EthernaLotto

Published: November 2022

Abstract

We hereby provide the design specification of **EthernaLotto**, a lottery game based on the Ethereum Virtual Machine (“EVM” in the following).

The game will be managed as a Decentralized Autonomous Organization (“DAO” in the following).

Motivations

Lottery games have ancient origins tracing back to the Chinese Han Dynasty, between years 205 and 187 BC. The core idea of the game is that every player pays a small amount of money and a randomly chosen lucky winner gets the money from all contributions.

Unfortunately, classic operation of such games requires a central running authority and subjects them to serious issues:

- **Censorship** - some countries make some or all lotteries illegal.
- **Locality** - lotteries are usually State-managed and subject to local regulations, making them impossible to extend across borders and turn global.
- **Obscure or unfair prize allocation** - in a typical lottery game a share of the jackpot is assigned to every winning category, but the way these shares are calculated is almost never explained and is often unfair.
- **Outright scams** - classic operation provides little to no guarantees about the fairness of the drawing process, as well as the actual value of the ticket sales. The drawn numbers may be biased to favor specific players, and / or the lottery operator may advertise a jackpot that is significantly lower (but still attractive enough) than the ticket sales. It is certainly suspicious that many lotteries constantly advertise perfectly round-numbered jackpots like $1M.

Case in point, Hot Lotto suffered one of the most famous lottery fraud scandals (not necessarily the worst). Other known frauds include a Serbian lottery announcing the numbers before they were drawn and Chinese lottery operators reportedly stealing money. Other frauds that might have occurred may never be uncovered.

We want to overcome all the above problems by implementing a new lottery game based on a decentralized, trustless, censorship-resistant, and cryptographically verified platform: the EVM.

Game Description

Players will play the game by buying one or more tickets using the native currency of the Ethereum blockchain, Ether (or ETH). 10% of the revenue from ticket sales will be transferred to the DAO running the game and distributed among partners, while the remaining 90% will be used as a jackpot.

When buying a ticket a player will choose 6 or more different numbers between 1 and 90 inclusive, and exactly 6 different numbers in the same range will be drawn randomly by the system every week. Tickets matching 2 or more of the drawn numbers will be given a prize.

Neither the order of the numbers played in a ticket nor that of the 6 drawn numbers matters.

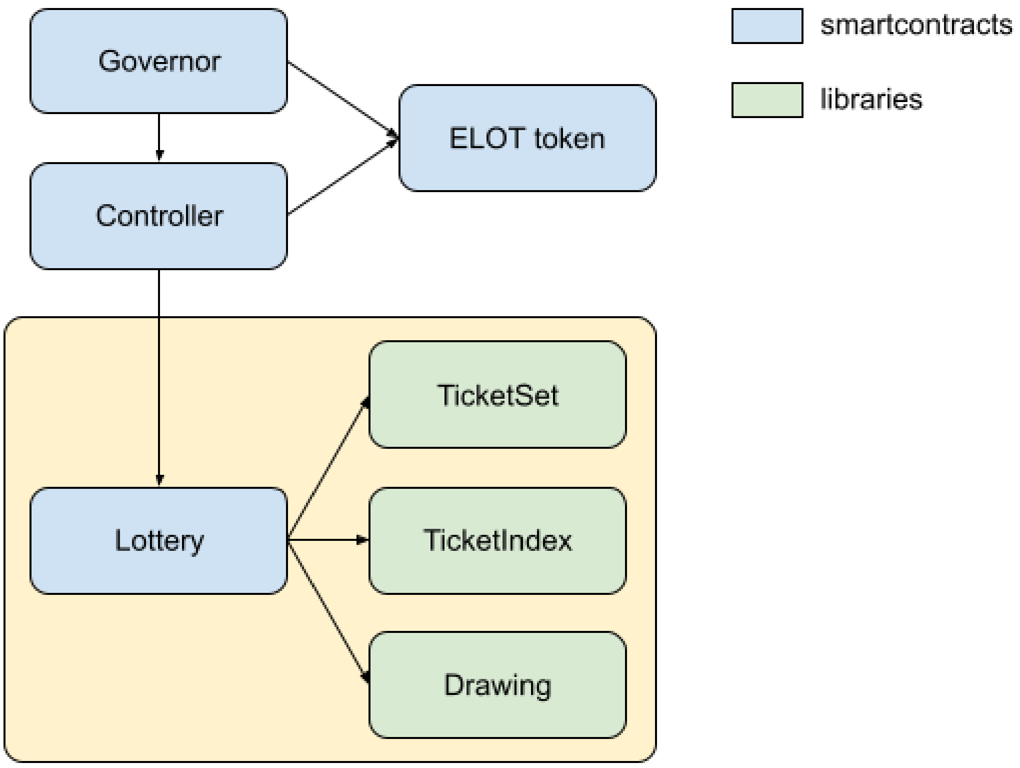
Tickets with more than 6 numbers are treated exactly as if the player had bought many tickets for all possible 6-combinations of those numbers. For example, the price of a 10-number ticket is $\binom{10}{6} = 210$ times the price of a 6-number ticket.[1]

Players are identified by their Ethereum addresses and will be able to withdraw any prizes won by their tickets after each draw. If a player buys more than one ticket, any prizes will add up.

The drawing process will use ChainLink's Verifiable Random Function, or VRF for short, which guarantees fairness and prevents cheating.

Architecture

The following is a high-level view of the components that make up EthernaLotto:



Each block in the diagram represents a separate smartcontract or library. The implementation of all contracts makes extensive use of OpenZeppelin smartcontracts v4 and can be found at https://github.com/ethernalotto/lottery.

The **Lottery** smartcontract implements the core logic of the lottery game. Since the algorithms turned out to be quite complex, some of the functionality has been isolated into the **TicketSet**, **TicketIndex**, and **Drawing** libraries so as to keep the contract size within the 24 KiB limit enforced by the EVM.

The whole **Lottery**+**TicketSet**+**TicketIndex**+**Drawing** group can be thought of as a single black box providing two high-level features: *buying tickets* and *drawing numbers*. The former is public (anyone can buy a ticket), while the latter can be run only by an unspecified *owner* of the contract, as per OpenZeppelin's Ownable contract.

[1] https://en.wikipedia.org/wiki/Binomial_coefficient

The **Controller** contract is an OpenZeppelin `TimelockController` with some additional functionality. Notably, the Controller is the owner of the Lottery and as such it is the recipient of all revenue (i.e. the 10% share of the ticket sales). It provides special methods allowing the DAO partners to withdraw their earnings at any time, as described in DAO and Governance.

The **Governor** contract is a regular OpenZeppelin Governor with specific OpenZeppelin extensions. Voting power is expressed by an ERC20 token we coined for the project, `ELOT`.

The Lottery contract is the only upgradeable one, while all others have been kept non-upgradeable for simplicity. If the DAO decides to perform an upgrade to its own structure it will need to deploy a new Governor+Controller pair and set the new Controller as the owner of the Lottery.

The next two sections describe the algorithms used to index the tickets and identify the winners as efficiently as possible. These algorithms have been optimized in several ways to reduce storage and computational costs.

Storage Structure

The Lottery contract keeps tracks of the sold tickets and their respective buyers.

Tickets are identified by a 64-bit incremental ID starting at 0.

Tickets are stored in a two-dimensional array called `ticketsByNumber`, which indexes them by their numbers:

```
uint64[][90] public ticketsByNumber;
```

The primary index of the array ranges from 0 to 89, matching the range of the playable numbers (offset by 1 because array indices are zero-based). Each element of the 90-element primary array is in turn an array containing the set of tickets that played that number.

For example, the expression `ticketsByNumber[22]` yields the set of all tickets that played the number 23.

Note that a ticket can and will appear more than once in the `ticketsByNumber` data structure: in fact it will appear at least 6 times, in 6 different sets. It cannot appear more than once in the same set because the played numbers must not have repetitions.

The `TicketSet` library is used for `uint64[]` arrays and provides methods to run various algorithms on ticket sets (see Intersections and Other Set Operations). The `TicketIndex` library is used for the whole `ticketsByNumber` data structure and provides a core piece of the drawing algorithm described in Round Closure; it makes use of the lower-level algorithms from `TicketSet`.

Ticket buyers (i.e. players) are stored in another array called `playersByTicket`, which indexes them by ticket:

```
address payable[] public playersByTicket;
```

Since ticket IDs are incremental and start at 0, they can be used as indices in an array. The index of the `playersByTicket` array is the ticket number, while the corresponding element is the address of the player who bought it. An address will show up multiple times if a player bought many tickets in the same round.

As an example, the `playersByTicket[12]` expression yields the address of the player who bought the 13th ticket in the current round.

An important invariant is that the insertion of a ticket in a set always happens by appending it at the end. Since ticket IDs are incremental it follows that all ticket sets are ordered arrays. The algorithms involved in the drawing process actively exploit this fact.

Aside from the two data structures described above, the Lottery contract contains a few other fields to describe the status of the game. At any given time the game can be in one of four possible states:

- OPEN,
- DRAWING,
- DRAWN,
- CLOSING.

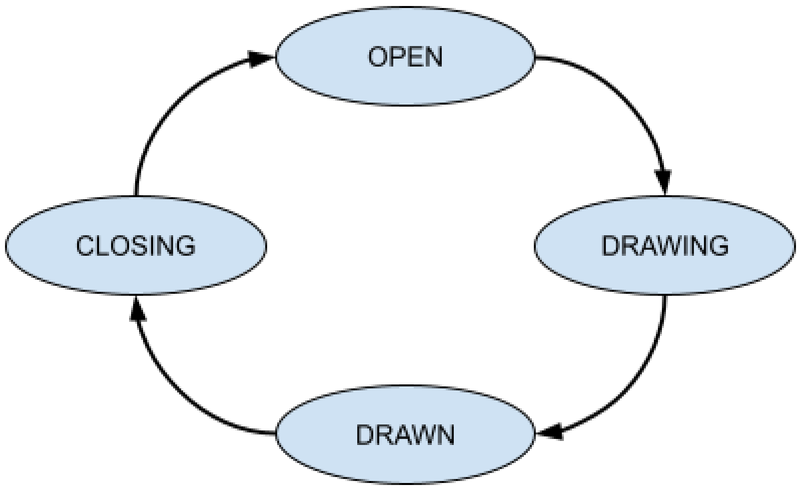
The OPEN state indicates that the game is selling tickets.

The DRAWING state indicates that ticket sales are closed and the ChainLink VRF has been triggered.

The DRAWN state indicates that the ChainLink VRF Coordinator has returned the randomness source and the game is ready to identify winners. See the Drawing Process section for more details.

The CLOSING state indicates that winning tickets have been identified and a separate call can be triggered to attribute prizes to the respective players.

The allowed state transitions are:



In theory the DRAWING, DRAWN, and CLOSING state could be merged into one if all the work for identifying winners and attributing prizes was done in the VRF callback, but in practice these two operations take up a large amount of gas and we prefer to manage them separately.

The Lottery contract allows triggering the drawing process only inside specific windows of time, called **drawing windows**. There is precisely one drawing window every week, lasting 4 hours from Saturday evening at 20:00 UTC until midnight. This gives the user 4 hours of time to trigger the draw of the week, giving some margin to account for possible downtimes of the Ethereum blockchain or other setbacks. At most 1 draw can be triggered in a drawing window, and if one is missed the community will have to wait for the next window.

These timing constraints are managed by storing the following variable in the Lottery contract:

```
uint public nextDrawTime;
```

`nextDrawTime` contains the timestamp of the *beginning* of the next closest drawing window, or the beginning of the current window if one is ongoing. The draw trigger, which transitions the Lottery smartcontract from the OPEN state to the DRAWING state, advances `nextDrawTime` to the beginning of the next window, making it impossible to trigger multiple draws in a week.

The next section describes the whole drawing process in detail.

Drawing Process

Drawing Windows

The `draw` method, responsible for triggering the drawing process, first checks if a drawing window is currently ongoing. It does so by:

- finding the last time a drawing window began,
- checking that the current `block.timestamp` is greater than or equal to that time,
- checking that the current `block.timestamp` is less than that time plus 4 hours.

The following formula is used to find the last drawing window start time:

```
function getCurrentDrawingWindow() public view returns (uint) {
  uint offset = 244800;
  return offset + (block.timestamp - offset) / 7 days * 7 days;
}
```

All timestamps are in seconds. `244800` is the Unix timestamp of the first Saturday evening 20 o'clock time since the Unix Epoch; the above formula uses it to align time to Saturday evenings before flooring it to the previous week boundary (week boundaries are aligned to Saturday evenings thanks to that offset). Flooring is achieved by integer-dividing by 7 days and then remultiplying by 7 days.

The formula used to initialize `nextDrawTime` at deployment is similar to `getCurrentDrawingWindow`, the only difference being that it *ceils* the current time to the next Saturday evening rather than *flooring* it. While flooring unsigned integers in Solidity can be achieved by integer divisions, ceiling is a bit more convoluted. We used the following formula:

```
function _ceil(uint time, uint window) private pure returns (uint) {
  return (time + window - 1) / window * window;
}
```

Having that, the initial value of `nextDrawTime` can be calculated at deployment by:

```
function nextDrawTime() public view returns (uint) {
  uint offset = 244800;
  return offset + _ceil(block.timestamp - offset, 7 days);
}
```

If the drawing window checks are successful, the `draw` method advances `nextDrawTime` by `7 days` so that it is not possible to trigger another drawing in the current window.

Actual Drawing

Once a draw is triggered, the ChainLink VRF is invoked. The VRF callback uses the provided 256-bit randomness to pick 6 different random numbers, which is achieved by initializing an array with the first 90 naturals, scrambling the first 6 slots using the Fisher-Yates shuffle, and returning the first 6 elements.

The drawing algorithm (implemented in the `Drawing` library) follows:

```
function getRandomNumbersWithoutRepetitions(uint256 randomness)
    public pure returns (uint8[6] memory numbers)
{
  uint8[90] memory source;
  for (uint8 i = 1; i <= 90; i++) {
    source[i - 1] = i;
  }
  for (uint i = 0; i < 6; i++) {
    uint j = i + randomness % (90 - i);
    randomness /= 90;
    numbers[i] = source[j];
    source[j] = source[i];
  }
}
```

Once the 6 numbers are known the lottery progresses into the DRAWN state and allows anyone to trigger **round closure**, which consists of the following steps:

1. identifying all tickets that matched two or more of the drawn numbers,
2. attributing the respective prizes so that they are available to the players for withdrawal, and
3. resetting the state of the game.

Steps #1 and #2 are the most complex part of the entire system, and are described in the rest of this section.

Round Closure

The `ticketsByNumber` data structure described in the Storage Structure section allows retrieving the set of tickets that played a specific number efficiently. Having that, we can enumerate all possible ways to choose:

- 6 of the drawn numbers,
- 5 of the drawn numbers,

- 4 of the drawn numbers,
- 3 of the drawn numbers,
- and 2 of the drawn numbers.

This way we are enumerating all possible ways to choose 2 or more numbers from the 6 drawn ones. For every enumerated way we get a combination of numbers, and for every combination we need to:

- look up the tickets that played the 1st number,
- look up the tickets that played the 2nd number,
- …

and **intersect all looked up sets**. The resulting intersection is the list of tickets winning that combination, which may be empty.

We will now describe an algorithm called `findWinningTickets` which takes the `ticketsByNumber` data structure defined in Storage Structure and the 6 drawn numbers as input, and returns a matrix containing the IDs of the winning tickets in each category as output. The output matrix is an array of 5 arrays, one for each winning category, and each secondary array is an ordered list of ticket IDs.

The signature of this algorithm is:

```
function findWinningTickets(
    uint64[][90] storage ticketsByNumber,
    uint8[6] storage numbers
)
    public
    view
    returns (uint64[][5] memory winners)
{
  // …
}
```

Note that the returned tickets are **not** weighted at this stage – the caller is in charge of weighing them as described in Weighting.

The following nested cycle structure is used to scan all possible combinations (aka the "parts") of the drawn numbers:

```
for (uint i0 = 0; i0 < 6; i0++) {
  for (uint i1 = i0 + 1; i1 < 6; i1++) {
    for (uint i2 = i1 + 1; i2 < 6; i2++) {
      for (uint i3 = i2 + 1; i3 < 6; i3++) {
        for (uint i4 = i3 + 1; i4 < 6; i4++) {
          for (uint i5 = i4 + 1; i5 < 6; i5++) {
            // …
          }
        }
      }
    }
  }
}
```

In fact, adding the following calls to the Hardhat console:

```
for (uint i0 = 0; i0 < 6; i0++) {
  for (uint i1 = i0 + 1; i1 < 6; i1++) {
    for (uint i2 = i1 + 1; i2 < 6; i2++) {
      for (uint i3 = i2 + 1; i3 < 6; i3++) {
        for (uint i4 = i3 + 1; i4 < 6; i4++) {
          for (uint i5 = i4 + 1; i5 < 6; i5++) {
            // ...
            console.log(i0, i1, i2, i3, i4, i5);
          }
          console.log(i0, i1, i2, i3, i4);
        }
        console.log(i0, i1, i2, i3);
      }
      console.log(i0, i1, i2);
    }
    console.log(i0, i1);
  }
  console.log(i0);
}
```

yields something like:

```
0 1 2 3 4 5
0 1 2 3 4
0 1 2 3 5
0 1 2 3
0 1 2 4 5
0 1 2 4
0 1 2 5
0 1 2
0 1 3 4 5
0 1 3 4
...
3 5
3
4 5
4
5
```

For a total 63 rows, or 64 if we include the empty set. The combinations of cardinality 0 and 1, namely {}, {0}, {1}, {2}, {3}, {4}, and {5}, are not relevant for the algorithm.

The `ticketsByNumber` data structure provides all the tickets containing a given number. The **TicketSet** library provides an intersection algorithm allowing us to progressively identify the tickets which played a given combination. Combining the two we get:

```
for (uint i0 = 0; i0 < 6; i0++) {
  uint64[] memory tickets0 = ticketsByNumber[numbers[i0]];
  for (uint i1 = i0 + 1; i1 < 6; i1++) {
    uint64[] memory tickets1 = tickets0.intersect(
        ticketsByNumber[numbers[i1]]);
    for (uint i2 = i1 + 1; i2 < 6; i2++) {
      uint64[] memory tickets2 = tickets1.intersect(
          ticketsByNumber[numbers[i2]]);
      for (uint i3 = i2 + 1; i3 < 6; i3++) {
        uint64[] memory tickets3 = tickets2.intersect(
            ticketsByNumber[numbers[i3]]);
        for (uint i4 = i3 + 1; i4 < 6; i4++) {
          uint64[] memory tickets4 = tickets3.intersect(
              ticketsByNumber[numbers[i4]]);
          for (uint i5 = i4 + 1; i5 < 6; i5++) {
            uint64[] memory tickets5 = tickets4.intersect(
                ticketsByNumber[numbers[i5]]);
          }
        }
      }
    }
  }
}
```

`tickets0` contains the tickets matching only 1 number, so it will be discarded. On the other hand, variables `tickets1` through `tickets5` contain actual winners, so they need to be returned in the output.

Let us now initialize and fill in the output data structure:

```
winners = [
  new uint64[](0),  // tickets matching 2 numbers
  new uint64[](0),  // tickets matching 3 numbers
  new uint64[](0),  // tickets matching 4 numbers
  new uint64[](0),  // tickets matching 5 numbers
  new uint64[](0)   // tickets matching 6 numbers
];
for (uint i0 = 0; i0 < 6; i0++) {
  uint64[] memory tickets0 = ticketsByNumber[numbers[i0]];
  for (uint i1 = i0 + 1; i1 < 6; i1++) {
    uint64[] memory tickets1 = tickets0.intersect(
        ticketsByNumber[numbers[i1]]);
    for (uint i2 = i1 + 1; i2 < 6; i2++) {
      uint64[] memory tickets2 = tickets1.intersect(
          ticketsByNumber[numbers[i2]]);
      for (uint i3 = i2 + 1; i3 < 6; i3++) {
        uint64[] memory tickets3 = tickets2.intersect(
            ticketsByNumber[numbers[i3]]);
        for (uint i4 = i3 + 1; i4 < 6; i4++) {
          uint64[] memory tickets4 = tickets3.intersect(
              ticketsByNumber[numbers[i4]]);
          for (uint i5 = i4 + 1; i5 < 6; i5++) {
            uint64[] memory tickets5 = tickets4.intersect(
                ticketsByNumber[numbers[i5]]);
            winners[4] = winners[4].union(tickets5);
          }
          winners[3] = winners[3].union(tickets4);
        }
        winners[2] = winners[2].union(tickets3);
      }
      winners[1] = winners[1].union(tickets2);
    }
    winners[0] = winners[0].union(tickets1);
  }
}
```

The algorithm written so far does not respect the property that a ticket must only be reported in its highest winning category. For example, a 6-ticket with 4 matches will make its way to `tickets3`, but as the algorithm unwinds back to the top it will still be present in `tickets2` and `tickets1`, so it will be incorrectly added to `winners[1]` and `winners[0]`.

There is no easy way to prevent this, because the highest winning category of a given ticket will not necessarily be detected on the first branch of the execution tree where that ticket is processed. Therefore we need to clean up the output data structure at the end with the following addition:

```
for (uint j = 0; j < winners.length - 1; j++) {
  for (uint k = j + 1; k < winners.length; k++) {
    winners[j] = winners[j].subtract(winners[k]);
  }
}
```

Last but not least, we can implement a few optimizations.

First, if at some point during the execution of the 6 stages we end up with an empty ticket set, we can bail out early (see the additions in red):

```
winners = [
  new uint64[](0),  // tickets matching exactly 2 numbers
  new uint64[](0),  // tickets matching exactly 3 numbers
  new uint64[](0),  // tickets matching exactly 4 numbers
  new uint64[](0),  // tickets matching exactly 5 numbers
  new uint64[](0)   // tickets matching exactly 6 numbers
];
for (uint i0 = 0; i0 < 6; i0++) {
  uint64[] memory tickets0 = ticketsByNumber[numbers[i0]];
  if (tickets0.length > 0) {
    for (uint i1 = i0 + 1; i1 < 6; i1++) {
      uint64[] memory tickets1 = tickets0.intersect(
          ticketsByNumber[numbers[i1]]);
      if (tickets1.length > 0) {
        for (uint i2 = i1 + 1; i2 < 6; i2++) {
          uint64[] memory tickets2 = tickets1.intersect(
              ticketsByNumber[numbers[i2]]);
          if (tickets2.length > 0) {
            for (uint i3 = i2 + 1; i3 < 6; i3++) {
              uint64[] memory tickets3 = tickets2.intersect(
                  ticketsByNumber[numbers[i3]]);
              if (tickets3.length > 0) {
                for (uint i4 = i3 + 1; i4 < 6; i4++) {
                  uint64[] memory tickets4 = tickets3.intersect(
                      ticketsByNumber[numbers[i4]]);
                  if (tickets4.length > 0) {
                    for (uint i5 = i4 + 1; i5 < 6; i5++) {
                      uint64[] memory tickets5 = tickets4.intersect(
                          ticketsByNumber[numbers[i5]]);
                      if (tickets5.length > 0) {
                        winners[4] = winners[4].union(tickets5);
                      }
                    }
                    winners[3] = winners[3].union(tickets4);
                  }
                }
                winners[2] = winners[2].union(tickets3);
              }
            }
            winners[1] = winners[1].union(tickets2);
          }
        }
        winners[0] = winners[0].union(tickets1);
      }
    }
  }
}
for (uint j = 0; j < winners.length - 1; j++) {
  for (uint k = j + 1; k < winners.length; k++) {
    winners[j] = winners[j].subtract(winners[k]);
  }
}
```

Secondly, we can increase our chances of emptying a ticket set by subtracting the tickets of the next set early (if they are found in the next set it means they have won in a higher category, so they are not supposed to be in the current set).

```
winners = [
  new uint64[](0),  // tickets matching exactly 2 numbers
  new uint64[](0),  // tickets matching exactly 3 numbers
  new uint64[](0),  // tickets matching exactly 4 numbers
  new uint64[](0),  // tickets matching exactly 5 numbers
  new uint64[](0)   // tickets matching exactly 6 numbers
];
for (uint i0 = 0; i0 < 6; i0++) {
  uint64[] memory tickets0 = ticketsByNumber[numbers[i0]];
  if (tickets0.length > 0) {
    for (uint i1 = i0 + 1; i1 < 6; i1++) {
      uint64[] memory tickets1 = tickets0.intersect(
          ticketsByNumber[numbers[i1]]);
      if (tickets1.length > 0) {
        tickets0 = tickets0.subtract(tickets1);
        for (uint i2 = i1 + 1; i2 < 6; i2++) {
          uint64[] memory tickets2 = tickets1.intersect(
              ticketsByNumber[numbers[i2]]);
          if (tickets2.length > 0) {
            tickets1 = tickets1.subtract(tickets2);
            for (uint i3 = i2 + 1; i3 < 6; i3++) {
              uint64[] memory tickets3 = tickets2.intersect(
                  ticketsByNumber[numbers[i3]]);
              if (tickets3.length > 0) {
                tickets2 = tickets2.subtract(tickets3);
                for (uint i4 = i3 + 1; i4 < 6; i4++) {
                  uint64[] memory tickets4 = tickets3.intersect(
                      ticketsByNumber[numbers[i4]]);
                  if (tickets4.length > 0) {
                    tickets3 = tickets3.subtract(tickets4);
                    for (uint i5 = i4 + 1; i5 < 6; i5++) {
                      uint64[] memory tickets5 = tickets4.intersect(
                          ticketsByNumber[numbers[i5]]);
                      if (tickets5.length > 0) {
                        tickets4 = tickets4.subtract(tickets5);
                        winners[4] = winners[4].union(tickets5);
                      }
                    }
                    winners[3] = winners[3].union(tickets4);
                  }
                }
                winners[2] = winners[2].union(tickets3);
              }
            }
            winners[1] = winners[1].union(tickets2);
          }
        }
        winners[0] = winners[0].union(tickets1);
      }
    }
  }
}
for (uint j = 0; j < winners.length - 1; j++) {
  for (uint k = j + 1; k < winners.length; k++) {
    winners[j] = winners[j].subtract(winners[k]);
  }
}
```

Now the generic stage of the algorithm turns out as follows:

```
for (uint i2 = i1 + 1; i2 < 6; i2++) {
  uint64[] memory tickets2 = tickets1.intersect(
      ticketsByNumber[numbers[i2]]);
  if (tickets2.length > 0) {
    tickets1 = tickets1.subtract(tickets2);

    ...

    winners[1] = winners[1].union(tickets2);
  }
}
```

It is still advantageous to subtract `tickets1` from `tickets0` even though `tickets0` does not participate in the output because doing so increases the likelihood of emptying all sets and stepping directly to the next `i0` index.

The last optimization actually needs to be implemented by the *caller* of `findWinningTickets`. In order to further increase the chance of reducing or emptying a set early, we need to process the drawn numbers in order from the least frequently played to the most frequently played. So the caller needs to sort them by `ticketsByNumber[...].length`. This way we will ensure that `tickets0` is the smallest possible set, `tickets1` is the next smallest, etc. and that will potentially reduce the total processing.

Since we are sorting only 6 elements, implementing efficient $O(N \cdot log(N))$ sorting was an overkill and probably even counterproductive, so we simply implemented the following quadratic time selection sort (in the `Drawing` library):

```
function sortNumbersByTicketCount(
    uint64[][90] storage ticketsByNumber,
    uint8[6] memory number
)
    public
    view
    returns (uint8[6] memory)
{
  for (uint i = 0; i < numbers.length - 1; i++) {
    uint j = i;
    for (uint k = j + 1; k < numbers.length; k++) {
      if (ticketsByNumber[numbers[k]].length <
          ticketsByNumber[numbers[j]].length)
      {
        j = k;
      }
    }
    if (j != i) {
      uint8 t = numbers[i];
      numbers[i] = numbers[j];
      numbers[j] = t;
    }
  }
```

```
    return numbers;
  }
}
```

Intersections

Thanks to the fact that ticket sets are ordered, intersections can often be calculated in sub-linear time by skipping leading ranges. For example, if we are intersecting sets A and B, with `A[0] = 12345` and `B[0] < 12345`, we can skip the leading range of B in logarithmic time until we find an element that is greater than or equal to 12345. In the best case scenario where `A[0] > B[B.length - 1]`, A and B do not have any elements in common and the entire sets will be skipped in logarithmic time.

This is the main intersection algorithm:

```
function intersect(uint64[] memory first, uint64[] storage second)
    public view returns (uint64[] memory result)
{
  uint capacity = second.length < first.length ? second.length : first.length;
  result = new uint64[](capacity);
  uint i = 0;
  uint j = 0;
  uint k = 0;
  while (i < first.length && j < second.length) {
    if (first[i] < second[j]) {
      i = _advanceTo(first, i, second[j]);
    } else if (second[j] < first[i]) {
      j = _advanceToStorage(second, j, first[i]);
    } else {
      result[k++] = first[i];
      i++;
      j++;
    }
  }
  return _shrink(result, k);
}
```

The two `advanceTo` subroutines are used to skip the leading ranges.

Note that the left-hand side input of `intersect` is stored in memory while the right-hand side is in storage, because `findWinningTickets` always intersects in-memory data with data from `ticketsByNumber`.

As a result we need two different `advanceTo` subroutines: one for in-memory data and one for storage data (unfortunately Solidity uses a first-order type system and doesn't allow writing a single generically typed algorithm).

We hereby provide the in-memory version of `advanceTo`; the storage version is identical.

```
function _advanceTo(uint64[] memory array, uint offset, uint64 minValue)
    private pure returns (uint)
```

```
{
  uint i = 1;
  uint j = 2;
  while (offset + j < array.length && array[offset + j] < minValue) {
    i = j + 1;
    j <<= 1;
  }
  while (i < j) {
    uint k = i + ((j - i) >> 1);
    if (offset + k >= array.length || array[offset + k] > minValue) {
      j = k;
    } else if (array[offset + k] < minValue) {
      i = k + 1;
    } else {
      return offset + k;
    }
  }
  return offset + i;
}
```

The `intersect` algorithm uses another subroutine called `_shrink` at the end. That is because we don't know the exact length of the output before calculating it, so we calculate the maximum length it can possibly have by taking the largest between `first.length` and `second.length` (see the `capacity` variable) and allocating an array of that size. However, the caller needs to have an array of the exact length, so we shrink the output to that length at the end.

The `shrink` algorithm is very straightforward:

```
function _shrink(uint64[] memory array, uint count)
    private pure returns (uint64[] memory result)
{
  if (count < array.length) {
    result = new uint64[](count);
    for (uint i = 0; i < result.length; i++) {
      result[i] = array[i];
    }
    delete array;
  } else {
    result = array;
  }
}
```

Other Set Operations

In addition to set intersections, the `findWinningTickets` algorithm described in Round Closure performs unions and subtractions. Both can be implemented relatively easily using the `advanceTo` and `shrink` subroutines we used for intersections.

The `union` algorithm doesn't actually need to use `advanceTo`. It works pretty much like the merge step of a merge sort:

```
function union(uint64[] memory left, uint64[] memory right)
    public pure returns (uint64[] memory result)
{
  if (right.length == 0) {
    return left;
  }
  result = new uint64[](left.length + right.length);
  uint i = 0;
  uint j = 0;
  uint k = 0;
  while (i < left.length && j < right.length) {
    if (left[i] < right[j]) {
      result[k++] = left[i++];
    } else if (left[i] > right[j]) {
      result[k++] = right[j++];
    } else {
      result[k++] = left[i];
      i++;
      j++;
    }
  }
  while (i < left.length) {
    result[k++] = left[i++];
  }
  delete left;
  while (j < right.length) {
    result[k++] = right[j++];
  }
  return _shrink(result, k);
}
```

We still need to call `_shrink` at the end due to the possibility that some tickets show up in both input ranges: in that case `union` must de-duplicate them, resulting in a shorter output than the expected `left.length + right.length`.

`subtract` on the other hand does need to skip leading ranges:

```
function subtract(uint64[] memory left, uint64[] memory right)
    public pure returns (uint64[] memory)
{
  if (right.length == 0) {
    return left;
  }
  uint i = 0;
  uint j = 0;
  uint k = 0;
  while (i < left.length && j < right.length) {
    if (left[i] < right[j]) {
      left[k++] = left[i++];
    } else if (left[i] > right[j]) {
      j = _advanceTo(right, j, left[i]);
    } else {
      i++;
      j++;
    }
  }
  while (i < left.length) {
    left[k++] = left[i++];
  }
  return _shrink(left, k);
}
```

Prize Distribution

Overview

Five winning categories are defined:

1. tickets matching exactly 2 of the drawn numbers,
2. tickets matching exactly 3 of the drawn numbers,
3. tickets matching exactly 4 of the drawn numbers,
4. tickets matching exactly 5 of the drawn numbers,
5. and tickets matching all 6 drawn numbers.

The jackpot is divided in 6 parts, one part for each category and then an extra part to fund the jackpot of the next round in case all categories have at least one winner.

Note that a winning 6-number ticket is awarded a prize only in the highest winning category. For example, if a ticket matches 4 numbers it will not be assigned any prizes from the 3-match and 2-match categories. That is also why a given ticket can win only once: if, for example, a ticket matched two different 3-combinations of the drawn set, and the union of those combinations is a set of 4 numbers, then the ticket is awarded a single prize from the 4-match category, not 2 from the 3-match category. This is true only for 6-number tickets, while for tickets with more numbers the calculation becomes more complex for the reasons explained in the Weighting section.

If a category has no winners in a given round, the corresponding share of the jackpot will be kept for the next round.

Jackpot Subdivision

The 6 shares of the jackpot are hard-coded and have been calculated as follows: 10% is always kept for the next round, while each winning category is assigned an 18% share. The shares for the various categories all have the same size because to award the fairest possible prizes the share for a given category must be inversely proportional to the probability of winning and directly proportional to the expected number of winners. In other words:

- it is fair to allocate a larger share for the winners of a **higher** category because they have a **lower** probability of winning; and
- it is fair to allocate a larger share for the winners of a **lower** category because we expect a **higher** number of winners.

So we have two conflicting criteria that balance each other out, as we show formally in the following.

For a given category k we can call the probability of winning in that category p_k, and the expected number of winners ξ_k. We can call the total number of played 6-combinations in a round n. ξ_k is a binomial variable, so its expected value is np_k. The share of the jackpot for category k can be measured as:

$$S_k = \frac{E(\xi_k)}{p_k} = \frac{np_k}{p_k} = n$$

This measure does not depend on p_k or any other k-dependent parameters, so **all shares must have the same size**.

Weighting

Throughout this section we will call a k-number ticket a k-ticket for simplicity.

As mentioned earlier, EthernaLotto has two properties:

1. 6-tickets must be rewarded only from their highest winning category (e.g. a ticket with 4 matches will **not** also be rewarded for matching 3 or 2 numbers in many ways);
2. tickets with more than 6 numbers must be treated exactly as if the player bought all possible 6-tickets with those numbers, which are $\binom{n}{6}$ for an n-ticket.

If a player buys the following 6-ticket:

1, 2, 3, 4, 5, 6

and the following numbers are drawn:

1, 2, 3, 4, 89, 90

property #1 dictates that the player will receive a prize from the 4-match category but **not** $\binom{4}{3}$ prizes from the 3-match category or $\binom{4}{2}$ from the 2-match one.

The `findWinningTickets` algorithm above returns the winning tickets in their highest category, so that provides all the necessary information to reward 6-tickets.

Higher-order tickets however make things more complex. Let's suppose a player bought the following 8-ticket:

1, 2, 3, 4, 5, 6, 7, 8

and the same numbers as in the example above were drawn, so there were 4 matches.

First off, an 8-ticket corresponds to many 6-tickets, some of which contain the 4 matching numbers; so the prize from the 4-match category must be weighted accordingly. The 6-tickets containing the 4 matching numbers are:

1, 2, 3, 4, 5, 6
1, 2, 3, 4, 5, 7
1, 2, 3, 4, 5, 8
1, 2, 3, 4, 6, 7
1, 2, 3, 4, 6, 8
1, 2, 3, 4, 7, 8

They are 6 in total and can be counted as the number of ways to choose the remaining 2 numbers that can be added to the 4 matching numbers to form a 6-ticket. In formulas:

$$\binom{n-k}{6-k} = \binom{8-4}{6-4} = \binom{4}{2} = 6$$

where n is the cardinality of the ticket (8 in this case) and k is its highest winning category.

Next we observe that not all 6-tickets represented by the 8-ticket have all four matching numbers. The following table provides the exhaustive list of 6-tickets, showing that some of them matched only 3 or 2 of the drawn numbers (highlighted in red):

#	**6-ticket**	**# matches**	**# of tickets in that rank**
1	1, 2, 3, 4, 5, 6	4	$\binom{4}{4} \cdot \binom{8-4}{6-4} = 1 \cdot \binom{4}{2} = 6$
2	1, 2, 3, 4, 5, 7	4	
3	1, 2, 3, 4, 5, 8	4	
4	1, 2, 3, 4, 6, 7	4	
5	1, 2, 3, 4, 6, 8	4	
6	1, 2, 3, 4, 7, 8	4	
7	1, 2, 3, 5, 6, 7	3	$\binom{4}{3} \cdot \binom{8-4}{6-3} = 4 \cdot \binom{4}{3} = 16$
8	1, 2, 3, 5, 6, 8	3	
9	1, 2, 3, 5, 7, 8	3	
10	1, 2, 3, 6, 7, 8	3	
11	1, 2, 4, 5, 6, 7	3	
12	1, 2, 4, 5, 6, 8	3	
13	1, 2, 4, 5, 7, 8	3	
14	1, 2, 4, 6, 7, 8	3	
15	1, 3, 4, 5, 6, 7	3	
16	1, 3, 4, 5, 6, 8	3	
17	1, 3, 4, 5, 7, 8	3	
18	1, 3, 4, 6, 7, 8	3	

19	2, 3, 4, 5, 6, 7	3	
20	2, 3, 4, 5, 6, 8	3	
21	2, 3, 4, 5, 7, 8	3	
22	2, 3, 4, 6, 7, 8	3	
23	1, 2, 5, 6, 7, 8	2	$\binom{4}{2} \cdot \binom{8-4}{6-2} = 6 \cdot \binom{4}{4} = 6$
24	1, 3, 5, 6, 7, 8	2	
25	1, 4, 5, 6, 7, 8	2	
26	2, 3, 5, 6, 7, 8	2	
27	2, 4, 5, 6, 7, 8	2	
28	3, 4, 5, 6, 7, 8	2	

There are $\binom{8}{6} = 28$ 6-tickets in total, of which:

- 6 match 4 numbers,
- 16 match 3 numbers,
- and 6 match 2 numbers.

The rightmost column of the table shows the formula to calculate the number of 6-tickets in each winning category, which can be generalized as:

$$\binom{k}{i} \cdot \binom{n-k}{6-i}$$

with:

- n = cardinality of the ticket,
- k = highest winning category,
- i = winning category.

So the generic n-ticket matching k of the drawn numbers must be awarded $\binom{k}{i} \cdot \binom{n-k}{6-i}$ prizes from the i-match category.

Note that the formula leads to undefined binomial coefficients when $6 - i > n - k$. For example, with $n = 6, k = 5, i = 4$ we have:

$$\binom{k}{i} \cdot \binom{n-k}{6-i} = \binom{5}{4} \cdot \binom{6-5}{6-4} = 5 \cdot \binom{1}{2}$$

but $\binom{1}{2}$ doesn't exist. We therefore adopt the convention that the binomial coefficient is 0 if the bottom side is greater than the upper side, or:

$$\binom{n}{k} = 0 \ \forall k > n$$

This way the formula is well-defined in every possible case and it can actually be generalized to the tickets of any order. In fact, the above sample assignment of n, k, and i describes a 6-ticket matching 5 numbers and receiving 0 prizes from the 4-match category, which is precisely what we want (as per property #1).

We hereby describe a weighting algorithm that applies said formula to the output of `findWinningTickets`.

We will need to calculate many binomial coefficients, for which we implemented the following subroutine (in the `Drawing` library):

```
  function choose(uint n, uint k) public pure returns (uint) {
    if (k > n) {
      return 0;
    } else if (k == 0) {
      return 1;
    } else if (k * 2 > n) {
      return choose(n, n - k);
    } else {
      return n * choose(n - 1, k - 1) / k;
    }
  }
```

which poses $\binom{n}{k} = 0 \ \forall k > n$, as explained above.

The inductive case is based on the following:

$$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}$$

Easily proven as follows:

$$\frac{n!}{k!(n-k)!} = \frac{n}{k} \cdot \frac{(n-1)!}{(k-1)!(n-k)!}$$

The function we use to weigh the winning tickets and calculate the prizes has the following signature:

```
  function _calculatePrizes(uint64[][5] storage winners)
      private view returns (PrizeData[] memory) {
```

where `winners` is the output received from `findWinningTickets` and `PrizeData` is the following struct:

```
struct PrizeData {
  uint64 ticket;
  uint256 prize;
}
```

The weighting algorithm uses an intermediate data structure declared as:

```
    PrizeData[][5] memory prizes;
```

Similarly to the `winners` data structure, the primary array referenced by `prizes` has a slot for each winning category. The secondary arrays are structured as follows:

- `prizes[4]` has as many elements as the 6-match tickets;
- `prizes[3]` has as many elements as the 6-match tickets plus the 5-match ones;
- `prizes[2]` has as many elements as the 6-match, 5-match, and 4-match tickets;
- and so on.

`prizes[0]` has an element for every winning ticket in all categories, and is the returned result. This structure reflects the fact that a k-match ticket is weighed in all categories, starting from k and all the way down to 0.

We initialize `prizes` as follows:

```
uint count = 0;
for (int i = 4; i >= 0; i--) {
  count += winners[uint(i)].length;
  prizes[uint(i)] = new PrizeData[](count);
}
```

Next we apply the formula to all tickets:

```
uint offset = 0;
for (int i = 4; i >= 0; i--) {
  uint ui = uint(i);
  uint maxMatches = ui + 2;
  for (uint j = 0; j < winners[ui].length; j++) {
    TicketData storage ticket = _getTicket(winners[ui][j]);
    for (int k = i; k >= 0; k--) {
      uint matches = uint(k) + 2;
      prizes[ui][offset + j].ticket = winners[ui][j];
      prizes[ui][offset + j].prize += Drawing.choose(maxMatches, matches) *
          Drawing.choose(ticket.cardinality - maxMatches, 6 - matches);
    }
  }
  offset += winners[ui].length;
}
```

The result of the formula is stored in the `prize` field of the corresponding `PrizeData` slot, meaning that at this stage `prize` contains an absolute weight rather than the actual prize amount in wei. We do the conversion in the next step by calculating the *relative* weights and storing a proportional prize amount in the `prize` field:

```
uint256 jackpot = address(this).balance;
for (uint i = 0; i < prizes.length; i++) {
  uint totalWeight = 0;
  for (uint j = 0; j < prizes[i].length; j++) {
    totalWeight += prizes[i][j].prize;
  }
  if (totalWeight > 0) {
    for (uint j = 0; j < prizes[i].length; j++) {
      uint weight = prizes[i][j].prize;
      prizes[i][j].prize = jackpot * weight * 18 / (totalWeight * 100);
    }
  } else {
    for (uint j = 0; j < prizes[i].length; j++) {
      prizes[i][j].prize = 0;
    }
  }
}
```

The `if (totalWeight > 0)` condition is needed to guard against the possibility that a category has no winners except the 6-tickets from higher categories (recall that 6-tickets only have a weight in the highest winning category, their weights in all other categories are 0). If `totalWeight` were 0 the weight-to-prize conversion formula would divide by zero and crash.

As a last step we collapse the rows of the `prizes` data structure into one adding up all prizes for each ticket and return `prizes[0]` as the result, as mentioned above:

```
  for (uint i = prizes.length - 1; i > 0; i--) {
    for (uint j = 0; j < prizes[i].length; j++) {
      prizes[0][j].prize += prizes[i][j].prize;
    }
  }
  return prizes[0];
```

The full algorithm is provided here for clarity:

```
function _calculatePrizes(uint64[][5] storage winners)
    private view returns (PrizeData[] memory)
{
  PrizeData[][5] memory prizes;
  uint count = 0;
  for (int i = 4; i >= 0; i--) {
    count += winners[uint(i)].length;
    prizes[uint(i)] = new PrizeData[](count);
  }
  uint offset = 0;
  for (int i = 4; i >= 0; i--) {
    uint ui = uint(i);
    uint maxMatches = ui + 2;
    for (uint j = 0; j < winners[ui].length; j++) {
      TicketData storage ticket = _getTicket(winners[ui][j]);
      for (int k = i; k >= 0; k--) {
        uint matches = uint(k) + 2;
        prizes[ui][offset + j].ticket = winners[ui][j];
        prizes[ui][offset + j].prize += Drawing.choose(maxMatches, matches) *
            Drawing.choose(ticket.cardinality - maxMatches, 6 - matches);
      }
    }
    offset += winners[ui].length;
  }
  uint256 jackpot = address(this).balance;
  for (uint i = 0; i < prizes.length; i++) {
    uint totalWeight = 0;
    for (uint j = 0; j < prizes[i].length; j++) {
      totalWeight += prizes[i][j].prize;
    }
    if (totalWeight > 0) {
      for (uint j = 0; j < prizes[i].length; j++) {
        uint weight = prizes[i][j].prize;
        prizes[i][j].prize = jackpot * weight * 18 / (totalWeight * 100);
      }
    } else {
      for (uint j = 0; j < prizes[i].length; j++) {
        prizes[i][j].prize = 0;
      }
    }
  }
  for (uint i = prizes.length - 1; i > 0; i--) {
    for (uint j = 0; j < prizes[i].length; j++) {
      prizes[0][j].prize += prizes[i][j].prize;
    }
  }
  return prizes[0];
}
```

DAO and Governance

Overview

As mentioned earlier, the game will be managed as a *Decentralized Autonomous Organization*, or DAO for short.

Participation in the DAO will be determined by EthernaLotto's ERC20 token (trading symbol: `ELOT`). The voting power of each participating address will be proportional to the amount of ELOT owned. This is very standard practice in the Ethereum ecosystem.

Upgrading

As mentioned in the Architecture section, `Lottery` is the only upgradeable contract while all others have been kept non-upgradeable for simplicity. This strategy is sound because the `Lottery` contract holds the jackpot money, so that is the only piece of functionality that we cannot upgrade by simply updating its reference in the frontend(s) and other smartcontracts.

The following table describes the procedures we will enact whenever we want to upgrade one of EthernaLotto's components (as per Architecture):

Component	**Upgrade procedure**
Lottery	Regular Hardhat upgrade.
Libraries	Update the `Lottery` contract and link it against the new libraries while

	redeploying.
Controller	Redeploy it referencing the existing ELOT token and lottery contracts; transfer ownership of the Lottery from the old controller to the new one.
Governance	Same as Controller.
ELOT token	Cannot be upgraded.

Notably, the smartcontract of the ELOT token **cannot be upgraded**. We believe its current implementation to suit anything we may be willing to do for EthernaLotto, but if we ever need to upgrade it we will have to migrate to a "version 2.0" token. The current implementation is based on OpenZeppelin's `ERC20`, `ERC20Burnable`, and `ERC20Votes` for the governance.

Administration

The governance part that is relatively unique to EthernaLotto is the controller: in addition to being an OpenZeppelin `TimelockController` it provides a few administration methods as well as all the necessary methods to trigger drawings. These methods mostly proxy their counterpart in the `Lottery` contract, but change their security model significantly. We wanted to keep the `Lottery` contract as simple as possible due to its upgradeability, so we just made it an `OwnableUpgradeable` whereas the controller uses OpenZeppelin's role-based security.

Method	**Lottery security model**	**Controller security model**
`pause`	`onlyOwner`	`onlyRole(TIMELOCK_ADMIN_ROLE)`
`unpause`	`onlyOwner`	`onlyRole(TIMELOCK_ADMIN_ROLE)`
`canDraw`	(public)	(public)
`draw`	`onlyOwner`	`whenNotPaused` `onlyRoleOrOpenRole(DRAW_ROLE)`
`findWinningTickets`	`onlyOwner`	`whenNotPaused` `onlyRoleOrOpenRole(DRAW_ROLE)`
`closeRound`	`onlyOwner`	`whenNotPaused` `onlyRoleOrOpenRole(DRAW_ROLE)`

Note that the `DRAW_ROLE` is always used with `onlyRoleOrOpenRole` and never with `onlyRole`, meaning that anyone can trigger the drawing process. An earlier iteration of the controller used `onlyRole(DRAW_ROLE)` for `draw`, meaning that only specific privileged users could *initiate* a drawing but anyone could finish it. That ensured the drawings were not initiated at random times, but it was based on trusting the members of `DRAW_ROLE`.

In order to ensure the greatest possible decentralization we later switched to a model where anyone can trigger the drawing process, so the `DRAW_ROLE` is no longer "secure" but we ended up keeping it anyway. Proper draw timing (i.e. once a week) is now enforced by the Lottery contract based on the constraints described in Drawing Windows.

Earnings

Throughout this section we call a *partner* any Ethereum address owning a non-zero amount of ELOT.

Every time a ticket is sold the Lottery contract transfers 10% of the `msg.value` to its owner, therefore the Controller holds all partners' earnings in its balance. Those earnings can be withdrawn by a partner at any time.

A key constraint that has determined the design and implementation of the whole withdrawal system is that **we had to avoid any per-partner operations during the sale of a ticket**, and **avoid all per-partner algorithms in general**. If that were not the case, a malicious partner could disrupt the game or its governance by distributing tiny amounts of ELOT to a very large number of addresses, e.g. 1 ELOT-wei to billions of different addresses, therefore inflating the gas cost of the ticket sales or any other per-partner algorithms.

So the requirements on the withdrawal system are:

- there is a single balance value accounting for the earnings of all partners,
- earnings must be withdrawn by each partner individually rather than being actively distributed by a smartcontract,
- the ELOT balance of a partner can and will change in time, and so will the share of their earnings.

The third requirement makes things especially complicated, but luckily OpenZeppelin has already solved this problem in the context of voting power in the governance. Specifically, the `ERC20Votes` mixin has a checkpointing system that is updated at every transfer and keeps track of the token share of every holder account. The `getPastVotes` and `getPastTotalSupply` methods can be used to determine the ELOT share / voting power of a holder at any point in time.

So we can use ELOT's checkpoints to determine how much of the current earnings balance is due to a given holder, but we need to store several additional pieces of information in the Controller. Our goal is to implement a new Controller method with the following signature:

```
function withdraw(address payable account) public whenNotPaused nonReentrant {
```

It will take care of performing the transfer of any due amount securely. Note that we are not allowing partial withdrawals: calling the method will transfer *all* outstanding due earnings to the recipient, and this assumption simplifies the implementation considerably.

The following pieces of information are stored in the Controller and, along with the ELOT checkpoints, allow `withdraw` to determine how much needs to be transferred:

```
struct Revenue {
  uint256 blockNumber;
  uint256 value;
  uint256 totalValue;
}

uint256 private _totalWithdrawn = 0;
Revenue[] private _revenue;
mapping(address => uint256) public lastWithdrawalBlock;
```

`_totalWithdrawn` contains the sum of all Ether ever withdrawn from the Controller's balance by all partners.

`_revenue` keeps track of the total revenue coming from the Lottery contract, checkpointing it in a way that is similar to how `ERC20Votes` checkpoints the ELOT balances. A new `Revenue` record is added at every draw and it contains the block number, the new revenue collected since the last draw (`value`) and the total revenue ever collected (`totalValue`).

Note that the `totalValue` of the last `Revenue` record minus `_totalWithdrawn` corresponds to the Controller's balance when the ticket sales are closed (i.e. during a draw). If the ticket sales are open the balance will be generally higher due to new earnings that will be accounted for at the next checkpoint (i.e. at the next draw).

`lastWithdrawalBlock` keeps track of the number of the last block that included a `withdraw` transaction for each holder.

Along with proxying the Lottery counterpart, the Controller's `draw` method records a new `Revenue` checkpoint:

```
  function _getLastRoundTotalRevenue() private view returns (uint256) {
    if (_revenue.length > 0) {
      return _revenue[_revenue.length - 1].totalValue;
    } else {
      return 0;
    }
  }

  function draw(uint64 vrfSubscriptionId, bytes32 vrfKeyHash, uint32
callbackGasLimit)
      public whenNotPaused nonReentrant onlyRole(DRAW_ROLE)
  {
    lottery.draw(vrfSubscriptionId, vrfKeyHash, callbackGasLimit);
    uint256 totalValue = address(this).balance - _totalWithdrawn;
    _revenue.push(Revenue({
      blockNumber: block.number,
      value: totalValue - _getLastRoundTotalRevenue(),
      totalValue: totalValue
    }));
  }
```

`withdraw` will make use of a separate `getUnclaimedRevenue` method returning the outstanding revenue amount for a given partner account. Its signature is:

```
  function getUnclaimedRevenue(address account) public view returns (uint256) {
```

`getUnclaimedRevenue` will find the first `Revenue` checkpoint corresponding to the `lastWithdrawalBlock` for the account, iterate over all subsequent checkpoints, and calculate the outstanding revenue by accumulating a share of all their `value`s. The share to accumulate is calculated using ELOT's checkpoint at the `blockNumber` of the `Revenue` checkpoint.

The implementation follows (`token` is a reference to the ELOT token smartcontract implementing `ERC20Votes`):

```
function getUnclaimedRevenue(address account)
    public view returns (uint256 revenue)
{
  revenue = 0;
  for (uint i = _getFirstUnclaimedRound(account); i < _revenue.length; i++) {
    uint256 pastBlock = _revenue[i].blockNumber;
    revenue += _revenue[i].value * token.getPastVotes(account, pastBlock) /
        token.getPastTotalSupply(pastBlock);
  }
}
```

`_getFirstUnclaimedRound` performs a binary search to find the `Revenue` checkpoint following the one corresponding to the `lastWithdrawalBlock`:

```
function _getFirstUnclaimedRound(address account)
    private view returns (uint)
{
  uint256 nextWithdrawalBlock = lastWithdrawalBlock[account] + 1;
  uint i = 0;
  uint j = _revenue.length;
  while (j > i) {
    uint k = i + ((j - i) >> 1);
    if (nextWithdrawalBlock > _revenue[k].blockNumber) {
      i = k + 1;
    } else {
      j = k;
    }
  }
  return i;
}
```

`getUnclaimedRevenue` is public and can be called separately by a user to know how much outstanding revenue is attributed to them.

With all the available pieces of information we can now implement `withdraw`:

```
function withdraw(address payable account) public whenNotPaused nonReentrant {
  require(_revenue.length > 0, 'nothing to withdraw');
  uint256 amount = getUnclaimedRevenue(account);
  require(amount > 0, 'no revenue is available for withdrawal');
  _totalWithdrawn += amount;
  lastWithdrawalBlock[account] = _revenue[_revenue.length - 1].blockNumber;
  account.sendValue(amount);
}
```

The actual money transfer is performed securely thanks to OpenZeppelin's `sendValue` and reentrancy guard. It is also performed *after* any state updates.

Note that `ERC20Votes` implements a delegation system in addition to the checkpointing system. A holder can delegate its voting power to another account, in which case the result of `getPastVotes` no longer corresponds to the balance of each account. This breaks our revenue claiming system, but since we are not interested in exploiting this capability in our governance we simply disabled delegation in ELOT. All ELOT holders implicitly delegate to themselves and all calls to `delegate` or similar methods are reverted. ELOT's implementation contains the following:

```
function delegates(address account) public pure override returns (address) {
  return account;
}

function _delegate(address, address) internal pure override {
  revert('not implemented');
}
```

Alternatives Considered

Choosing the winning addresses randomly

Rather than creating a number-based game and implementing the complex algorithms described in the Drawing Process section we could have simply used ChainLink's randomness to choose one player (or a few ones) randomly.

This is fundamentally different from a traditional number-based game because it does not require any input on the player's side. It also rules out the possibility that many players win in different categories, or that no one wins at all.

Guessing lucky numbers is a much more interesting game, and the various categories only add to the thrill!

Alternative indexing algorithm

The algorithms described in the Drawing Process section take up a lot of gas, and in order for the game to be sustainable the cost of that gas will always have to be funded by (a small percentage of) the ticket sales.

In order to make sure that the bulk of the gas cost is always charged on the players, we initially designed and implemented a completely different data structure to index the tickets, and a completely different drawing process. We hereby describe this alternative model.

Let us define a bijection from the first 90 natural numbers 0, 1, 2, … to the first 90 primes 2, 3, 5, …

We have:

$0 \leftrightarrow 2$
$1 \leftrightarrow 3$
$2 \leftrightarrow 5$
$3 \leftrightarrow 7$
$4 \leftrightarrow 11$
$5 \leftrightarrow 13$
$6 \leftrightarrow 17$
…
$87 \leftrightarrow 457$
$88 \leftrightarrow 461$
$89 \leftrightarrow 463$

As usual, the first 90 naturals correspond to the numbers that can be played in EthernaLotto, offset by 1 so that they are zero-based.

Let us call the above table p so that p_i is the i-th prime.

A ticket playing 6 numbers a_0, a_1, a_2, a_3, a_4, and a_5, **independently of their order**, can be uniquely identified by the following product:

$$h = p_{a_0} \cdot p_{a_1} \cdot p_{a_2} \cdot p_{a_3} \cdot p_{a_4} \cdot p_{a_5}$$

So we can index it in a Solidity `mapping` where a key is the product h and the corresponding value is an array of addresses corresponding to the set of players who played those numbers.

The drawing process then becomes much simpler because we can just calculate the product of the 6 drawn numbers and the list of winners is found by looking it up in the mapping.

This provides the general idea, while the actual algorithm was more complex because it had to account for all possible ways to choose 2 of the played numbers, 3 of them, 4 of them, etc. and had to index the ticket in all corresponding slots of the mapping. A ticket playing 6 numbers would then be stored in:

$$\binom{6}{2} + \binom{6}{3} + \binom{6}{4} + \binom{6}{5} + \binom{6}{6} = 15 + 20 + 15 + 6 + 1$$

… = 57 storage slots.

Note that we did not need to account for $\binom{6}{1}$ and store the ticket in 63 slots because tickets matching only 1 number do not win any prize.

This model was discarded because in our tests it used way more gas than the current one overall, and posed a risk of inflating the ticket prices to unacceptable levels. Through this experience we learned that storage costs are much more critical than computational costs in the EVM.

Sharded drawings

Since the drawing process is gas-heavy, we have considered sharding it as follows.

Enumerating all possible subsets of 6 elements (the 6 drawn numbers) is equivalent to counting from 0 to 63 in binary: it takes exactly 6 bits and the i-th bit can be interpreted as a boolean flag indicating whether the i-th drawn number is included in the current subset. Numbers with less than 2 bits set, namely 0 and the powers of 2, can be discarded.

This way the process of identifying the winning tickets can be divided in $64 - 7 = 57$ smaller parts, allowing for better manageability. For example, if the transaction for the 50th shard fails we would not have to repeat all previous 49 shards.

This approach had to be discarded because overall it uses substantially more gas than the `findWinningTickets` algorithm described in Round Closure. In fact it requires re-calculating all intersections for every shard, whereas `findWinningTickets` reuses partial results and also discards tickets for which a higher win has been detected in a previous iteration.

Managing 6-number tickets only

The formulas and algorithms described in the Weighting section add considerable complexity.

Rather than calculating the weights and proportional prizes of generic k-number tickets, we could have implemented the internal algorithms to manage 6-number tickets only. When a user buys a higher order ticket the lottery could store $\binom{k}{6}$ separate tickets internally. This way the drawing process would not need the weighting part at all, and could use the output of `findWinningTickets` directly.

We discarded this idea because in our tests we found out that the gas usage of the `findWinningTickets` algorithm is critical and increasing the number of tickets based on this idea poses a concrete risk of exceeding Ethereum's gas limits. It can also cause substantial gas cost increases on the buy operations due to the cost of EVM storage, therefore making the game less user-friendly.