Blockchains & Distributed Ledgers

School of Informatics, University of Edinburgh

Ethereum Smart Contract Lottery

s1771232 - Gregory D. Hill

School of Informatics **November 2017**

Ethereum Smart Contract Lottery

1 Contract

1.1 Description

Ethereum provides a blockchain with a Turing complete programming language [2]. Developers can deploy arbitrary, consensus-based, decentralised applications. One such 'smart contract' is defined within this document. It exchanges wei for an internal currency, which is used to weight the random selection of a winner who receives the entire balance of ether after a certain threshold has been hit. Algorithm 1 defines the main pseudo-code for this process.

Algorithm 1 Exchange & Selection

```
global variables
    weight ← 0
    counter ← 0
    party ← [address → uint]                                ▷ mapping
end global variables
procedure CHOOSE_WINNER
    seed ← H (head)                                         ▷ hash of the newest block
    random ← (seed mod weight) + 1                          ▷ random value between 1 and total token count
    queue ← quicksort (party, 0, length(party) - 1)         ▷ sort token holders from least to most
    for user in queue do
        random += user (tokens)
        winner ← user
        if random ≥ weight then                             ▷ select winner
            break
    transfer contract balance to winner
procedure EXCHANGE (address, wei)
    require wei > 0
    weight += wei / 100
    party [address] ← wei / 100
    if balance ≥ 500000000000000000 then                    ▷ 0.5 ether
        choose_winner
```

The contract maintains a dynamic 'queue' of players who can actively update their balances. Although not fully defined in Algorithm 1, because Solidity restricts access to their interpretation of a hash-map, this was achieved with two 'mapping' variables. The first one links addresses to tokens (the party) and the second one maintains a queue of addresses based on a counter. When a new player interacts with the contract, it will first check if the account already exists in its party by evaluating the associated balance and then incrementing the queue with this new address if the balance is not greater than zero. Else it will just add to the player's previous balance. This mechanism enables the contract to iterate through the queue with membership queries based entirely on the number of players which is essential for sorting the list as required in Section 2.

After each new conversation, if the contract's total running balance is ≥ 0.5 ether then it will select a new winner proportional to the number of tokens they hold. The contract will then transfer its entire account balance to the chosen winner.

1.2 Address

The contract was deployed onto the private block chain, mined at the following location:

`0x8b031f3c67321899daa103db8e7a24cd5c44333c`

Algorithm 2 Basic Quicksort

```
procedure QUICKSORT(array, left, right)
    if left < right then
        p ← partition(array, left, right)
        quicksort(array, left, p-1)
        quicksort(array, p+1, right)
procedure PARTITION(array, left, right)
    pivot = left
    for i = left+1 to right do
        if array[i] ≤ array[left] then
            pivot += 1
            swap array[i] with array[pivot]
    swap array[left] with array[pivot]
    return pivot
```

2 Analysis

It is hard to ensure that publicly verifiable randomness is secure [6]. Most decentralised applications typically take some form of entropy from the environment. For instance, one method suggested in Ethereum's original whitepaper [2] was to use the hash of the previous block as a source of randomness. This approach similarly seeds a random value based on the hash of the current head of the blockchain, modulo the total token balance plus one. This should return a random value between one and the total weight.

In probability theory, values are defined between zero and one. Unfortunately, floating points are not managed by Solidity which means that any probabilistic values are required to be positive (unsigned) integers. So the ideal distribution would be discrete with probabilities equivalent to the number of tokens held divided by the total, such that all sum to one. But as the contract can not deal in floating point precision, the division is removed such that the sum of all tokens is equal to the total expected weight.

The list of participants is sorted by the number of tokens in ascending order, using a rudimentary in-place recursive quicksort, as defined in Algorithm 2. Each count is then added to the random value until it surpasses the total weight (which was initially used as the modulus to obtain the random value). Therefore, in this weighted probabilistic draw, the likelihood of winning for a player who traded a large number of wei is very high, but the lower weights are still probable if the random value is higher. If the list was sorted into descending order, then a player with fewer tokens is very unlikely to be chosen. Another solution for weighting the draw would be to grow the list with an equivalent number of elements, iterate as before and pick the corresponding address. However this would incur a significant memory cost which is undesirable.

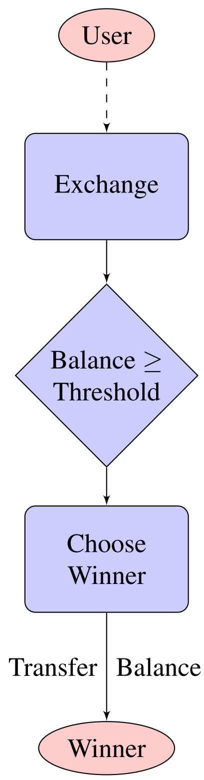

Figure 1: Lottery

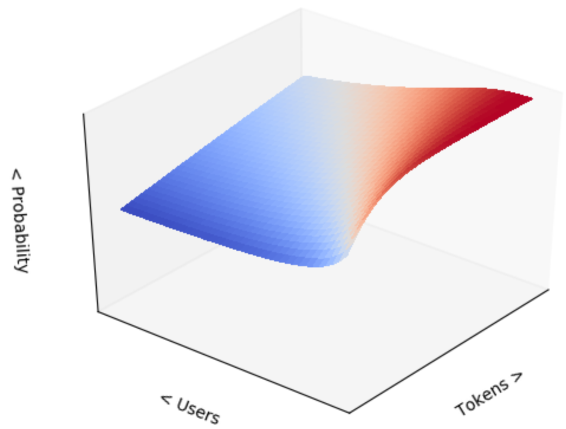

Figure 2: Probability Distribution Over Weighted Selection

Figure 2 illustrates how the probability of a player winning changes with the number of competitors and tokens. In red we can see that a player with more tokens and less competition has a higher likelihood of winning versus less tokens and more competition in blue.

A player's likelihood in the ideal distribution is $t/\sum_{n=1}^{N} t^{(n)}$ where t is the number of tokens an individual owns divided by the sum of all t - the total weight. In this paper's distribution, any one likelihood is technically equal to t. We can measure the variation between these using statistical distance, where X is the ideal distribution and Y is our definition. Scaling Y forces it to reside in the same probability space S for analysis, which is done by dividing t by the total weight as before. Therefore, as the two distributions are equal, the statistical distance is zero which means this is already the ideal distribution.

$$
\begin{aligned}
\Delta[X,Y] =& \frac{1}{2}\sum_{s\in S}|P[X=s]-P[Y=s]| \\
=& \frac{1}{2}\sum_{s\in S}|\frac{t^{(s)}}{\sum_{n=1}^{N} t^{(n)}} - \frac{t^{(s)}}{\sum_{n=1}^{N} t^{(n)}}| = 0
\end{aligned} \tag{1}
$$

3 Deployment & Engagement

The MetaMask bridge [5] for Ethereum is designed to simplify blockchain interaction. This meant that the smart contract code could be compiled and deployed directly onto the private blockchain from Remix in Firefox. Once the contract had been sufficiently tested in Remix's *JavaScript VM*, the environment was changed to *Injected Web3* which allowed MetaMask to catch the transaction and send it directly to the University's private blockchain. The contract was published on November 9th (2017) at 11:29 am.

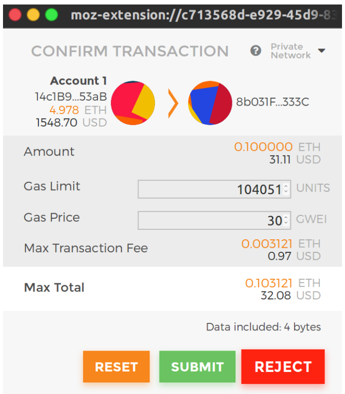

Figure 3: MetaMask Exchange

One further transaction was submitted on November 9th (2017) at 11:37 am. This was to exchange 0.1 ether with the contract from a custom address:

`0x14c1B930989c59e44c2172A9240cdb5AE2f153aB`

Figure 3 shows the MetaMask confirmation transaction box presented before submitting the transaction. We can see that 0.1 ether was equal to approximately 31.11 United States Dollars (USD) at time of writing. There was also an additional 0.003121 ether surcharge for gas fees which amounted to less than 1 USD.

The first transaction made to the contract was by *Andres Monteoliva* who submitted 0.2 ether. This incurred a transactional fee of 74051 gas.

`0x22a32dE7633c11E0eb5A75fD39d04eA3A4F5244C`

Jinming Cui subsequently submitted two payments. The first was 0.05 ether, which cost 74051 gas. The second update to his balance was 0.1 ether, met by a final surcharge of 33287 gas.

`0xe23d31e274631b84b44100d724d0d2bbd8ba4451`

The final tester of the contract was *Maqing Gao* who triggered the winner selection after trading 0.3 ether. With four unsorted players, her fees amounted to 158310 gas.

`0xe03a322f17a9dd6fbce69bfbc5dde85d75216146`

4 Gas Consumption

The gas price of a transaction denotes the computational costs incurred by the server in terms of wei per unit [7]. There is also a limit to halt complex operations that could result in denial of service conditions [2]. Alternatively, if the contract terminates earlier than expected, the client will receive a partial refund, should any gas remain. It is therefore important to optimise solidity code to run on minimal resources and prevent the users of a contract from being overcharged.

Transaction costs are determined by the size of the data sent to the blockchain. This includes the base cost of 21000 and creation cost of 32000 gas [7]. The remaining costs are determined by the number of bytes. The execution cost is dictated by the cost of storage and computation - as performed by the miner. Table 1 presents some test costs for deploying and running the contract with up to five clientèle in the Remix JavaScript VM environment.

Action	Condition	Transaction Cost	Execution Cost	
Deployment	Contract Mined	770964 gas	551556 gas	
Exchange	1^{st} Player	104051 gas	82779 gas	Balance $<$ Threshold
Exchange	Player > 1	74051 gas	52779 gas	
Selection	1 Player	78840 gas	117568 gas	Balance $\geq$ Threshold
Selection	2 Players, Sorted List	62494 gas	103715 gas	
Selection	2 Players, Unsorted List	62647 gas	104022 gas	
Selection	3 Players, Sorted List	70684 gas	120095 gas	
Selection	3 Players, Unsorted List	71144 gas	121016 gas	
Selection	4 Players, Unsorted List	78553 gas	135833 gas	
Selection	4 Players, Sorted List	78990 gas	136708 gas	
Selection	5 Players, Unsorted List	85923 gas	150574 gas	
Selection	5 Players, Sorted List	87413 gas	153553 gas	
Selection	5 Players, 0.1 Ether	88948 gas	156623 gas	

Table 1: Test Costs

Disregarding the deployment cost, the biggest transactional cost was that incurred by the 1^{st} player. Prior to this, the contract was not subject to storage fees so subsequent players are presumably not liable to the same initialization costs - because they merely increment the count.

The selection process was designed to be triggered when the contract's balance was $\geq$ 0.5 ether. Therefore, the results from the third section in Table 1 were collected from the last player who met this condition. It is clear that the costs not only scale with the number of users, but increase depending on the list's prior layout. Mathematically, quicksort's [4] computational complexity to sort n items is $O(n \log n)$ on average, but can increase to $O(n^2)$ in the worst case. Depending on the exchange amount, a substantially larger number may also increase execution costs in the conversion process due to division.

When deployed onto the private blockchain, the transaction cost for creation was measured to be 728168 gas which is 42796 units less than the cost determined in Table 1. The first transaction cost for exchange (Figure 3) measured at 104051 gas - equal to that observed when testing.

5 Improvements

From the results highlighted above in Section 4, the smart contract is clearly not fair to all players. For example, the 1st player incurs a higher gas cost when interacting with the contract. This is obviously not ideal as every user should be charged at the same rate, regardless of when they exchange funds. This is solvable with pre-initialisation of the respective variables.

The gas costs for selection are charged to the user who exceeds the pre-defined threshold. This scales with the number of users and the original order of input. Therefore, the trader that initiates the final selection is variably charged as in Table 1. For instance, a shorter, or more easily sorted queue, would incur less transaction fees than the worst case 'pre-sorted' scenario. Regardless, the act of charging the final participant is not fair. An approach to solving this problem would be to make the winner selection function public, so anyone can run it (providing the balance $\geq$ threshold). Though this incurs an additional problem. Even when public membership queries are disallowed, by nature of the ethereum blockchain, an adversary could actively read transactional data pertaining to the contract and build a map of users and token counts. The adversary could then trigger the selection process when certain of victory - based on the publicly available random value. To guarantee input-independence, it might make sense to build a scheme which allows users to first participate in the lottery with a hash of the sum that they wish to bet. In another round the users would then send wei to the contract, though this could also be observed by an adversary who may choose to abort. The only way to guarantee that all parties comply to this procedure would be to introduce a maximum cap and an additional round of interaction. Whereby each user would send a commitment (hash) of their desired bet (under the cap), a nonce and the exact maximum amount of ether required. After the winner is selected, each party would then be forced into a further interaction to reclaim the difference between their bet and the cap. Once all refunds have been issued, the remaining balance should be transferred to the victor as in Figure 4. A time limit could also be introduced to restart the lottery procedure based, for example, on the number of blocks mined since the winner was selected.

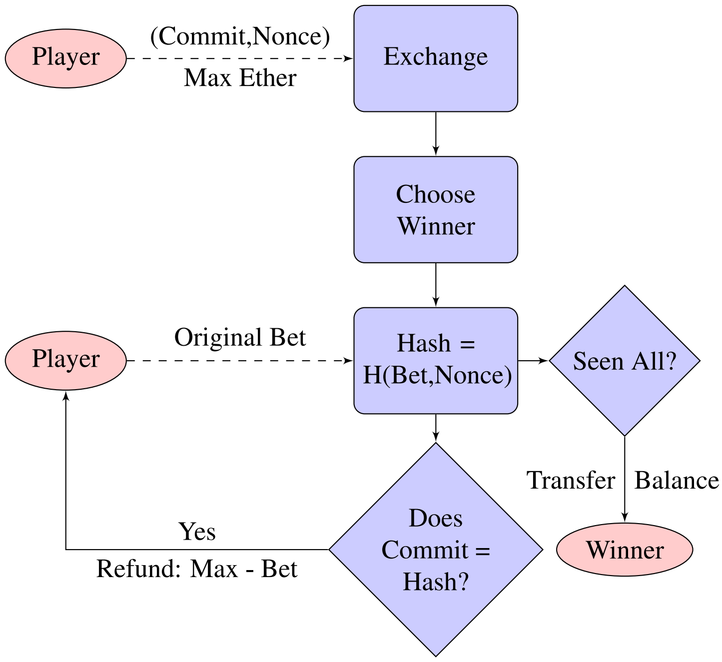

Figure 4: Input-Independent Lottery

Regardless of who pays for the final selection, it will be inherently more costly than other operations due to the fact queue has to be sorted in order for the assumptions made in Section 2 to hold. Hoare's Quicksort algorithm [4] is popular due to its good, balanced, performance. Though by default the previous implementation started with the leftmost value as the pivot, in practice it makes more sense to select a random value - to prevent the worst case in sorting a pre-sorted list. There are also methods to find the median of a set for use as pivot [3], but these are known to incur additional costs over just choosing a random element. Alternatively, instead of sorting the list prior to selection it might make sense to dynamically place users in a queue relative to the number of tokens they bid - under the assumption that their tokens are visible at run-time. One solution for this would be to implement the online insertion sort, but it would actively require more passes through the data.

Section 2 briefly explained the concept of randomness in the ethereum blockchain and presented a solution based on hashing the current head block. Unfortunately, this approach is susceptible to attack. It is possible that an adversary who has the hash of the last block could determine the likelihood of becoming the next winner [1], given that they trade a sufficient number of wei. There are alternate solutions to this problem, but one example builds upon the solution for input-independence in Figure 4. The contract could take the nonce or hash of each party into consideration and generate a unique value that cannot be predicted prior to actually seeing all nonces. Although in this scenario it is plausible that players may provide weak nonces, for example, a coalition may input the same value, or zero. This is known as the entropy hazard.

6 Optimizations

6.1 Fairness & Incentivization

The original contract code (Appendix A.1) was modified to ensure fairness in the initial exchange, incentivize the selection function, and fix a few other unrelated bugs. In the revised code in Appendix A.2, the counter and weight variables were preinitialised to 1. This prevents the 1st trader from being overcharged, hence the general exchange cost for any player was found to be 52499 gas with an overall transaction fee of 73771 gas. Instead of the previous wei threshold, a new player limit was introduced to bound the previously scaled costs of queue sorting and prevent an adversarial coalition from exceeding the maximum call stack or submitting enough transactions to render winner selection infeasible in terms of gas. A refund mechanic was developed to respond to players who joined the draw after a test maximum of 5 'stakeholders'. Recursion is an important aspect of all operations in this challenge - which clearly scales with the number of players. Therefore, the only solution to this problem is limiting to prevent an adversarial coalition from abusing the cost of sorting.

One of the external methods which previously initialized the fixed size queue for quicksort was re-adapted inside the main selection function. As highlighted in Section 5 the best practice for a quicksort is to select a random starting value. In the revised code (Appendix A.2), this is based on the previous block's hash, which replaces the leftmost element - to minimize worst case computational costs. Before state reset, a 5% reimbursement is made to the caller of the function. As before, the remaining funds are transferred to the winner.

Action	Condition	Transaction Cost	Execution Cost
Deployment	Contract Mined	806794 gas	586362 gas
Exchange	All	73771 gas	52499 gas
Selection	< 5 Players	22104 gas	832 gas
Selection	5 Players, Sorted List	69210 gas	117147 gas
Selection	5 Players, Unsorted List	70643 gas	120014 gas
Selection	5 Players, 0.1 Ether	72574 gas	123876 gas

Table 2: Revised Costs

With the limited party size, the costs in Table 2 are fixed $\pm$ a negligible amount. Greater costs are incurred for deployment due to pre-initialization, but it ensures the exchange is fair to all players. We can see that the scenario in which the players all trade an equal number of wei is still the most expensive, but conversely to Table 1, sorting the pre-sorted list is cheaper than the unsorted list due to the random pivot.

The true deployment cost and exchange costs were measured to be 806794 gas and 73771 gas, respectfully. Which are equal to the values measured in Table 2. The address of the revised contract is:

`0x5f1762db587efa4f1e26b37776bcff01e0513152`

6.2 Other

Two considerations were made in Section 5 which pertained to the security of the contract. The first reasoned about additional rounds of interaction to ensure the secrecy of all bets. A player should initially send a hashed commit of their bet (under a certain cap) along with the full amount and a nonce. This should work in much a similar way as to what has already been implemented. When all commits have been taken the winner should be computed secretly, and a new function should then enable players to refund the different between their bet and the max cap:

```
function redeem(uint bet) {
        if (sha256(bet,nonce[msg.sender])==commitments[msg.sender]) {
                commitments[msg.sender] = 0x00;
                uint256 refund = 1000000000000000000 - bet;
                weight += bet;
                msg.sender.transfer(refund);
                party[msg.sender] = msg.value;
        }
}
```

When all refunds have been issued, or after a certain amount of time has elapsed, another interaction should trigger the transferral of the remaining funds to the winner's address. The computational cost for this can be incentivized as before.

The second task of increasing the random value's entropy can be implemented in a number of ways, including simply adding the numerical value of the nonce to the random value. Another method would be to combine each hash with the head seed using some operation, for example the exclusive or:

```
bytes32 seed = block.blockhash(block.number-1);
for (uint i = 0; i < counter; i++) {
        seed ^= player[i].nonce;
}
uint random = (uint(seed)%weight + 1);
```

Both tasks would dramatically increase computational costs, but the treatment of fairness against security is one to take under heavy consideration.

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[2] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[3] Charles AR Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[4] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[5] Aaron Kumavis and Dan Finlay. Metamask version 3.12.0, 2017.

[6] Peter Mell, John Kelsey, and James Shook. Cryptocurrency smart contracts for distributed consensus of public randomness. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 410–425. Springer, 2017.

[7] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

A Code

A.1 Original Lottery

```
pragma solidity ^0.4.0;

contract lottery {

struct account {
address user;
uint256 tokens;
}

event Draw (address user, uint256 tokens);

mapping (address => uint256) public party;
mapping (uint => address) private queue;
uint256 counter = 0;
uint256 weight = 0;

uint256 rate = 100; // divides wei for tokens
uint256 threshold = 500000000000000000; // maximum balance for draw

function partition(uint left, uint right,
account[] memory acc) internal returns (int) {
int pivot = int(left);
for (uint i = left+1; i < right+1; i++) {
if (acc[i].tokens<=acc[left].tokens) {
pivot++;
account memory new_i = acc[uint(pivot)];
account memory new_p = acc[i];
acc[i] = new_i;
acc[uint(pivot)] = new_p;
}
}

account memory new_min = acc[uint(pivot)];
account memory new_piv = acc[left];

acc[left] = new_min;
acc[uint(pivot)] = new_piv;

delete new_i;
delete new_p;
delete new_min;
delete new_piv;

return pivot;
}
```

```
function sort(int left, int right, account[] memory acc) internal {
        if (left<right) {
                int pivot = partition(uint(left), uint(right), acc);
                sort(left, pivot-1, acc);
                sort(pivot+1, right, acc);
        }
}

function quicksort() private returns (account[] memory) {
        account[] memory acc = new account[](counter);
        for (uint i = 0; i < counter; i++) {
                acc[i].user = queue[i];
                acc[i].tokens = party[queue[i]];
                party[queue[i]] = 0;
                queue[i] = 0;
        }
        sort(0, int(counter-1), acc);
        return acc;
}

// choose a token holder at random with probability
// proportional to their token balance
function choose_winner() private {
        require(counter>0);

        // hash head of chain
        bytes32 head = block.blockhash(block.number-1);

        // select random number
        uint random = (uint(head)%weight + 1);

        account[] memory acc = quicksort();

        uint winner = 0;
        for (uint j = 0; j < counter; j++) {
                random = random + acc[j].tokens;
                winner = j;
                if (random>weight) {
                        break;
                }
        }

        // log order of draw
        for (uint k = 0; k < counter; k++) {
                Draw (acc[k].user, acc[k].tokens);
        }

        // reset state and transfer funds
        weight = 0;
        counter = 0;
        acc[winner].user.transfer(this.balance);
        delete acc;
}
```

```
// wei for tokens
function exchange() public payable returns (uint256) {
        require(msg.value>0);
        if (party[msg.sender]==0) {
                queue[counter] = msg.sender;
                counter = counter + 1;
        }
        party[msg.sender] = msg.value/rate;
        weight = weight + msg.value;
        if (this.balance>=threshold) {
                choose_winner();
        }
        return party[msg.sender];
}
}
```

A.2 Revised Lottery

```
pragma solidity ^0.4.0;

contract lottery {

struct account {
        address user;
        uint256 tokens;
}

event Draw (address user, uint256 tokens);

mapping (address => uint256) public party;
mapping (uint => address) private queue;

// pre-initialise to prevent surcharge to first trader
uint256 counter = 1;
uint256 weight = 1;

uint256 rate = 100;     // divides wei for tokens
uint256 limit = 5;

function partition(uint left, uint right, account[] memory acc)
                        internal returns (int) {
        int pivot = int(left);
        for (uint i = left+1; i < right+1; i++) {
                if (acc[i].tokens<=acc[left].tokens) {
                        pivot++;
                        account memory new_i = acc[uint(pivot)];
                        account memory new_p = acc[i];
                        acc[i] = new_i;
                        acc[uint(pivot)] = new_p;
                }
        }
        account memory new_min = acc[uint(pivot)];
        account memory new_piv = acc[left];
```

```
        acc[left] = new_min;
        acc[uint(pivot)] = new_piv;
        return pivot;
}

function quicksort(int left, int right, account[] memory acc) internal {
        if (left<right) {
                int pivot = partition(uint(left), uint(right), acc);
                quicksort(left, pivot-1, acc);
                quicksort(pivot+1, right, acc);
        }
}

// choose a token holder at random
// with probability proportional to their token balance
function choose_winner() public {
        require(counter>=limit+1);
        weight -= 1;

        // hash head of chain
        bytes32 head = block.blockhash(block.number-1);
        // select random values
        uint random = (uint(head)%weight + 1);
        uint pivot = (uint(head)%counter + 1);

        account[] memory acc = new account[](counter);      // build account
        for (uint i = 0; i < counter; i++) {                // stack of users
                acc[i].user = queue[i];
                acc[i].tokens = party[queue[i]];
                party[queue[i]] = 0;
                queue[i] = 0;
        }

        account memory new_min = acc[uint(pivot)];      // swap random pivot
        account memory new_piv = acc[0];                // for leftmost value
        acc[0] = new_min;
        acc[uint(pivot)] = new_piv;

        quicksort(0, int(counter-1), acc);

        uint winner = 0;
        for (uint j = 0; j < counter; j++) {
                random += acc[j].tokens;
                winner = j;
                if (random>=weight) {
                        break;
                }
        }

        Draw (acc[winner].user, acc[winner].tokens); // log winner

        msg.sender.transfer((this.balance/100)*5); // reimburse caller
        acc[winner].user.transfer(this.balance); // transfer funds to winner
```

```
        // reset state
        weight = 1;
        counter = 1;
        delete acc;
}

// wei for tokens
function exchange() public payable {
        // prevent division into floating points
        require(msg.value>0);
        // prevent spam
        if (counter>limit+1) {
                msg.sender.transfer(msg.value);
        } else {
                if (party[msg.sender]==0) {
                        queue[counter-1] = msg.sender;
                        counter++;
                }
                party[msg.sender] = msg.value/rate;
                weight = weight + msg.value/rate;
        }
}}
```

B Transaction History

Incomplete entries correspond to transactions of which I was unable to obtain a hash.

B.1 Wallet

Block Number	Gas	Hash
114885	79248	0x7785ca899ac980166caf8a2e5eddd9d483946cdae532aaeb1cc101b182bec5f0
-	-	-
115022	63556	0x23d56f9aa063500b3a8a807972494bdc5e5c36d693fc31b71f0449c55cb1a876
-	-	-
160368	80582	0x347b4fa2ef6cb51f197b5f5d43c5d231a44a62f0b298b8b2576f921c2e7c6639
160861	79116	0xcc2f4ddb8dd936fc31b0dafecb2ea1fb810e5b18507ade82f9db0a0c4b8e9ad4

B.2 Contract

Block Number	Gas	Hash
100878	728168	0xf36f56c9054d088d7dd43f40009a61b48d06379cf42666f9b17b6fd7f633eaea
-	-	-
114865	74051	0x07ef49ddd480236628d73c5033396ceb87c70894c0ed277e3a479baebf7b9e14
114913	74051	0x3607e54fe1eac8b12d4dd5e03b175d5dd364e0a1beb83e496bc56a018689c21b
114988	33287	0x728d2e0fa7e4041717c0d7956255aeed4972f41dd6eba84186e0090a887745cd
115048	158310	0x9fb241daf0e54d9dbca30401ed64fd213c74d6de985063a62aab8f543159a60d