CS251 Final Exam
Due Wednesday, Nov. 18, 2020, at beginning of lecture

Instructions:
You have 6 days to complete the final exam. You may use any (non-human) resource to answer the questions. You may not collaborate with others.

Your Name: _____

SUNet ID: _____ @stanford.edu

Check if you require expedited grading: ☐

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 6 days to complete them.
- Please submit your answers via Gradescope (**92ZNG8**)
- Keep your answers concise.

1	/18
2	/13
3	/17
4	/16
5	/18
6	/18
Total	/100

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

A) Briefly explain what is an Ethereum re-entrancy attack and why it can lead to loss of funds.

B) In class we presented an HD wallet as a way to generate many addresses that are unlinkable on the blockchain. The online wallet maintains a key, denoted in the lecture by (k_2, h), that lets it construct all these addresses and calculate the user's overall balance. Recall that public key number i is defined as $\mathrm{p}k_i = h \cdot g^{\mathrm{HMAC}(k_2,i)}$ where $h = g^{k_1}$. If this key (k_2, h) is stolen, can the funds associated with any of these addresses be stolen?

Circle your answer: **yes** **no**

Brief justification:

C) In a confidential transaction (Lecture 16) every transaction includes a zero knowledge proof that (i) the sum of the inputs is equal to the sum of the outputs, and (ii) all the outputs are positive. What goes wrong if instead the proof were to prove that (i) the sum of the inputs is equal to the sum of the outputs, and (ii) all the *inputs* are positive. Describe an attack on the system.

D) What is the difference in the privacy models provided by Zcash vs. confidential transactions?

E) The Bitcoin blockchain contains many UTXOs that will never be spent, either because of an `OP_RETURN` script, or because the signing key needed to spend the UTXO has been lost. Nevertheless, they must be maintained as part of the active UTXO set. A frequent proposal is to modify Bitcoin so that UTXOs that are more than two years old will be automatically removed from the UTXO set. Is this a good idea? What will happen to transaction fees for a UTXO that is about to expire in one month?

F) All stablecoin systems maintain some collateral so that when the price of the stablecoin drops, the collateral can be used to shrink the supply of coins and bring the price back up. Some projects maintain on-chain collateral (like MakerDao, which uses ether for collateral) while others maintain off-chain collateral (like USDT, which uses fiat currencies such as the US dollar for collateral). What is the primary difference between these systems in terms of the required level of trust by the public? Specifically, which design allows the public to verify that the collateral is properly managed and maintained?

Problem 2. [13 points]: Consensus protocols.

A) Suppose Alice buys a digital product (e.g., a digital book) from an online merchant and pays using Bitcoin. The merchant sends the product to Alice the minute it sees the payment transaction in a block in the Bitcoin network. Why does this strategy present a significant risk for the merchant?

B) Proof of work. Recall that the mining difficulty in Bitcoin is updated roughly every two weeks so that the expected inter-block time remains at 10 minutes. The difficulty right after the genesis block is quite low, and it increases as more miners join the network. Suppose the blockchain currently contains n blocks. An attacker creates a blockchain of length $n + 100$, starting at the correct genesis block, but where all blocks have the same low difficulty as the genesis block. Because the difficulty is low in all blocks, creating such a long chain takes relatively little work. If miners follow the "mine the longeset chain" rule they would abandon the correct chain and mine this fake longer chain instead. Why is this not an attack on Bitcoin consensus?

Problem 3. [**17 points**]: Solidity bugs.

The following buggy Solidity contract implements a fund-raise for an independent film project by an unknown director. Anyone can send ether to the contract and receive tokens for their contribution, as implemented in the fallback function below. Once the film is finished, royalties from the film will be distributed to all participants based on token ownership. The film producer will fund the production of the film by withdrawing ether from the contract using the `withdraw` function below. The contract should ensure that every month the producer can only withdraw 1 more ether than has ever been withdrawn.

```
contract MovieToken {
  address owner;                           // record the producer's address
  mapping (address => uint256) balances;  // investor balances
  uint256 totalSupply;                     // total supply in contract
  uint256 lastWithdrawDate;                // last withdrawal time

  function MovieToken() {                  // Constructor
    owner = msg.sender;
    balances[owner] = msg.value;           // The producer may fund the movie
    totalSupply     = balances[owner];
  }

  function withdraw(uint256 amount) {     // note: function is not payable

    if (msg.sender != owner) { throw; }   // Only the producer may withdraw funds
    if (amount == 0 || amount > this.balance) { throw; }
    if (now < lastWithdrawDate + (1 months))  { throw; } // Only once per month
    lastWithdrawDate = now;

    // Withdraw schedule: only withdraw 1 more ether than has ever
    // been withdrawn.
    uint256 maxAmount = (totalSupply - this.balance + (1 ether));
    if (amount > maxAmount) { throw; }

    if (!owner.send(amount)) { throw; }  // send the funds to the producer
  }

  // Fallback function: record a fund-raise contribution
  function () payable {
    balances[msg.sender] += msg.value;   // transfer tokens to investor
    totalSupply += msg.value;            // record amount
  }
}
```

A) What is the maximum amount that can be withdrawn in each of the first four months, assuming the maximum amount is withdrawn every month?

month 1: ____ , month 2: ____ , month 3: ____ , month 4: ____ .

B) Describe an attack that lets the producer withdraw all the funds sent to the project immediately on the first month.
Hint: if someone sends funds to a contract address *before* the contract is created, then upon creation of a contract at that address, the variable `this.balance` is equal to the amount of funds sent prior to creation. Note also that all variables in the `MovieToken` contract are *unsigned* integers, so that -1 is actually $2^{256} - 1$.

C) When a contract, say contract X, executes a `selfdestruct` with contract Y's address as an argument, this kills contract X and adds X's balance to Y, without triggering any code execution at Y. With this in mind, suppose we try to prevent your attack from part (B) by changing the second line in the constructor to:

```
balances[owner] = this.balance + msg.value;
```

Is there an attack that lets the producer withdraw all the funds on the first month?

D) How should the contract be fixed so that it correctly implements the intended withdrawal schedule, without changing the program logic?
Hint: add at most four lines to the contract.

Problem 4. [16 points]: Scaling.

Recall that a Rollup server can be used to scale payment transactions in the Ethereum network. The basic SNARK-Rollup system provides no privacy for the payer or payee because all transactions are recorded in the clear on the blockchain.

A) Suppose Rollup were to use a zkSNARK and not record transactions on the blockchain. Would this provide the same level of privacy as Zcash? Justify your answer.
Hint: recall that in (simplified) Zcash, no one other than the payer and payee learn the transaction details.

B) Recall that optimistic Rollup is a variant of Rollup that eliminates the need for a SNARK. The Rollup server simply posts an updated Merkle root to the blockchain, without a SNARK. The update is signed by the Rollup server. Clearly this design is insecure: the Rollup server could issue a fraudulent transaction to steal a user's funds. How does an optimistic Rollup system protect against this?

C) Scaling by sharding. Consider a permissioned blockchain with a fixed set of n validator nodes. One proposal for scaling the blockchain is to split it into n independent blockchains, assigning one validator to each independent blockchain. Now, every decentralized application (DAPP) randomly selects a blockchain to run on, thereby increasing throughput by a factor of n. What are two problems with this design, in terms of security and interoperability between DAPPs?

Problem 5. [18 points]: Front running.

In Lecture 7 we saw an example Ethereum contract for the NameCoin decentralized application (DAPP). The DAPP manages domain names, and lets anyone register a domain name of their choice, by paying the DAPP.

A) Explain why the `NameNew` function (Lecture 7, slide 24) is vulnerable to front running. In particular, explain how miners can profit from front running.

B) Let's redesign `NameNew` to prevent front running. One option is to use cryptographic commitments:

`commit(name,r) = sha3(name,r)` where `r` is a random 512-bit value.

The plan is to split `NameNew` into two functions: one called `NameNewBegin` and one called `NameNewDo`. To register a new domain, the customer first calls the function `NameNewBegin` giving it a commitment to the domain name being requested (the commitment is computed by the caller). Later, the customer calls `NameNewDo` giving it an opening to the commitment (i.e., `name` and `r`). This function registers the new domain in the customer's name. Write the code for these two functions. It is up to you to design these functions so that they properly prevent cheating and front running by either miners and customers. If needed, you may change the `nameEntry` structure.
Caution: note that once the user calls `NameNewDo` the front runner can call `NameNewBegin` and `NameNewDo` in quick succession to try and pre-register the domain that the user wants to reserve. Your code should prevent this.

C) Explain why your implementation prevents front running by miners. In particular:

(a) Why is it important that the commitment be hiding? What would go wrong if the committed value could be extracted from the commitment string?

(b) Why is it important that the commitment be binding? What would go wrong if the user could open the commitment string to any value?

Problem 6. [18 points]: Multisig vs. threshold sigs.

Let $\mathbb{G} = \{1, g, g^2, \ldots, g^{q-1}\}$ be a finite cyclic group of prime order q with generator g. Let H be a hash function $H : \mathbb{M} \to \mathbb{G}$. In a signature scheme called BLS the secret key x is a random number in $\{0, 1, \ldots, q-1\}$ and the public key is $pk = g^x \in \mathbb{G}$. A signature on message $m \in \mathbb{M}$ is simply $\sigma = H(m)^x \in \mathbb{G}$.

A) 3-out-of-3 signing. To protect its signing key x, Bob can split up x into three shares and distribute the shares among three parties as follows: choose three random numbers $x_1, x_2, x_3 \in \{0, 1, \ldots, q-1\}$ such that $x = x_1 + x_2 + x_3 \bmod q$. Bob deletes its local copy of x. To sign a message m each party computes $\sigma_i = H(m)^{x_i}$ for $i = 1, 2, 3$ and sends this partial signature to Bob. Explain how Bob obtains the signature on m from these three partial signatures.

B) Let's generalize the approach above to 3-out-of-5 signing with five parties which we will call P_1, P_2, P_3, P_4, P_5. As in Bitcoin's multisig, we want the signature to identify the subset of three parties that signed the transaction.

Bob does the following: he generates $\binom{5}{3} = 10$ public keys $pk_i = g^{x_i}$ for $i = 1, \ldots, 10$, one public key for each subset of three parties. Say, pk_1 corresponds to the subset $\{P_1, P_2, P_3\}$ and say pk_8 corresponds to the subset $\{P_2, P_4, P_5\}$.

Next, for $i = 1, \ldots, 10$ Bob applies 3-out-of-3 sharing as in part (A) to each of the secret keys x_i to get $x_i = x_{i,1} + x_{i,2} + x_{i,3} \bmod q$. Bob gives the share $x_{i,j}$ to party number j (for $j = 1, 2, 3$) in subset number i. For example, for pk_8 party P_2 gets $x_{8,1}$, P_4 gets $x_{8,2}$, and P_5 gets $x_{8,3}$. Each of the five parties now has one share for each subset of size 3 that it belongs to (each party has 6 shares total). When three parties want to sign a message m, they sign using the three shares that correspond to their subset. For example, P_2, P_4, P_5, will sign using their shares of x_8, namely $x_{8,1}, x_{8,2}, x_{8,3}$.

What information needs to be written to the blockchain to use this 3-out-of-5 signing approach as a replacement for Bitcoin's multisig?

Hint: The funding transaction will create a UTXO containing the Merkle root of a Merkle tree that has the 10 public keys $pk_1, \ldots, pk_{10}$ as its leaves. What signature data is needed in the spending transaction to spend the UTXO?

C) Suppose that (i) the Merkle tree is implemented using SH256 (32 byte hashes), (ii) each ECDSA or BLS public key is 32 bytes, and (iii) each ECDSA or BLS signature is 48 bytes. Does the method used in part (B) result in more or less data written to the blockchain compared to Bitcoin's multisig method for 3-out-of-5 signing?

D) Does your conclusion from part (C) hold for t-out-of-n signing for all t and n? Or is there a t and n where your conclusion would change?

E) When using the method from Part (B), what is the running time for creating the funding transaction in terms of n and t? Is this practical for, say, $n = 100$ and $t = 50$?