Smart Contracts: Coin Flip

In this worksheet we will create a simple betting smart contract. The betting application that we will create involves two parties (the players) which bet on the outcome of a seemingly random event (e.g. a coin flip) and where one of the two parties will win while the other will lose (so ties are not possible).

Getting Started

As we saw in the lectures, smart contracts are typically written in a language called Solidity. Donâ€™t worry if you have never used Solidity in the past, the language has similarities with languages like C++, JavaScript and Python and so it will be easy to pick-up if you have experience with those languages.

Before we begin, here are a few resources that might be useful:

- Solidity documentation: https://docs.soliditylang.org/en/latest/
- Solidity by example: https://solidity-by-example.org/

I recommend that you search the documentation (or just use a search engine) if you are not sure about something (e.g. what a command does). Solidity by example is also a great resource to see code examples, common patterns etc.

Additionally, to save time on set-up, we will be using an online IDE called Remix to code, compile and execute our smart contracts. Here are links to the IDE as well as itâ€™s documentation:

- Remix IDE: https://remix.ethereum.org/
- Remix Docs: https://remix-ide.readthedocs.io/en/latest/

Blockchain coin flip

In order to create a smart contract for a coin flip we will need to change the game a little bit. First, as our bet will take place online; the two players (P1 and P2) donâ€™t need to know each other or agree on a bet beforehand. What might be a better idea to make the protocol more general is for P1 to set a bet amount and then allow anyone online to take his bet if they want to.

The main change is in the way we â€œflipâ€ the coin. In real-life, we just flip a coin. This is difficult to implement as the players are not be in the same room, so itâ€™s difficult for P2 to trust that P1 will flip the coin fairly. Flipping a coin is equivalent to generating a random Boolean value (heads corresponds to true while tails corresponds to false or vice versa) so we could simulate the flip using a random Boolean generator.

Random values are however tricky to implement in smart contracts (or better implement securely â€“ see this answer and the comments for possible solutions https://ethereum.stackexchange.com/a/207). For this reason, we will eliminate this from our coin flip. Instead, we will assign the role of coming up with a Boolean value to P1. P2 then â€œguessesâ€ what P1 has chosen, instead of the outcome of the coin flip.

Therefore, the smart contract coin flip should look something like this:

1. P1 selects one of the outcomes (0/1, heads/tails) and sets the betting amount. P1 commits to this value in a way that is both **binding** (cannot change after this point) and **hiding** (looking at this commitment doesnâ€™t reveal P1â€™s selection).
2. P2 then guesses on the outcome committed by P1.
3. Finally, P1 reveals their selection and if P2 guessed correctly they win, otherwise P1 wins.

To eliminate the possibility that P1 might not reveal their selection, we could also implement an expiration for the bet. In that system, P1 will have some time (letâ€™s say 24 hours) to reveal their selection, otherwise P2 will be able to claim the reward themselves no matter the actual outcome.

Step 1 might seem a bit strange, how can P1 commit to a value such that the commitment is both binding and hiding? This is where **cryptographic hashes** come into action! As we have learned in week 1, cryptographic hashes are deterministic (the same message always produces the same hash) and it is **computationally impossible** for anyone to:

- find a message given a hash (gives us the hiding property)
- find two messages that hash to the same value (gives us the binding property)

Code

Letâ€™s start writing our smart contract code - please create a new file in Remix. If you are stuck in any of the questions, try looking at the solidity documentation, search online or ask a TA.

Note: It might be a good idea to start following the recommended coding style guide (https://docs.soliditylang.org/en/latest/style-guide.html)

Question 1

Letâ€™s start by creating an empty contract called CoinFlip.

Question 2

The first thing we need to do is defined a function that will give us the commitment of P1. Remember we will use a cryptographic hashing function for this. In solidity we have the â€œkeccak256â€ function available so letâ€™s use this. This can be done the following way:

```
keccak256(abi.encode(choice));
```

This will give us a random-looking number which can act as a commitment for P1. We should store this in our contract to make sure P1 doesnâ€™t change its commitment later. This way when P1 reveals his choice we can validate if they are saying the truth or trying to deceive us.

There is one problem, since we only have two values (true and false) P2 could just run this function with both values, and they will be able to tell what P1 chose. To fix this, we need to expand the range of inputs our function takes. The easiest way to do this is to allow P1 to add both their choice and a random number which will be hashed together, and the result will be used as the commitment.

Write a function that accepts a Boolean, and an unsigned integer (preferably of size 256 bits) and returns P1â€™s commitment.

▶ Hint 1:
▶ Hint 2:
▶ Hint 3:
▶ Hint 4:

Question 3

The next thing we need to do is set sone variables which we will use later. As we have two players, we need some way to **store their addresses** (keep in mind that we will need to send money to these addresses). We also need some way to **store the commitment of P1** (the one we will get from the function above) and the **guess of P2**. Finally, we need some way to **store the value and the expiration of the bet** (in case P1 doesnâ€™t reveal). Fill the types and initial values for the following variables (by replacing [type] and [value] to match the correct types and initial values):

```
// player address variables - set player addressees to zero
[type] player1 = [value];
[type] player2 = [value];
// player selection variables
```

```
[type] p1selection;
[type] p2selection;
// bet and expiration variables
[type] public bet;
[type] public expiration = 2**256-1; // set to max
```

Question 4

Now that we have all the â€œscaffoldingâ€ we can start working on the actual functions of our contract. The first thing we need to do is create a function with the following declaration:

```
function makeBet(bytes32 hash) external payable
```

This function takes the commitment of P1 as input and should perform the following operations:

- Check if P1 hasnâ€™t already played
- Then we need to check that some value is given along with this function call (i.e. value is not zero) and set this as the betting amount that P1 is asking
- Finally store P1â€™s commitment in a variable so that we can later check if P1 revealed correctly

▶ Hint:

Question 5

Now letâ€™s create the function that allows P2 to take on P1â€™s bet. It should have the following declaration:

```
function takeBet(bool choice) external payable
```

This function should perform the following:

- First in a similar way as before, check that P1 has already played and that P2 hasnâ€™t
- Then make sure that the value along with this call matches the bet set by P1
- Store P2â€™s choice to check later if itâ€™s the same as P1â€™s choice
- Finally set the expiration to 24 hours from now

▶ Hint:

Question 6

Now that we have functions for bets for both players letâ€™s create the reveal function for P1. This function should:

- Take the same input as the getHash function
- Make sure that the input when hashed matches P1â€™s commitment
- Make sure that P2 played, we donâ€™t want P1 to reveal without P2 first guessing
- Select the winner based on the input and transfer the value stored in the contract to their address

Question 7

We should also give the option to P1 to cancel their bet. This should only be possible if P2 hasnâ€™t yet bet as otherwise P1 might try to cheat by cancelling a losing bet. The function should:

- Only allow P1 to cancel
- Make sure that P2 hasnâ€™t played
- If both conditions are met, return to P1 their bet

Question 8

The last thing we need to do to complete our coin flipping protocol is to provide a way for P2 to win by timeout if P1 doesnâ€™t reveal their selection and the expiration set when P2 played has passed. The function should perform the following actions:

- First check that the expiration has passed
- If the condition is true, then transfer all the value stored in the contract to P2

Motivation

This section provides some motivation for our application. ***This is not necessary to complete the worksheet, so feel free to skip it.***

Real-life coin flips

Letâ€™s start by first looking at a traditional coin flip bet. First the two players (for convenience, letâ€™s name them P1 & P2) are located in the same room. The players first agree on the bet, which could be anything they want. Then one assumes the responsibility of flipping the coin (letâ€™s say P1) and the other (P2) assumes the responsibility of betting on the outcome. As P1 might try to cheat during the flip, the two players could agree that first P1 will flip the coin in the presence of P2 and catch it in a way that **keeps the result hidden** from P2 (e.g. catch it with one hand and cover it with the other) and then P2 will **guess the outcome** (heads or tails). Then P1 will **reveal the outcome** of the flip and the **winner will be determined**.

Letâ€™s break this down into steps so that we can more easily analyze it:

1. The players first agree on the bet.
2. Then P1 flips the coin but keeps the results hidden. At this point, the result is **already determined** and can be **later verified** but the **bet has not yet been â€œsetâ€** by P2. It also doesnâ€™t matter if P1 sees if the coin landed heads or tails (assuming they canâ€™t manipulate P2 or the coin).
3. Then P2 bets on the outcome of the flip. As P1 can hear P2â€™s guess, in a way P2 commits to an outcome.
4. Then P1 reveals the outcome and if P2 guessed correctly they win, otherwise P1 wins.

This might seem perfect, but there are a few problems if you think about it. For example:

- Can both players be certain that the other hasnâ€™t cheated?
- How can any of the players make sure that the loser will honour their end of the deal?
- How can P2 make sure that P1 will reveal the outcome, for example if P1 loses they might just throw the coin so that the outcome cannot be determined.
- Even if the other player keeps their end of the deal, how can you later prove that you won?

In the real world, you might need a witness and/or an enforcer to make sure these things donâ€™t happen. But what if a third party/parties is not available? Even worse, how can you trust that third party? Having someone else, doesnâ€™t eliminate the need for trust, it just shifts it around!

Virtual coin flips

Coin flips in real life seem too difficult to implement securely, so at this point, you might think that it would be a good idea to transform a real-life coin flip into a computer program. You might think that the â€œvirtualâ€ version of a coin flip would be easy to implement securely, after all we have so many secure systems we can rely on. However, this is actually **surprisingly challenging**.

Discussion Question: Take a few minutes to think why this is. Is there anything you can do to make things better? Think of the case where both players are physically in the same location. How can you add the coin flip data to a computer in this case? Can you eliminate trust? How about if you want to have a bet with someone remotely? Can this easily be turned into an online application? What are some of the challenges that you might face?

Food for thought:

- Article on trusting code: https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf
- A video on online voting: https://www.youtube.com/watch?v=LkH2r-sNjQs

Discussion Question: Using a blockchain we can achieve better results as we can eliminate the need to trust a third party.* Why is there an asterisk on the previous statement?

Nice article on trust: https://policyreview.info/glossary/trust-blockchain