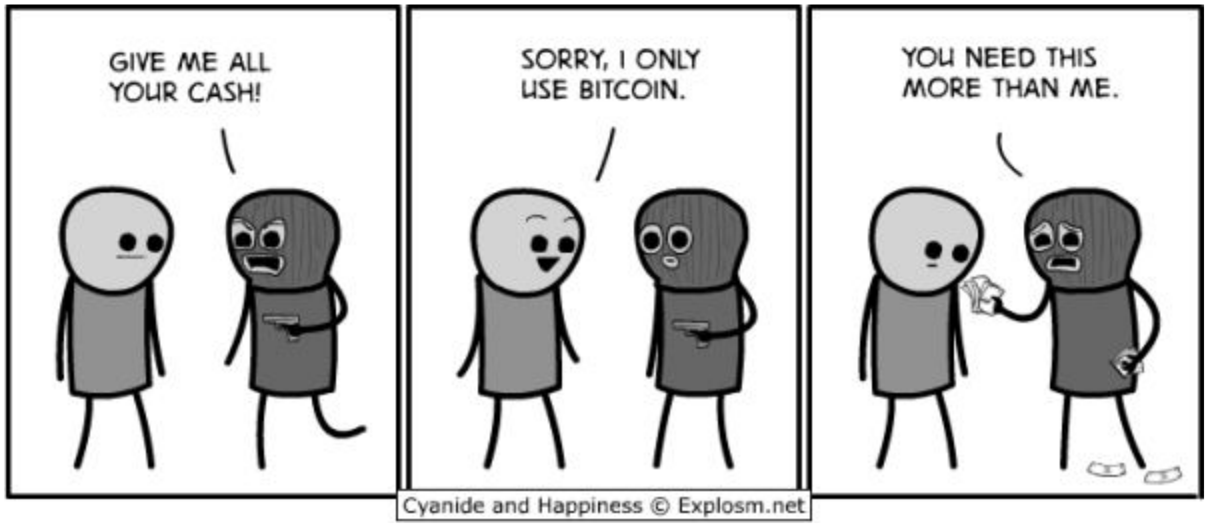
CS 251: Bitcoin and Cryptocurrencies Fall 2016

Final Exam

- The exam is open book and open notes.
- **You have 2.5 hours.**
- Please answer all five questions. All questions are weighted equally.
- The exam is open book and open computer. You may use course notes and documents that you have stored on a laptop, **but you may NOT use the network connection on your laptop in any way, especially not to search the web or communicate with a friend.**
- You are bound by the Stanford honor code not to give or receive unpermitted aid.


GIVE ME ALL YOUR CASH!
SORRY, I ONLY USE BITCOIN.
YOU NEED THIS MORE THAN ME.
Cyanide and Happiness © Explosm.net

<u>Congrats on making it to the end of CS 251!</u>

<u>Question 1</u>: Short-answer questions

A. Explain why it is computationally infeasible for anyone to generate a Bitcoin transaction that references its own output as an input.

B. One early proposal to mitigate temporary block withholding (selfish mining) was to have Bitcoin nodes choose between competing blocks by mining on top of whichever block has the lowest hash value. Describe why this is not a good solution.

C. Consider the following Solidity function contained in a Ethereum contract:

```
function foo(uint256 x, uint256 y) {
    while (uint256(sha3(x)) != y)
        x++;
}
```

Using an x and a y for which this function never terminates, explain how a malicious miner could use this function to waste other miners' resources, without spending any of its own resources.

D. You're consulting for a big-box retail store who want to implement their gift card system using blockchains. You're considering either using colored coins or using an Ethereum contract. Name one advantage each approach has over the other.

E. Suppose that you want to generate each student's draw number for the Stanford housing lottery as H(studentID || B_i) where B_i is the *i*th block header in the Bitcoin blockchain. To make the scheme more secure, your friend suggests using H(studentID || B_{i-1} || B_i) instead. What does this buy you in terms of protecting the scheme against manipulation by miners?

F. Bob posts the following wallet contract to Ethereum to manage his personal finances:

```
contract BobWallet {
    function pay(address dest, uint amount) {
        if (tx.origin == HardcodedBobAddress)
            dest.send(amount);
    }
}
```

Suppose Mallory can trick Bob into calling a method on a contract she controls. Explain how Mallory can transfer all of Bob's money to her own account.

G. When the ZCash genesis block was created, among the extra data the block header included was the hash of a Bitcoin block that was mined that same day. Why did the ZCash creators include this? What were they trying to prove?

H. Recall that Bitcoin uses SHA256^2 as its proof of work function and that the current difficulty is set to approximately $D = 2^{70}$. Suppose Alice discovers a way to solve the Bitcoin proof-of-work problem in expected time $D/2^{20}$, but only for $D < 2^{80}$. For $D \geq 2^{80}$, the best algorithm is still brute-force, taking expected time D. What is the best strategy for Alice to capitalize on this discovery?

Question 2: (Post-block reward mining strategy) Bitcoin's rules call for for fixed coinbase block rewards to disappear over time and be replaced by transaction fees as the sole source of revenue for miners. In this question we'll explore the implications of this transition on miner behavior.

- We'll assume that the block size is increased so that miners can always include all available transactions in a block (e.g. assume blocks are essentially infinite).
- We'll also assume that new transactions are broadcast at a constant rate, all including the same transaction fee. This means that the value of any block is directly proportional to the *time since the last block was found.* We'll say that the block reward is R(t) = rt where t is the time since the last block, and r is a constant.
- We'll assume that all miners have the same cost to compute block hashes (e.g. to pay for the electricity for their mining hardware). We'll say the marginal cost of performing d hashes is c, where d is the current block difficulty (e.g. $d \cong 2^{69}$). We'll also assume that miners can turn their equipment on and off instantaneously and pay nothing when it's off.

Given these assumptions, miners clearly will not want to mine immediately after a block is found, as the value starts at 0, because the transaction pool is empty.

a. How long will miners wait after a block is found before turning on their hardware?
b. Assume that the initial difficulty level d_0 is set to ensure the average block time would be b=10 minutes if all miners were mining constantly. After 2016 blocks using the above strategy, what will the new difficulty d_1 be (in terms of b, c, and r)? Assume the total amount of mining hardware is not changing.
c. Will d_2 change again in the next round (assuming all of the parameters stay the same) or will it stay the same as d_1?
d. Another possible mining strategy is the following: immediately after a block is found, it is not profitable to attempt to find a new block (as the transaction pool is empty) but it is profitable to try to re-mine the previous block, mining ε fewer transactions so that mining on top of this new block will be more attractive than mining on top of the original block. Suppose you are the only miner executing this strategy. Suppose the previous block was found after t_0 time (so that the block reward for the previous is rt_0), and t_1 is the time since the previous block was found. For what value of t_1 should you switch from trying to steal the previous block, to trying to mine a fresh block?

<u>**Question 3**</u>**:** (Centralized blockchains) Consider two designs for a centralized blockchain protocol maintained by N designated parties with signing keys $K_1 \ldots K_N$:

A. *Rotating signers*. Block i must be signed by the jth key K_j where $j=i$ mod N. Each signer has complete control of which valid transactions appear in a block that it signs. Invalid blocks are ignored and forked around, as with the Bitcoin blockchain.
B. *Designated block generator*. Every block is first signed by a designated party called a *block generator* (using a special signing key K_0). The block must then be further confirmed by a quorum t of the N signers using a threshold signature scheme (assume t>1).

Only properly signed blocks are posted on the blockchain. For each of the following properties, briefly compare the two schemes (A) and (B). Where applicable, describe what has to happen (and how many parties need to misbehave) to violate the property.

1. Liveness: what does it take for the system to become unable to post further blocks.
2. Consistency: ensuring that everyone agrees on the current state of the blockchain.
3. The work to generate a new block
4. The work to verify the entire blockchain
5. Censorship resistance

Question 4: (cross-chain exchange) OP_CHECKLOCKTIMEVERIFY is a new Bitcoin instruction (added via soft fork) which examines the top item on the stack and, if it is greater than the current block's timestamp,[1] halts execution with an error. If not, it is a NOP (it is also a NOP for any old client that has not adopted the soft fork). Consider the following Bitcoin script (assume Bob gets to pick *Y*):

```
IF
      <t1>
      CHECKLOCKTIMEVERIFY
      DROP                    //Needed to discard t1 from the stack
      <Alice's public key>
      CHECKSIG
ELSE
      OP_SHA256
      <Y>
      EQUALVERIFY
      <Bob's public key>
      CHECKSIG
ENDIF
```

A. Explain the conditions under which this transaction output can be spent.
B. How could you achieve a similar effect with Bitcoin prior to the availability of OP_CHECKLOCKTIMEVERIFY? What were the downsides of the old way?
C. Show how Alice and Bob can use this transaction to achieve *atomic exchange* of bitcoins for ether. Alice will send her bitcoins to a transaction output with the above script. Bob will send his ether to a special (one-time use) smart contract sketched below. Either Bob should end up with Alice's bitcoins and Alice with Bob's ether, or neither should change hands.

```
contract CrossChain {
    uint256 t2, y;
    address alice, bob;
    void AliceRedeem(uint256 x) {} //implement this
    void BobRedeem() {} //implement this
}
```

Assume the storage variables are suitably initialized, implement the two functions `AliceRedeem` and `BobRedeem`. Neither should be longer than 5 lines.

Hint: you'll want to use block.timestamp to get the timestamp of the current block

D. Assume Bob gets to pick *Y*. How should he choose it?
E. Assume Bob gets to pick *Y*. Should Alice fund her transaction first or should Bob fund the CrossChain contract first?
F. Assume Bob gets to pick *Y*. Should the timestamp t_1 be greater than, less than, or equal to the timestamp t_2? If they are different, how big should the difference be?

[1] It is possible to use OP_CHECKLOCKTIMEVERIFY with either a block number or timestamp, but for here we'll just assume we're using a timestamp.

G. Congrats on implementing atomic exchange! Given that you can do a version of this between any two cryptocurrencies with a powerful enough scripting language, are 2-way pegged sidechains dead? Why or why not?

Question 5: (Ethereum programming) The following Solidity contract implements a distributed ticket sales system. Anybody can create an event (specifying the initial price and number of tickets). Anybody can then purchase one of the initial tickets or sell those tickets peer-to-peer. At the event, gate agents will check that each attendee is listed in the final attendees list on the blockchain.

Surprise! This contract is poorly implemented. Identify at least 5 different problems that cause a specific functionality loss or security vulnerability (not simply things that are generally bad practice). For each problem, state what the implications are (e.g. how it can be exploited or what intended functionality won't work) as well as how it could be fixed.

```
pragma solidity ^0.4.0;
contract TicketDepot {

   struct Event{
        address owner;
        uint64 ticketPrice;
        uint16 ticketsRemaining;
        mapping(uint16 => address) attendees;
   }

   struct Offering{
        address buyer;
        uint64 price;
        uint256 deadline;
   }

   uint16 numEvents;
   address owner;
   uint64 transactionFee;
   mapping(uint16 => Event) events;
   mapping(bytes32 => Offering) offerings;

   function ticketDepot(uint64 _transactionFee){
       transactionFee = _transactionFee;
       owner = tx.origin;
   }

   function createEvent(uint64 _ticketPrice, uint16
_ticketsAvailable) returns (uint16 eventID){
       numEvents++;
       events[numEvents].owner = tx.origin;
       events[numEvents].ticketPrice = _ticketPrice;
       events[numEvents].ticketsRemaining = _ticketsAvailable;
       return numEvents;
   }

   modifier ticketsAvailable(uint16 _eventID){
       _;
       if (events[_eventID].ticketsRemaining <= 0) throw;
   }

   function buyNewTicket(uint16 _eventID, address _attendee)
ticketsAvailable(_eventID) payable returns (uint16 ticketID){
       if (msg.sender == events[_eventID].owner || msg.value >
events[_eventID].ticketPrice + transactionFee){
          ticketID = events[_eventID].ticketsRemaining--;
          events[_eventID].attendees[ticketID] = _attendee;
          events[_eventID].owner.send(msg.value - transactionFee);
          return ticketID;
       }
   }

   function offerTicket(uint16 _eventID, uint16 _ticketID, uint64
_price, address _buyer, uint16 _offerWindow) {
       if (msg.value < transactionFee) throw;
       bytes32 offerID = sha3(_eventID+_ticketID);
       if (offerings[offerID] != 0) throw;
       offerings[offerID].buyer = _buyer;
       offerings[offerID].price = _price;
       offerings[offerID].deadline = block.number + _offerWindow;
   }

   function buyOfferedTicket(uint16 _eventID, uint16 _ticketID,
address _newAttendee) payable{
       bytes32 offerID = sha3(_eventID+_ticketID);
       if (msg.value > offerings[offerID].price &&
          block.number < offerings[offerID].deadline &&
          (msg.sender == offerings[offerID].buyer ||
           offerings[offerID].buyer == 0)) {

          events[_eventID].attendees[_ticketID]
             .send(offerings[offerID].price);
           events[_eventID].attendees[_ticketID] = _newAttendee;
           delete offerings[offerID];
       }
   }
}
```