CS251 Final Exam, Winter 2022
Wednesday, Dec. 14, 2022

Instructions:

- You may use any non-human (and non-AI) resource to answer the questions. You may not collaborate with others. Students are bound by the Stanford honor code.
- To submit your answers please either (i) use the provided LaTeX template, or (ii) print out the exam and write your answers in the provided spaces, or (iii) write your answers on blank sheets of paper, but please make sure to start each question on a new page. When done, please upload your solutions to Gradescope (**DJ66V3**).

- The exam has 6 questions totaling 100 points.
- You have three hours to complete them.
- Please keep your answers concise.

1	/21
2	/16
3	/13
4	/20
5	/20
6	/10
Total	/100

Problem 1. [21 points]: Questions from all over.

A) Suppose Alice wants to pay Bob 2 ETH via an Ethereum transaction. She sends a transaction to the Ethereum network whose value is 2×10^{18} Wei. What prevents a malicious validator from changing the transaction value to 3×10^{18} Wei, thereby making Alice pay 3 ETH to Bob?

B) There are many active EVM blockchains: the Ethereum mainnet, Ethereum testnets (e.g., Goerli), other EVM chains (e.g., Polygon mainnet), several L2 chains and their testnets, and many others. Each chain is identified by a ChainID: Ethereum mainnet is 1, Goerli is 5, Polygon mainnet is 137, etc. Every EVM transaction contains the ChainID of the intended chain. What would go wrong if the ChainID field were not included in the transaction data?
Hint: Recall that on all EVM chains, Alice's address is derived from a hash of her public key, and nothing else.

C) What is the main reason that Ethereum moved from Proof of Work conensus to Proof of Stake consensus?

D) Briefly explain what is a flash loan and give one application.

E) In Solidity, what is the difference between an `external` function and a `public` function? Which one typically costs more gas to call?

F) In class we used collision resistant hash functions to construct commitment schemes. Let $H : \mathcal{X} \to \mathcal{Y}$ be a collision resistant hash function, where $\mathcal{X} = \{0,1\}^n$ and $\mathcal{Y} = \{0,1\}^t$ for some $t < n$. Which of the following three hash functions is not collision resistant (circle all that apply):

A. $H_1(x) := H(x)\|0$ for $x \in \mathcal{X}$

B. $H_2(x\|b) := H(x)$ for $x \in \mathcal{X}$ and $b \in \{0,1\}$

C. $H_3(x\|b) := H(x)\|b$ for $x \in \mathcal{X}$ and $b \in \{0,1\}$

Recall that $\|$ indicates concatenation, for example, $010\|1$ is 0101.

G) The EVM supports both volatile and non-volatile memory. Data is written to volatile memory using the `MSTORE` instruction, and is written to non-volatile memory using the `SSTORE` instruction. The minimum gas cost of one of these instructions is much higher than the other. Which is higher and why is it higher?

Problem 2. [16 points]: Consensus.

Consider n parties, where $n \geq 3$, and where one of the parties is designated as a *sender*. The *sender* has a bit $b \in \{0, 1\}$. A *broadcast protocol* is a protocol where the parties send messages to one another, and eventually every party outputs a bit b_i, for $i = 1, \ldots, n$, or outputs nothing. The parties are connected via an *asynchronous* network, and every pair is connected via an authenticated channel.

- We say that the protocol has **consistency** if for every two honest parties, if one party outputs b' and the other outputs b'', then $b' = b''$.
- We say that the protocol has **validity** if when the *sender* is honest, the output of all honest parties is equal to the *sender*'s input bit b.
- We say that the protocol has **totality** if whenever some honest party outputs a bit, then eventually all honest parties output a bit.

A *reliable broadcast protocol* (RBC) is a broadcast protocol that satisfies all three properties.

A) We are given a broadcast protocol Π among n parties, for some fixed n (say $n = 18$), that is reliable as long as at most f of the parties are corrupt, for some $f < n/3$ (say $f = 5$). Use protocol Π as a black box to construct a reliable broadcast protocol Π' for $m > n$ parties (say $m = 100$) that can tolerate up to f corrupt parties. Please be explicit when describing the messages sent in your protocol Π' outside of Π. Make sure to explain why your Π' has consistency, validity, and totality.
Hint: Protocol Π' only needs to run protocol Π once.

B) Suppose that n is divisible by three and the broadcast protocol Π used in part (A) is reliable even when $f = n/3$ parties are corrupt. Is your protocol Π' from part (A) a reliable broadcast protocol when $f = n/3$ parties are corrupt?

C) Continuing with part (A), suppose the given broadcast protocol Π is reliable for $n > 3$ parties (say $n = 18$) as long as at most $f = n/3$ are corrupt. You may assume n is divisible by three. Use Π as a black box to construct a broadcast protocol Π'' for exactly three parties that is reliable as long as at most one of the three parties is corrupt. Make sure to explain why your protocol Π'' has consistency, validity, and totality.

Problem 3. [13 points]: The value of sandwitch attacks.

Consider two assets X and Y on the Uniswap exchange. The X pool contains x tokens of type X and the Y pool contains y tokens of type Y. Recall that Uniswap v2 ensures that $x \cdot y = k$ for some constant k. Suppose that Uniswap charges no fees.

Alice submits a transaction Tx that sends δx tokens of type X to Uniswap, for some $\delta \geq 0$ (here δx means δ times x). When Tx executes, Uniswap will send back $\delta' y$ tokens of type Y to Alice, where $\delta' = \frac{\delta}{1+\delta}$. This ensures that the constant product is maintained.

Searcher Sam sees Alice's transaction in the mempool and decides to execute a sandwitch attack. Sam issues two transactions Tx_1 and Tx_2 and arranges with the current block proposer that Tx_1 will appear before Alice's transaction in the proposed block and Tx_2 will appear after.

A) Suppose Sam's Tx_1 sends ϵx tokens of type X to Uniswap, for some $\epsilon \geq 0$. Now, when Alice's Tx executes, she will receive back $\delta'' y$ tokens of type Y. What is δ'' as a function of ϵ and δ? Is she getting more or less tokens of type Y than before?

B) Sam's Tx_2, which executes after Alice, will send just enough Y tokens to Uniswap to get back his ϵx tokens of type X that he spent in Tx_1. Thus, his position in X tokens is unchanged after the block executes. How did Sam make a profit from this?

C) Does Sam's profit increase or decrease with the amount he spends in Tx_1? In other words, is a larger ϵ better or worse for Sam (assuming he stays below Alice's max exchange rate)?

D) The validator Victor who is proposing the block sees both Sam's transactions. Victor realizes that he can issue both transactions itself and keep all the profit to itself. However, Victor has no X tokens, as needed for Tx_1. Can Victor mount the sandwitch attack without Sam?

E) Can Alice use MEV-boost to protect herself from these shenanigans by Sam and Victor?

Problem 4. [20 points]: Solidity bugs.

Consider the following (abbreviated) implementation of an NFT marketplace. Anyone can call `createBid()` to create a bid for an NFT and offer payment in an ERC-20 token. The NFT is represented by the address of an ERC-721 contract and a TokenId. Recall that an ERC-721 contract can hold many NFTs, and the tokenId identifies the specific NFT in the collection.

Once the bid is recorded, the NFT owner can call `acceptBid()` to accept payment from the bidder and transfer the NFT to the bidder.

```
contract InsecureMarket {

  // a mapping holding all active bids
  mapping(address => mapping(uint160 => mapping(uint256 => uint256))) bids;

  function createBid(ERC721 erc721Token, uint256 erc721TokenId,
                     ERC20 erc20Token, uint256 price) external
    {
        // pack the two token addresses into a single field
        uint160 key = _getKey(erc20Token, erc721Token);
        // record the bid
        bids[msg.sender][key][erc721TokenId] = price;
    }


  function acceptBid(address bidder, ERC721 erc721Token,
                     uint erc721TokenId, ERC20 erc20Token) external
    {
        uint160 key = _getKey(erc20Token, erc721Token);
        uint256 price = bids[bidder][key][erc721TokenId]; // get bid price
        require(price != 0, "no bid");  // ensure bid exists
        delete bids[bidder][key][erc721TokenId];
        // do the exchange:  ERC721 TokenId for the ERC20 payment
        require(erc20Token.transferFrom(bidder, msg.sender, price));
        require(erc721Token.transferFrom(msg.sender, bidder, erc721TokenId));
    }

  function  _getKey(ERC20 erc20Token, ERC721 erc721Token)
              private pure returns (uint160 key)
    {
        // pack the two addresses into one by computing their XOR
        return uint160(address(erc20Token)) ^ uint160(address(erc721Token));
    }
```

While this marketplace appears to work correctly, it contains several devastating vulnerabilities. Consider the fictitious Zebra collection, an ERC-721 contract.

A) Alice loves TokenId 10 in the Zebra collection and places a bid of 100 USDC for this NFT. That is, Alice posts a transaction that calls

```
createBid(Zebra, 10, USDC, 100)
```

The seller of TokenId 10 is willing to sell the NFT at this price, and posts a transaction that calls

```
acceptBid(alice, Zebra, 10, USDC)
```

Show that Alice can now obtain the NFT by paying only 1 USDC for it (not 100 USDC).

To fix the problem from part (A) the implementation of `acceptBid()` is changed to the following: (all the changes are underlined)

```
function acceptBid(address bidder, ERC721 erc721Token, uint erc721TokenId,
                   ERC20 erc20Token, uint256 price) external {
  uint160 key = _getKey(erc20Token, erc721Token);
  uint256 bidPrice = bids[bidder][key][erc721TokenId]; // get bid price
  require(bidPrice != 0, "no bid");  // ensure bid exists
  require(price <= bidPrice, ‘‘bid too low’’);
  delete bids[bidder][key][erc721TokenId];
  // do the exchange:  ERC721 TokenId for the ERC20 payment
  require(erc20Token.transferFrom(bidder, msg.sender, price));
  require(erc721Token.transferFrom(msg.sender, bidder, erc721TokenId));
}
```

This lets the seller specify the sale price and ensures that the item is not sold below its asking price.

Alice is happy with the modified contract and uses it to buy and sell several NFTs. To do so she calls the `setApprovalForAll` function of the Zebra ERC-721 to indicate that the `InsecureMarket` contract is authorized to sell NFTs on her behalf.

B) There is still another attack on this contract. Suppose Alice recently bought TokenId 53 in the Zebra collection and wants to keep it. She also likes TokenId 34 and places a bid of 80 USDC for it by calling

```
createBid(Zebra, 34, USDC, 80)
```

Show that Bob can now call the `acceptBid()` function and steal Alice's TokenId 53 by paying only 34 USDC for it.
Hint: use the fact that $a \operatorname{xor} b = b \operatorname{xor} a$.

C) Explain how to fix the implementation by changing the `_getKey()` function.

Problem 5. [20 points]: Proof of solvency (simplified).

A proof of solvency lets an exchange prove that it has enough assets to cover all of its obligations to its clients. For simplicity, suppose that the exchange is willing to reveal its on-chain assets (as some exchanges have recently done). This leaves the question of how to provably reveal its total obligation to its clients, but do it privately without revealing every client's balance. We will do so using a zk-SNARK.

To simplify, suppose the exchange only accepts deposits in ETH. Let $bal_i \in \{0, \ldots, 2^{20}\}$ be the balance in ETH of client number i. The exchange builds a Merkle tree where leaf number i is the balance bal_i of client number i, for $i = 1, \ldots, n$. For a technical reason, leaf number 0 is set to a random 256-bit number. Let us assume that the total number of leaves is a power of two. Let R be the resulting Merkle root, and let $B := \sum_{i=1}^{n} bal_i$ be the sum of all the balances.

In addition, the exchange builds a zk-SNARK proof π that B is equal to the sum of all the leaves in the tree. More precisely, the zk-SNARK statement is defined by:

- the public statement is (R, B),
- the secret witness is $\{(bal_i, \Pi_i)\}_{i=1}^{n}$, and
- the arithmetic circuit C outputs 0 if and only if (i) $\sum_{i=1}^{n} bal_i = B$, and (ii) Π_i is a valid Merkle proof for bal_i for all $i = 1, \ldots, n$.

The zk-SNARK proof π proves that the circuit C outputs 0 given the public statement and secret witness as input.

The exchange publishes the triple (R, B, π). Anyone can verify that π is a valid proof, and that B is smaller than the total assets that the exchange owns. In addition, the exchange uses a secure chat to send to client i its balance bal_i and its Merkle proof Π_i. Every client can verify that its balance is correct and that the Merkle proof is correct with respect to the published root R. This convinces the client that its correct balance was included in the calculation of the sum B. If no client complains, then the public can conclude that the exchange is solvent: its assets are greater than its obligations.

There are a number of vulnerabilities in this simplistic construction.

A) Suppose Alice and Bob have the same balance. Then a malicious exchange can use the same leaf for both Alice and Bob. Explain why this would cause the total obligation B to be lower than its true value, and yet no client will complain that the proof is invalid.

B) How would you enhance the proof of solvency to mitigate the attack from part (A)?

C) Suppose client Eve is collaborating with the exchange (and both are malicious). Show that by working together they can cause the total obligation B to be much lower than its true value, say half its true value, and yet no client will complain that the proof is invalid.

D) How would you enhance the proof of solvency to mitigate the attack from part (C)?

Problem 6. [10 points]: Code immutability.

A new contract can be created in the EVM using two opcodes `CREATE` and `CREATE2`.

- When a new contract is created with `CREATE` the address assigned to the new contract is computed as `keccak256(sender, nonce)`, where `sender` is the address of the contract that executed `CREATE`, and `nonce` is the sender's current nonce value. Recall that the nonce of a contract is initialized to zero when it is first created, and is incremented by one whenever the contract creates a new contract.
- When a new contract is created with `CREATE2` the address of the new contract is computed as `keccak256(0xFF, sender, salt, bytecode)`, where `0xFF` is a fixed string, `sender` is as before, `salt` is supplied by the sender, and `bytecode` is the hash of the initialization code of the new contract, as supplied by the sender.

Recall that `keccak256` is a hash function that is believed to be collision resistant. The sender provides both `CREATE` and `CREATE2` with the initialization code and the main body code for the new contract.

A) Suppose `CREATE2` did not exist (i.e., there was no such opcode). Can two different contracts created with `CREATE` collide and end up at the same address?

B) Bob regularly interacts with a contract at address X. He inspected its code and trusts it. Suppose that X was created using `CREATE2` by a contract at address Y. Describe a sequence of events that can cause the code running at address X to be replaced by some new code, without Bob realizing it.
Note that `CREATE2` will fail if it tries to create a new contract at an address already occupied by an existing contract. However, if the existing contract has previously called `SELFDESTRUST` (so that the address is currently vacant) then `CREATE2` will succeed. You may assume that at time T the current contract at address X self destructs.

C) Let's show that your answer from part (B) can also apply to a contract created with `CREATE`, despite your answer from part (A). Suppose a contract at address A uses `CREATE2` to create a contract at address B. The contract at address B uses `CREATE` to create a contract at address C. Describe a sequence of events that could cause the code body of the contract at address C to be replaced by entirely new code. You may assume any self destruct activity that you need.