CS251 Final Exam 2021
Due Wednesday, Dec. 8, 2021, 11:59pm on gradescope

Instructions:
You may use any (non-human) resource to answer the questions. You may not collaborate with others.

Your Name: ______________________________

SUNet ID: ________________`@stanford.edu`

In accordance with both the letter and the spirit of the Stanford Honor Code, I neither received nor provided any assistance on this exam.

Signature: ______________________________

- The exam has 6 questions totaling 100 points.
- You have until the deadline to complete them.
- Please submit your answers on Gradescope (**D5GKRX**)
- Keep your answers concise.

1	/18
2	/20
3	/20
4	/16
5	/15
6	/11
Total	/100

Problem 1. [18 points]: Questions from all over.

A) Briefly explain why a Rollup system stores all transaction data on chain? What would go wrong if transaction data were discarded and not stored anywhere?

B) Consider the following Solidity code:

```
pragma solidity ^0.8.0;
contract ERC20 is IERC20 {
  mapping(address => uint256) private _balances;
  event Transfer(address indexed from, address indexed to, uint256 value);
  function _transfer(address sender, address recipient, uint256 amount) {
    emit Transfer(sender, recipient, amount);
}}
```

Suppose this code is deployed in two contracts: a contract at address X and a contract at address Y. Which of the following can read the state of `_balances` in contract X? Circle the correct answer(s).

A Code in the `_transfer()` function in contract `ERC20` at address X

B Code in the `_transfer()` function in contract `ERC20` at address Y

C An enduser using `etherscan.io`

C) Continuing with part (B), which of the following can read the log entry `Transfer` emitted when the function `_transfer()` is called? Circle the correct answer(s).

A Code in the function `getBalance()` defined in contract `ERC20` at address X

B Code in the function `getBalance()` defined in contract `ERC20` at address Y

C An enduser using `etherscan.io`

D) Two Ethereum transactions, tx_1 and tx_2, are submitted to the network at the same time. Transaction tx_1 has `maxPriorityFee` set to y and transaction tx_2 has `maxPriorityFee` set to $2y$. Will tx_2 necessarily be executed on chain before tx_1? Justify your answer. You can assume that `maxFee` for both tx_1 and tx_2 is greater than `baseFee` $+$ `maxPriorityFee`.

E) Alice wants to buy a car from dealer Bob. She sends 1 BTC to Bob's Bitcoin address. Bob waits for a transaction where (i) the input is from Alice's address, and (ii) one of the outputs is a UTXO bound to Bob's address with a value of 1 BTC. As soon as Bob sees this transaction on the Bitcoin blockchain, he gives Alice the keys, and she drives away. Is this safe? Could Alice get the car for free? If so, explain why. If not, explain what Bob should do to ensure that he gets paid.

F) Alice owns a brand new Tesla Model Y. Can she currently use her car as collateral for a loan in the Compound system? (without selling the car) If yes, explain how. If no, explain why not.

Problem 2. [20 points]: Byzantine broadcast.

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.

- 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.

Let us assume that there is a public key infrastructure (PKI), meaning that every party has a secret signing key, and every party knows the correct public signature verification key for every other party.

In a synchronous network, consider the following broadcast protocol:

- *step 0:* The *sender* sends its input bit b (along with its signature) to all other parties. The *sender* then outputs its bit b and terminates.
- *step 1:* Every non-sender party i echoes what it heard from the *sender* to all the other non-sender parties (with i's signature added). If the party heard nothing from the *sender*, it does nothing in this step. Similarly, the party does nothing in this step if the *sender*'s message is malformed: for example, if the *sender*'s signature is invalid, or the message is not a single bit.
- *step 2:* Every non-sender party collects all the messages it received (up to $n-1$ messages, with at most one from the *sender* in step 0 and at most one from each non-sender party in step 1). If some two of the received messages contain a valid signature by the *sender*, but for opposite bits (i.e., in one signed message the bit is 0 and in the other signed message the bit is 1) then the *sender* is dishonest and the party outputs 0 and terminates. Otherwise, all the properly signed bits from the *sender* are the same, and the party outputs that bit. If the non-sender received no messages, it outputs nothing.

For each of the following questions, describe an attack or explain why there is no attack.

A) If there is at most one dishonest party, does the protocol have consistency?

B) If there is at most one dishonest party, does the protocol have validity?

C) If there are at most two dishonest parties, show that the protocol does not have consistency.

D) If there are at most two dishonest parties, does the protocol have validity?

E) Does the protocol have totality (for any number of dishonest parties)?

Problem 3. [20 points]: An automated market maker (AMM).

Suppose that 1 ETH is worth 1000 DAI. You act as a liquidity provider for Uniswap V2 and contribute 5 ETH and 5000 DAI to the DAI/ETH pool. The combined value of your contributed assets is 10K USD, assuming 1 DAI is worth 1 U.S. dollar.

A) A few months later the price of 1 ETH goes up to 2000 DAI. After the DAI/ETH pool stabilizes to accommodate this new exchange rate, you decide to withdraw your entire position as a liquidity provider. How much ETH and how much DAI will you receive, assuming no fees are charged in the system ($\phi = 1$)?

B) If you had held on to your 5 ETH and 5000 DAI yourself, your assets would now have a value of 15K DAI, a profit of 5K DAI over the starting point. What is the loss that you experienced compared to the "hold yourself" strategy, as a result of being a liquidity provider for Uniswap V2 for these few months? Express the loss as an absolute number of U.S. dollars, assuming 1 DAI = 1 USD. This is called *impermanent loss*, although in this case the loss is quite permanent.

C) If you lost x USD as a result of acting as a liquidity provider for Uniswap V2, where x was calculated in part (b), where did those funds go? Specifically, who gained x USD in this process?

D) Let us now turn to using the Uniswap V2 exchange. Suppose Bob performs a large exchange selling DAI for ETH using the DAI/ETH pool. When the transaction is completed, the amount of DAI in the DAI/ETH pool is a bit higher than before, and the amount of ETH is a bit lower. As a result, the ratio of assets in the DAI/ETH pool is a bit off its equilibrium point.

Alice the arbitrager spots this opportunity and wants to issue an exchange in the opposite direction that will re-balance the pool. She stands to profit from this transaction, and wants to make sure that her transaction is executed immediately after Bob's transaction. This strategy is called *back-running*.

How can Alice implement a back-running strategy? Suggest an approach that has a reasonable chance of making Alice's transaction execute immediately after Bob's.

E) Suppose ten different arbitragers execute the same back-running strategy at the same time in order to capture the arbitrage opportunity created by Bob's transaction. They all use the exact same mechanism you described in part (D). Which one of the ten will win?

Problem 4. [16 points]: The Hashmasks re-entrency bug.

In Lecture 8 and in Section 3 we discussed solidity re-entrency bugs. In this question we will look at an interesting real world example. Consider the following solidity code snippet used in a drop of 16384 NFTs. A user can claim up to twenty NFTs at a time by calling the `mintNFT()` function on this NFT contract. You may assume that all the internal variables are initialized properly by the constructor (not shown).

```
function mintNFT(uint256 numberOfNfts) public payable {
  require(totalSupply() < 16384, "Sale has already ended");
  require(numberOfNfts > 0, "numberOfNfts cannot be 0");
  require(numberOfNfts <= 20, "You may not buy more than 20 NFTs at once");
  require(totalSupply().add(numberOfNfts) <= 16384, "Exceeds NFT supply");
  require(getNFTPrice().mul(numberOfNfts) == msg.value, "Value sent is not correct");

  for (uint i = 0; i < numberOfNfts; i++) {
    uint mintIndex = totalSupply();    // get number of NFTs issued so far
    _safeMint(msg.sender, mintIndex);  // mint the next one
} }

function _safeMint(address to, uint256 tokenId) internal virtual {
  // Mint one NFT and assign it to address(to).
  require(!_exists(tokenId), "ERC721: token already minted");
  _data = _mint(to, tokenId);   // mint NFT and assign it to address to
  _totalSupply ++;              // increment totalSupply() by one

  if (to.isContract()) {

    // Confirm that NFT was recorded properly by calling
    // the function onERC721Received() at address(to).
    // The arguments to the function are not important here.
    // If onERC721Received is implemented correctly at address(to) then
    //      the function returns _ERC721_RECEIVED if all is well.

    bytes4 memory retval =
      IERC721Receiver(to).onERC721Received(to, address(0), tokenId, _data);

    require(retval == _ERC721_RECEIVED, "NFT Rejected by receiver");
} }
```

Let's show that `_safeMint` is not safe at all (despite its name).

A) Suppose 16370 NFTs were already minted, so that `totalSupply() == 16370`. Explain how a malicious contract can cause more than 16384 to be minted. What is the maximum number of NFTs that the attacker can cause to be minted?
Hint: what happens if `onERC721Received` at the calling address is malicious? Examine the minting loop carefully and think of re-entrency bugs.

B) Write the code for a malicious Solidity contract that implements your attack from part (a), assuming the current value of `totalSupply()` is 16370.

C) What single line of Solidity would you add or change in the code on the previous page to prevent your attack? Note that a single transaction should not mint more than 20 NFTs.

Problem 5. [15 points]: Bitcoin questions.

A) The benefit of the Lightning Network proposal is executing payments without posting a transaction to the Bitcoin network. Can Lightning Network payments eventually replace all Bitcoin transactions completely, making the blockchain unnecessary?

B) Recall that a Bitcoin transaction has a set of input addresses and a set of output addresses. Usually, each input address signs the entire transaction (excluding the signatures) to authorize payment. This signature type is called SIGHASH_ALL.

Suppose that instead, the secret key of each input address is used to sign the entire Txin (the input part of the transaction, excluding the signatures) and nothing else. That is, the Txout (the output part of the transaction) is not signed. (this signature type is called SIGHASH_NONE).

Can a miner steal funds from an input address of a transaction that uses the SIGHASH_NONE method, once the transaction is submitted to the Bitcoin network? If so, explain how; if not, explain why not.

C) How would Bitcoin be affected if someone discovered a way to forge an ECDSA signature on an arbitrary message just given an ECDSA public key? Assume it takes 30 minutes of computation to forge one signature, and this cannot be sped up.

Problem 6. [11 points]: Tornado cash.

In Lecture 14 we looked at the Tornado Cash mixer. Recall that the Tornado cash contract needs to store a large list of nullifiers, one nullifier for each withdrawal from the tree. During withdrawal the contract needs to ensure that the nullifier of the note being withdrawn is not in the set of already withdrawn nullifiers. If so, the contract adds this nullifier to the set. The tornado cash contract implements this as a mapping:

```
mapping(bytes32 => bool) public nullifierHashes;
```

During withdrawal the contract verifies the provided zk-SNARK proof, and if valid the contract does:

```
bytes32 _nullifierHash;    // nullifier of note being withdrawn
require(!nullifierHashes[_nullifierHash], "The note has been spent");
nullifierHashes[_nullifierHash] = true;
```

A) Suppose k successful withdrawal from the tree have been processed. Consider a miner that is verifying Ethereum transactions. As a function of k, how much storage space will this miner need to allocate to storing the mapping `nullifierHashes`? You may assume the Tornado contract requires no other long term storage beyond this `nullifierHashes` mapping.

B) It would be better if we could store the set of withdrawn nullifiers S_k off-chain, say somewhere in the cloud. The Tornado contract would only store a short commitment to the current nullifier set S_k. When calling the withdrawal function, the user would provide all the current arguments to that function, and in addition, the user would provide

- a proof π that the nullifier nf of the coin being withdrawn is not in the committed set of nullifiers, namely $nf \notin S_k$, and
- enough information to enable the Tornado contract to calculate the commitment to the updated nullifier set $S_{k+1} := S_k \cup \{nf\}$.

The contract would verify the proof π that $nf \notin S_k$, compute the commitment to S_{k+1}, and replace the current commitment to S_k with the updated commitment to S_{k+1}.

There are several data structures that provide these capabilities where the commitment to S_k is a single 32-byte hash value, and the proof π contains only $2\lceil \log_2 k \rceil$ 32-byte hashes. Moreover, this short proof π enables the Tornado contract to calculate the short commitment to S_{k+1}. One example can be obtained by adapting the Merkle Patricia Tree presented in Lecture 7, but we will leave this as a puzzle for another time.

While this approach will greatly reduce the size of the contract's storage array, it is only worth implementing if it will reduce the gas needed to call the withdrawal function. Consider the following gas costs:

- writing to a zero entry in the storage array: 20K gas,
- writing to a non-zero entry in the storage array: 5K gas,
- `calldata` (the string containing the function arguments): 16 gas per byte.

Suppose we only count the gas consumed by the three items above. For what values of k will this change save gas when calling withdrawal over the current implementation? Recall that the proof π comes out to $32 \times 2\lceil \log_2 k \rceil$ bytes that must be supplied as part of `calldata` to the withdrawal function.

C) Recall that Tornado cash provides a compliance tool that enables a user to de-anonymize their coin: the tool generates a document that links the user's deposit to a specific withdrawal. This document may need to be presented to a centralized exchange (like Coinbase) before the exchange can accept the funds.

Suppose n people deposit one coin each into a single Tornado pool, so that the anonymity set for that pool is of size n (say, $n = 1000$). Later, all n withdraw their coins into n fresh Ethereum addresses (one coin goes into each fresh address). An observer cannot tell which fresh Ethereum address corresponds to which of the n people, so that the anonymity set has size n.

However, suppose $n - 1$ of the people use the compliance tool and send the resulting document to Coinbase. What does this mean for the privacy of the last remaining person who wanted to have a private address?