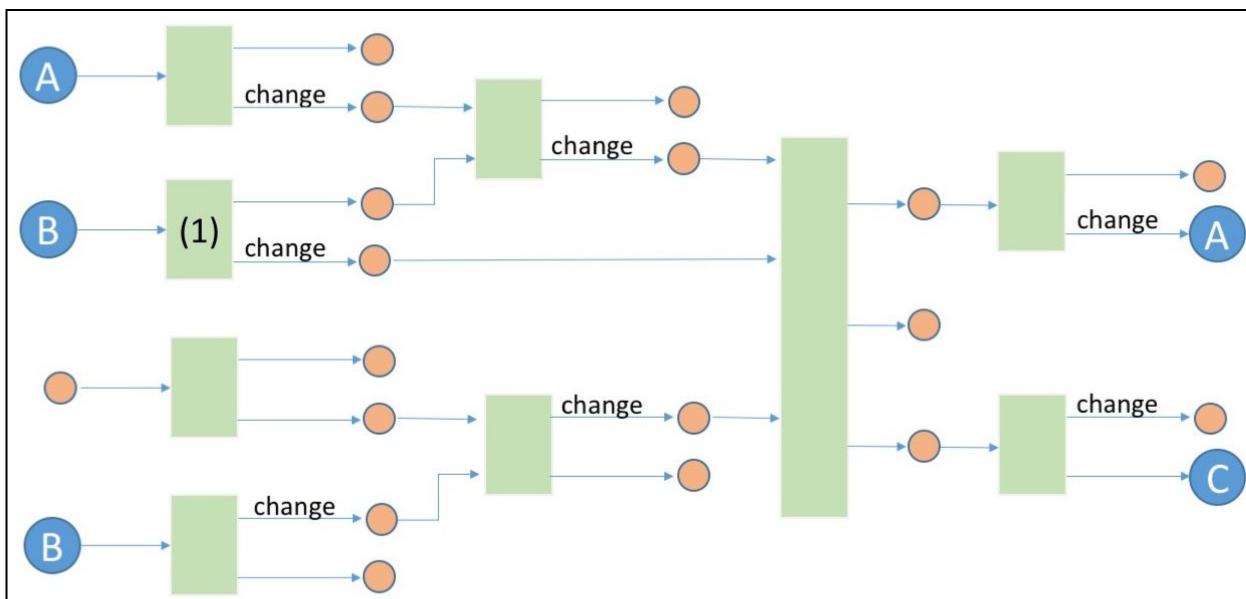
CS 251: Bitcoin and Cryptocurrencies Fall 2016

Homework 3

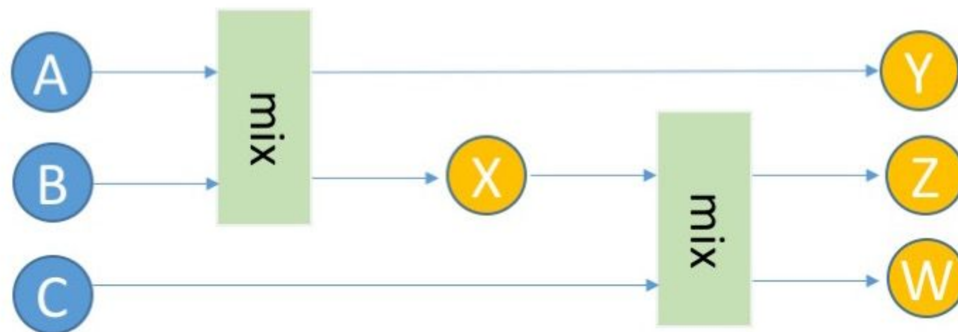
due: 2016-11-28, 23:59 via Gradescope (entry code M4YJ69)

---

1. **Idioms of use:** Consider the transaction graph in the figure below: rectangles represent transactions, empty circles represent fresh addresses, and filled in circles represent addresses controlled by the named entity (i.e., 'A' stands for Alice, 'B' stands for Bob, and 'C' stands for Carol). An edge labeled "change" means that the end node is the change address for that transaction, as identified by the heuristics discussed in class. Note that not every transaction has an identified change address.
    a. Can an observer identify who was paid by Bob in the transaction marked (1)? Explain how or explain why they cannot be identified with certainty.
    b. Can an observer identity who paid Carol? Explain how or explain why she cannot be identified with certainty.



2. **Mix aggregation:** Suppose Bitcoin was changed such that transactions can have at most two inputs and two outputs (this is the case in Zerocash, for example). Alice, Bob, and Carol each have a single UTXO worth 1 BTC. They want to mix their coins among the three of them (without a trusted server) so that three output UTXOs are worth 1 BTC each and they are perfectly shuffled: all six (3!) permutations are equally likely. They decide to do the following: Alice and Bob post a transaction with their two bitcoins as input and two fresh addresses, X and Y, as outputs, each holding 1 BTC. Alice controls one address and Bob controls the other, but an observer cannot tell which is which. Suppose X is listed as the first output in this transaction. Next, Carol and the owner of X post a similar transaction: Carol's coin and the coin X are the inputs, and two fresh addresses W and Z are the outputs, each holding 1 BTC. Carol controls one address and the owner of X controls the other. As before, an observer cannot tell which is which. They use Y,Z,W as the final shuffled coins. Pictorially, the mixing process looks as follows:



   a. What are the anonymity sets for the addresses Y, Z, and W?
   b. Assuming that each mix independently assigns equal probability to both outcomes, what is the probability that Alice controls address W? What is the probability that Carol controls W?
   c. Suppose at a later time it is revealed that Bob controls W. What else does this reveal?
   d. Clearly this mix is inadequate as the six possible permutations are not equally likely. By adding one transaction to the mix, can you help Alice, Bob, and Carol design a better mix so that all six (3!) permutations are equally likely to an outside observer?

   Hint: Adding one more mix transaction is sufficient. The three mixes can function independently, but one or more of the mixes must choose the ordering of outputs non-uniformly, namely with one-third probability one way and two-thirds the other way.

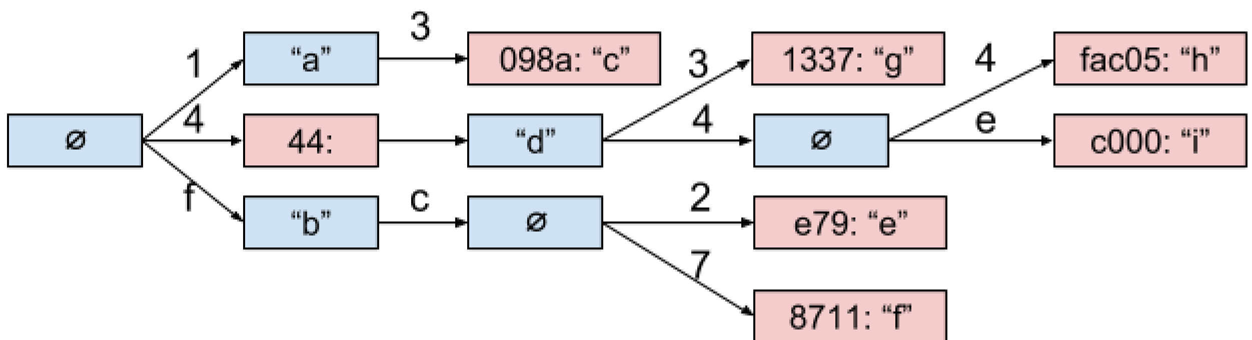
   Discussion: It can be shown that a mix of size two is sufficient to build a mix of size *n*, for any *n*>2, using only O(*n* log *n*) mixes of size 2. In other words, Coinjoin can be made to work even if the number of inputs per transaction is limited to two.

   However, note that if each mix is uniform (assigning an equal probability to both outcomes) and operates independently of the other mixes, then it is impossible to build a perfectly uniform mix for *n* users with any number of mixes. To see why, consider the number of possible outcomes for *m* uniform and independent mixes and compare with *n!*, the number of possible permutations of *n* users.

3. **Ethereum data structures** Ethereum replaces Bitcoin's standard Merkle trees with "Merkle Patricia trees," also known as radix tries, prefix trees, tries, and several other names. Ethereum's specific implementation maps *keys* (or *paths*) of 256 bits to *values* of 256 bits. The tree allows three types of nodes, each represented by a 256-bit identifier:

- empty nodes, represented as all zeroes (elided from the diagram below)
- diverge nodes, which are represented by the hash of a 17-member array. The first 16 items in the array are child node identifiers which contain the hashes of up to 16 child nodes (which will be 0 for empty children). Each child node extends the path of its parent by one nibble (4 bits) of key, defined by its place in the array. This is shown as an edge label in the figure below. The 17th item is a data value (which may be 0) which is mapped to the key representing the path to this node. Internal nodes are represented in blue below, with the child array represented by pointers to child nodes.
- kv nodes, which include an arbitrary-length *path*, plus either the hash of another diverge node or the hash of a data item (making the node a leaf). This path is added to the path built up by this node's place in the tree. These nodes are shown in pink below.

In the following example, 9 keys are present, with 13098a mapped to "c" and 444 to "d":
Explain why in the above example there is no label on the arrow coming out of the intermediate node with the path 44.



a. Which data would you need to supply for a proof that the key "44431337a" has no data mapped to it? How about the keys "fc" and "1a3098a"?
b. Given random keys of arbitrary length, the average-case proof lengths (for both absence and presence) are O(log *n*) for a tree with *n* elements, as desired. However, constants may vary. Describe a worst-case tree with just *k* keys that requires proofs of length O(*k*).
c. Addresses in Ethereum (for contracts and owned accounts) are 160 bits. Explain why this worst-case proof time is not a serious problem for the tree storing account state.
d. Explain why this does not hold for the trees maintaining the long-term storage for each contract (addressable by 256-bit addresses). How might you write a malicious contract which stores *k* words in memory and then makes a single write which is expensive as possible?

4. **Ethereum micropayments.** In class we saw a protocol for serial micropayments in Bitcoin: Alice wants to send a series of payments to Bob, so she puts *n* units of currency into an escrow account which is a multisig account shared by Alice and Bob. She then repeatedly signs transactions, the first paying 1 unit to Bob and returning n-1 to Alice, the second paying 2 units to Bob and returning n-2 units to Alice, and so on. Alice stops sending transactions whenever she wishes, and Bob can sign and publish the last transaction received. A time-locked refund transaction is used to ensure Alice can recover her money if Bob goes offline.

   Let's consider solving this problem in Ethereum. It is very easy to implement the above logic. We can actually add a cool feature: each micropayment need only require hashing, rather than signatures. This is perfect for lightweight clients (although initialization will require a signed message). The scheme works as follows: Alice picks a random *X* and computes $Y = H^n(X)$ [that is, iterate the function H n times starting at X, e,g., for n=2 we have Y = H(H(X))]. She then creates a contract with *n* units of currency and embeds the value *Y* in the contract (and sends *Y* to Bob). To pay Bob *k* units (for $k<n$), she sends Bob the value $Z = H^{n-k}(X)$.

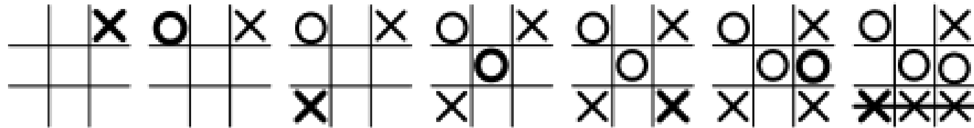
   a. Describe (in pseudocode or prose) two conditions under which the contract should send money to Alice and/or Bob (and then call SUICIDE).
   b. What security property should the hash function satisfy to ensure that Bob cannot steal more money than Alice intended to give him?
   c. For a given *n and k*, how expensive is this protocol: how many hashes does the contract need to to compute before distributing funds? How much data will Alice and Bob need to store?
   d. In practice, we would like to minimize the resources consumed by the contract above all else as gas is expensive. Since the above scheme is a linear chain, you might guess it can be improved using a tree structure. You'd be right! Describe this tree structure. What are the storage and computation costs (in terms of n and *k*) for Alice, Bob, and the contract?

5. **Ethereum mixing** Recall that ethereum does not have multisig transactions nor transactions with multiple inputs/outputs, so Bitcoin-style mixing approaches like CoinJoin do not work directly. Assume that three parties (call them Alice, Bob, and Carol), have established the following: Alice has a random byte-array k_a that only she knows, Bob has a random byte-array k_b that only he knows, and Carol has a random byte-array k_c that only she knows. These byte-arrays are all the same length and satisfy $k_a \oplus k_b \oplus k_c = 0$ (the $\oplus$ operator represents a bitwise exclusive-or in cryptography). You may assume that these byte-arrays are as long as you need them to be. There are several cryptographic protocols that could establish these byte-arrays, including by means of an Ethereum contract, but you do not need to implement that part.

   Explain in pseudocode how to implement a mix contract (analogous to CoinJoin) between Alice, Bob, and Carol using these random byte-arrays. Each input account should be able to specify its desired output account, but the output account should be unlinkable to the input account. Recall that every message/transaction sent to the contract costs gas and therefore the account that originated the message/transaction will be known. Your contract should require only one message each from Alice, Bob, and Carol. You should not assume any other infrastructure beyond the Ethereum contract. Make sure to handle the case where one or more participants never send their funds to the mix contract, in which case the other participants should be refunded (at their original address).

Hint: it might help to familiarize yourself with one-time pad encryption.

6. **Question 5:** (Ethereum programming) The contract code presented below is an attempt to implement a two-player game (with a wager on the line) of Tic-Tac-Toe, also known as Noughts and Crosses:



This implementation contains *at least* 9 serious bugs which compromise the security of the game. Identify 6 of them and briefly describe how they might be fixed. Recall that Ethereum initializes all storage to zero. Assume that the function `checkGameOver()` is correctly implemented and returns true if either player has claimed three squares in a row on the current board.

```
1.  contract TicTacToe {
2.      // game configuration
3.      address[2] _playerAddress;     // address of both players
4.      uint32     _turnLength;        // max time for each turn
5.
6.      // nonce material used to pick the first player
7.      bytes32    _p1Commitment;
8.      uint8      _p2Nonce;
9.
10.     // game state
11.     uint8[9]   _board;             // serialized 3x3 array
12.     uint8      _currentPlayer;     // 0 or 1, indicating whose turn it is
13.     uint256    _turnDeadline;      // deadline for submitting next move
14.
15.     // Create a new game, challenging a named opponent.
16.     // The value passed in is the stake which the opponent must match.
17.     // The challenger commits to its nonce used to determine first mover.
18.     function TicTacToe(address opponent, uint32 turnLength,
19.                                        bytes32 p1Commitment) {
20.         _playerAddress[0] = msg.sender;
21.         _playerAddress[1] = opponent;
22.         _turnLength = turnLength;
23.         _p1Commitment = p1Commitment;
24.     }
25.
26.     // Join a game as the second player.
27.     function joinGame(uint8 p2Nonce) {
28.         // only the specified opponent may join
29.         if (msg.sender != _playerAddress[1])
30.             throw;
31.         // must match player 1's stake.
32.         if (msg.value < this.balance)
33.             throw;
```

```
34.
35.         _p2Nonce = p2Nonce;
36.     }
37.
38.     // Revealing player 1's nonce to choose who goes first.
39.     function startGame(uint8 p1Nonce) {
40.         // must open the original commitment
41.         if (sha3(p1Nonce) != _p1Commitment)
42.             throw;
43.
44.         // XOR both nonces and take the last bit to pick the first player
45.         _currentPlayer = (p1Nonce ^ _p2Nonce) & 0x01;
46.
47.         // start the clock for the next move
48.         _turnDeadline = block.number + _turnLength;
49.     }
50.
51.     // Submit a move
52.     function playMove(uint8 squareToPlay) {
53.         // make sure correct player is submitting a move
54.         if (msg.sender != _playerAddress[_currentPlayer])
55.             throw;
56.
57.         // claim this square for the current player.
58.         _board[squareToPlay] = _currentPlayer;
59.
60.         // If the game is won, send the pot to the winner
61.         if (checkGameOver())
62.             suicide(msg.sender);
63.
64.         // Flip the current player
65.         _currentPlayer ^= 0x1;
66.
67.         // start the clock for the next move
68.         _turnDeadline = block.number + _turnLength;
69.     }
70.
71.     // Default the game if a player takes too long to submit a move
72.     function defaultGame() {
73.         if (block.number > _turnDeadline)
74.             suicide(msg.sender);
75.     }
76. }
```