element to an already sorted array. Therefore, storing is only called twice per recursion for swapping.

We assume that M is the constant factor of the running cost of quick sort, then the expected total cost of openCommitment becomes:

$$Mlog_2 n + 160000$$

And the gas fee is:

$$50Mlog_2 n + 8{,}000{,}000\ GWei = 8PWei + 50Mlog_2 n\ GWei.$$

If we assume M to be 10000, The gas fee is then $8 + 0.5log_2 n\ PWei.$. The first time when the total fee exceeds the base case is that there are already 2^{16} orders opened. Now assume that the transaction cost is 800,000 gas ($40\ Pwei$), n would then becomes 2^{64}. Because corresponding amount of ETH should be locked in round2 to reveal an order, there should be 2^{64} ETHs locked. But the circulating supply of tokens are approximately 120 million, it is not possible to have that much orders opened.

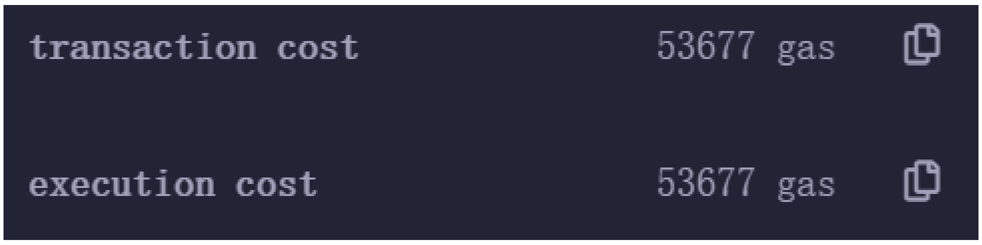
Even for the aggressive investors, the opening cost is just

$$12 + 0.75log_2 n\ PWei$$

So, the cost of opening orders is also acceptable.

Issue

In the base case (no opened shares), around 54,000 gas are used.



```
transaction cost        53677 gas

execution cost          53677 gas
```

By sampling, the approximate increment of transaction cost for each opened order is 2750 gas. And the complexity of this function is apparently $O(n)$. Then the expected cost is:

$$50(2750n + 54000)\ Gwei = 137{,}500 + 2{,}700{,}000\ Gwei \approx 0.14n + 2.7\ Pwei.$$

Summary

The estimated cost of those four functions is, assuming the exchange rate is 4300AUD/ETH:

Function	**Eth cost**	Approx. AUD cost ($)
Verify	$3.35\ Pwei$	14.4
Submit	$3.7\ Pwei$	15.9
Open	$8 + 0.5log_2n\ Pwei$	$34.4 + 2.2\ log_2n$
Issue	$0.14n + 2.7\ Pwei$	$0.6n + 11.6$

It is clear that the costs of verifying and submitting orders are fairly low, this reduces the cost of participating in bidding. With a lower cost, investors are willing to place orders more actively, this is especially beneficial to price investigation. The cost of opening orders is relatively higher comparing with the previous two stages. But the complexity is only logarithm to the number of orders, so investors can expect to pay a nearly constant fee for each order to reveal. Although the complexity of issue is linear to n, the cost may also be acceptable because this function is only called once by each investor.

Suitability of the Ethereum platform

There are several aspects to examine the suitability of an approach of shares offering, they are: fairness, efficiency, cost. The platform chosen should provide equal opportunity for all investors to participate in the auction. The platform should also be capable for investors to perform their strategy effectively with acceptable cost.

The advantage of using Ethereum for this application is to ensure the fairness and transparency of the process. Thanks to the design of the block chain, investors can access the same public information and then to determine their optimal offer. Because the design of the Auction contract, the whole process is decentralised and self-serviced. This ensures that all investors promise to following the same rule by fair competition. Moreover, as the cost of submitting orders is considerably low comparing to the value of the shares. Participants have motivation to place more orders and to design more complicated strategies. This is especially beneficial to price investigation, which is a critical process of shares offering. In addition, ERC-20 provides decent liquidity to the shares, this is also a benefit of using Ethereum platform. To Never Pay, Ethereum is also a reliable platform to avoid any potential loss from cheating.

However, there are also some drawbacks due to the nature of Ethereum block chain. Currently, usually a transaction takes more than ten seconds to be fully confirmed. However, timing is an important factor in bidding process. Especially in round1, the order of submission determines sequence of an order in the openOrder list if tie happens. Although the design of this application aims to solve the timing problem by separating round1 and round2, it is still an inevitable problem. Another concern is that although the process is designed to minimize the cost, it is still much higher than traditional method for investors. However, to put it another way, the cost for issuer can be lower because most costs are borne by investors.

Overall, Ethereum is a good choice for issuing coins, it can guarantee of the quality of both bidding process and the coins themselves. Although there are some inevitable drawbacks, they are still acceptable. Besides, the participants are all sophisticated investors, they care the quality more than cost.

Security

Approach to common vulnerabilities

Re-Entrancy

This application is not subject to re-entrancy attack (Samreen & Alalfi, 2021) for two reasons: the payee of issue() is guaranteed an account address of an sophisticated investor. So the address of payee is not likely to be a contract. Even a malicious contract calls the issue() function, we defined a variable issued that record whether the message sender has already called issue(). We also use transfer to send ethers back to the investors, potential function calls can be avoided by the limited gas allowance.

Arithmetic Over/Under Flows

Design of this application restricts the arithmetic operations to perform. The two main variables that are associate with computing are orderNum to record numbers of orders submitted and remainingShares that records the shares available to be issued. OrderNum is stored in uint256, and it is not possible to submit more than 2^{256} orders. In the computation of remianinShares, we also restrict this value not to be less than zero. So the code is not invulnerable to arithmetic over flows.

Weak Field Modifiers

The private filed modifier may not guarantee the value to be completely invisible to public because any change in value should be published in the blockchain. So, we store the most important information, order detail in hashed form in the first round to avoid being exposed.

Security considerations

Order detail is the most important information in the whole process. To keep this secret, we have already provided a python script to generate order hash. Please run this script locally and do not save the order info in an unsecured digital storage.

One potential security risk is that insufficient gas limit may lead to unexpected behavior. Because we need to ensure that each share is only processed once and each investor only call issue function once. Any call to issue function from the same investor will be rejected. So, please attach sufficient gas fee in order to get the desired shares back. The recommended gas limit is 800,000.

Stretch goal and test

Implementation

Two files are created for verifying a sophisticated investor: Certificates.sol and certify.js. As described before, certify.js is a script for institutions to issue certificates to sophisticated investors. A contract named SophisticatedInvestorCertificateAuthorityRegistry is defined in the solidity file.

Two values are defined in the contract, one is the address of ASIC, and the other is a apingg variable from address to bool to record whether or not a public key is authorized. Four public functions are defined, which are: authorise(), cancel(), isAuthorised() and verify().

Authorise is called by the contract deployer, in another word, ASIC to register a public key on the list. Verify is used to verify a signature, the return value represents the validity of a certificate.

A certificate is generated by signing the message

'The owner of Ethereum address <address> is a sophisticated investor for year 2022.'

using the private key. After signing, the institution will give the public key, message and signature value to the investor.

In order to verify that an address is possessed a sophisticated investor, he need to call verify() function define in Auction contract by passing the public key, message and signature. Then the function will call the verify function defined in the Registry contract. If the result is true, the address of message sender will be recorded verified. Verification process is done by using ecrecover() to induce the public key from the message and signature. The function returns true if the given public key is equal to the recovered public key.

Test

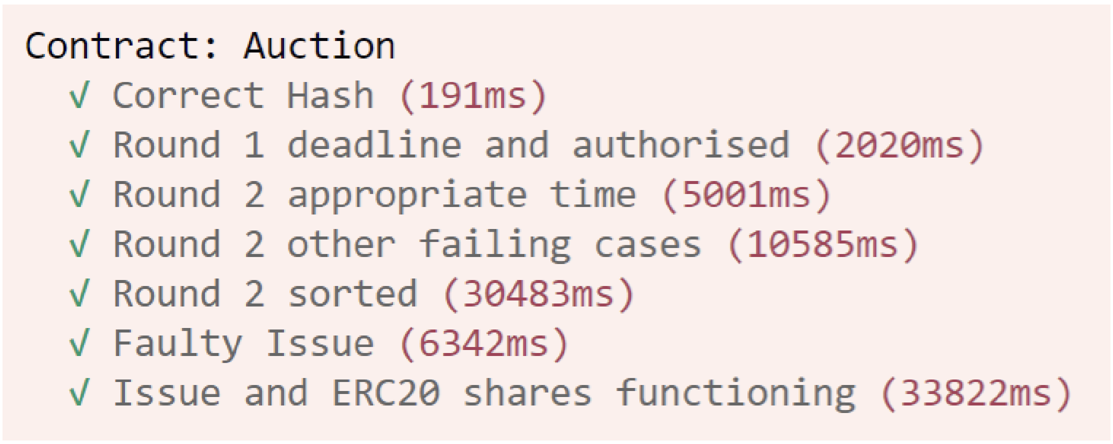
A truffle test script called BiddingTest.js is ready to run by typing *truffle test*. Before testing, a local ganache instance is needed. The recommended setup of the ganache instance is:

1. Address: 127.0.0.1
2. Port: 7545
3. Accounts: 10
4. Initial fund: 100,000 ETH

A package called ganache-time-traveler is used in the test script, please install the package with npm.

npm i ganache-time-traveler

I constructed unit tests for each functionality in different scenarios. There are seven test cases:



```
Contract: Auction
  √ Correct Hash (191ms)
  √ Round 1 deadline and authorised (2020ms)
  √ Round 2 appropriate time (5001ms)
  √ Round 2 other failing cases (10585ms)
  √ Round 2 sorted (30483ms)
  √ Faulty Issue (6342ms)
  √ Issue and ERC20 shares functioning (33822ms)
```

1. Correct Hash: Verify that the hash of off-chain script is the same as the hash behavior expected by the solidity contract.
2. Round 1 deadline and authorized: Any late submission and submissions from unauthorized account is rejected.
3. Round 2 appropriate time: Check that the orders can only be opened in appropriate time slots.
4. Round 2 other failing cases: Test for the constraints/assumptions for round 2 separately e.g. wrong orderHash, invalid number/price, incorrect eth transferred.
5. Round 2 sorted: Check that if the orders are correctly sorted.
6. Faulty Issue: Test for invalid call to issue() including: inappropriate time and no orders opened.

7. Issue and ERC20 shares functioning: Test a Success IPO and test the functions of ERC20 tokens like transferring and approving.

(https://www.npmjs.com/package/ganache-time-traveler)

Manual test is also available, please follow the steps:

1. Comment require(); phrases marked by "COMMENT THIS".
2. Deploy Certificates.sol
3. Generate Signatures using certify.js
4. Deploy Auction.sol with the address of SophisticatedInvestorCertificateAuthorityRegistry contract
5. Verify using pubKey(signer), Message and signature (notice that, the address that is going to be verified is the address of msg.sender to verify()).
6. Generate order hash using hash.py
7. Submit orders, open orders, issue().
8. Try to trade with NeverPayShares.

Reference

SamreenFatimaNoama, & AlalfiHManar. (2021). A Survey of Security Vulnerabilities in Ethereum Smart. arXiv.