University of Basel
Center for Innovative Finance

Smart Contracts and Decentralized Finance

Hashing and Complex Types

Prof. Dr. Fabian Schär
University of Basel

```
Release Ver.:  (Local Release)
Version Hash:  (None)
Version Date:  (None)
```

Sealed Bid Auctions

Sealed bid auction: Bidders submit sealed/secret bids so that no bidder knows the bid of any other participant.

Open auction:

Alice

Valuation: v_a

Bob

Valuation: v_b

Incremental increase if $p \leq v_i$

Sealed bid auction:

Alice

Valuation: v_a

Bob

Valuation: v_b

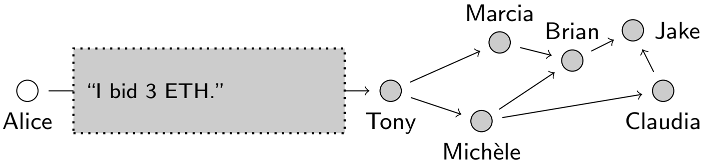
Bid based on own valuation and expectation about other participants' valuations.

Bid and Reveal Phases

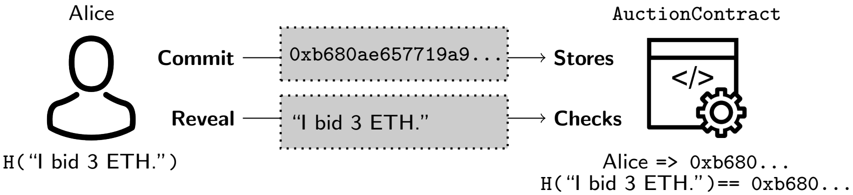
In sealed bid auctions, participants bid quasi-simultaneously, i.e., the auction process consists of two phases. First, bidders submit their bids, and second, the bids are revealed and the highest bidder is determined.

Problem 1: Sealed Bids

Problem: Transaction data on public blockchains/blockchain networks is transparent.



Solution: Commit and reveal
Send a hash of the bid during the bidding phase. Send the unencrypted value during the reveal phase.



Problem 2: Binding Auction

Problem: Ensure that the highest bidder pays if they win.

Separate problems:

- No way to force the bidder to pay later → value must be deposited at the time of the bid.
- Value transfers are visible on-chain / in the network → how can we ensure that bids are still secret?

Solution: Allow for "fake" bids. During the reveal phase, check:

- Deposit $<$ Bid: Bid invalid, full refund
- Deposit $==$ Bid: Bid valid, no refund.
- Deposit $>$ Bid: Bid valid, excess deposit (Deposit - Bid) refunded.

→ Additionally, allow the bidder to secretly state if their bid is fake or legitimate.

Hashing the Bid

Hashing in Solidity: `keccak256(bytes) returns (bytes32)` can be used for any arbitrary input.

Creating a bytes array from any variables:
`abi.encodePacked()` combines all variables in a single bytes array without padding or extending them.

What we will hash:

1. Value of the bid
2. Indicator whether the bid is fake or real
3. Secret (salt) to prevent guessing

Strings

Problem of hashing the bid: Others can brute-force the hash.

Pseudo-code

```
hash_from_alice = 0xb680ae657719a9...

for(x in 1:100) {
    hash = H("I bid x ETH.")
    print(hash == hash_from_alice)
}
```

Solution: add additional arbitrary information to the input.

- In our example, we use a string because it is intuitive and can be used similarly to a password.
- A string stores text and is written with single (’ ’) or double quotes (” ”).
- Unicode strings can be used by prefixing `unicode`, e.g., `unicode"Secret 🌍"`.

Hashing the Bid

Create a pure function to generate a sealed bid:

```
contract SealedBidAuction {
  function generateSealedBid(uint _bidAmount, bool _isLegit, string memory _secret) public pure returns (bytes32 sealedBid) {
    sealedBid = keccak256(abi.encodePacked(_bidAmount, _isLegit, _secret));
  }
}
```

No trace of a pure/view function call is stored on-chain.

Creating the New Contract I

Procedure:

- Keep the basics from the simple auction contract.
- Split the auction into two periods by setting an end time for both periods.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract SealedBidAuction {
  // Auction parameters
  address public immutable beneficiary;
  uint public biddingEnd;
  uint public revealEnd;

  // State of the auction
  uint public highestBid;
  address public highestBidder;
  bool public hasEnded;

```

Creating the New Contract II

```
  // Allowed withdrawals of previous bids
  mapping(address => uint) public pendingReturns;

  event AuctionEnded(address winner, uint amount);

  constructor (address _beneficiary, uint _durationBiddingMinutes, uint _durationRevealMinutes)   {
    beneficiary = _beneficiary;
    biddingEnd = block.timestamp + _durationBiddingMinutes * 1 minutes;
    revealEnd = biddingEnd + _durationRevealMinutes * 1 minutes;
  }

  function withdraw() external returns (uint amount) {
    amount = pendingReturns[msg.sender];
    if (amount > 0) {
      pendingReturns[msg.sender] = 0;
      payable(msg.sender).transfer(amount);
```

Creating the New Contract III

```
    }
  }

  function generateSealedBid(uint _bidAmount, bool _isLegit, string memory _secret) public pure returns (bytes32 sealedBid) {
    sealedBid = keccak256(abi.encodePacked(_bidAmount, _isLegit, _secret));
  }
}
```

Structs

Purpose:

- Keep track of hashed sealed bids with the corresponding deposit amount.
- Complex user defined types with any number of properties.

```
struct Bid {
  bytes32 sealedBid;
  uint deposit;
}
```

Bid can now be used as a variable type, e.g.,

```
Bid newBid = Bid(generateSealedBid(50e18, true, "secret"), 50e18);
```

Structs usage

- Structs can be used in mappings and arrays.
- Structs can contain mappings and arrays.

Arrays

Idea: use mapping to store one bid per address:

```
mapping(address => Bid) bids;
```

Problem: what if users want to create multiple bids?
→ Use a variable sized list of elements that is enumerable:
Dynamic Arrays.

Arrays in Solidity:

- `T[<k>]`: Fixed size array of type `T` and length `k`.
- `T[]`: Dynamic size array of type `T`.

Array properties and methods

Both array types have the `.length()` property. Fixed arrays will return `k`, dynamic arrays the current length.
Dynamic arrays have the `.push(<value>)` and `.pop()` methods to add or remove an element at the end of the array.

Store Bids per Address

Use a mapping to store a dynamic array (a variable size list) for each address:

```
struct Bid {
  bytes32 sealedBid;
  uint deposit;
}
mapping(address => Bid[]) bids;
```

Commit and Reveal Exercise

Exercise 1

Preparation:

- Read the introduction for the ↗ Ethereum Name Service (ENS) `ETHRegistrarController` and the ↗ `makeCommitment` description.
- Check out the `makeCommitment` function in the "Read Code" section of the deployed contract on ↗ etherscan.

Question 1: What is the output of the function if you use

- `name: vitalik`
- `owner: 0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045`
- `secret: 0x6162636400000000000000000000000000000000000000000000000000000000`

as the input values?

Commit and Reveal Exercise

Exercise 1 (part 2)

Question 2: Assume someone created the commitment hash `0x5af80c257639b6180b3d8e91ad2fefa8006afe66bbc98f2e46384e7ccefbe823`. They used the `owner` and `secret` from **Question 1**. Which one of the following names did they want to register?

A. vitalik

B. aaron

C. mary

D. patricia

Question 3: Assume someone created the commitment hash `0xedc18ec53ab6729380c138ef6ab3f04c1a73fec9540cfc09d5ea84ff27ebe796`. They used the `owner` from **Question 1**. You do not know the `secret` they used. Are you able to find out which one of the four names from **Question 2** they wanted to register?

Commit and Reveal Exercise

Exercise 2

1. Create a new contract file named SealedBidAuction.sol
2. Copy the SimpleAuction.sol code to the new file.
3. Delete the `bid()` function and `auctionEnd()` function. Also delete any associated events.
4. Add the `biddingEnd` and `revealEnd` variables and set them as part of the `constructor()`.
5. Add the `generateSealedBid()` function.
6. Deploy the contract and test the `generateSealedBid()` function. Note that we have not yet reimplemented the rest of the auction contract, i.e. the bidding and resolution part.