CS251: Cryptocurrencies and Blockchain Technologies **Fall 2018**

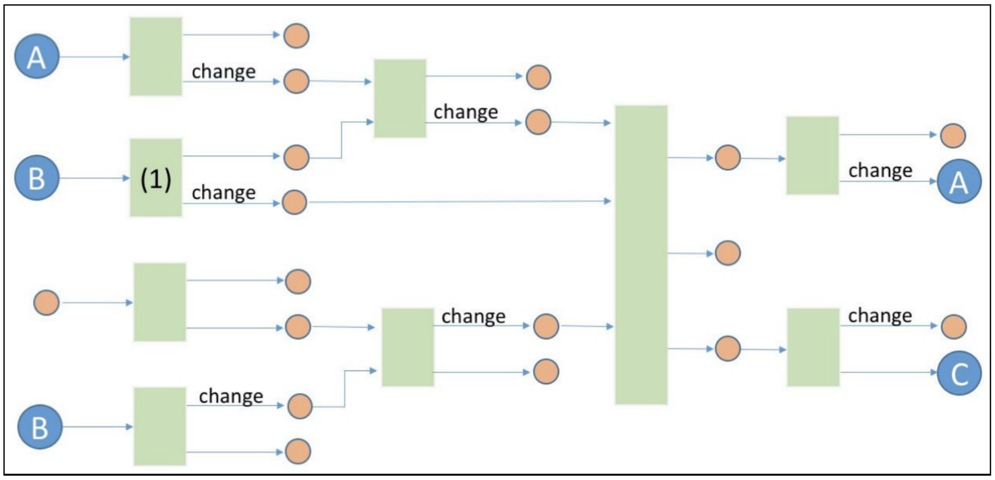
Assignment #3

Due: 11:59pm on Mon., **Dec. 3, 2018**
Submit via Gradescope (each answer on a separate page) code: **9RZGVZ**

Problem 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.



Problem 2. Ethereum payment channel. Let's implement a one-sided payment channel in Ethereum using a hash function $H : X \to X$. The scheme works as follows: to setup the channel Alice chooses a random $x \in X$ and computes $y = H^{(n)}(x)$ (here $H^{(n)}(x)$ means iterating the function H n times starting at x, so that $H^{(2)}(x) = H(H(x))$). Alice 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 a total of k units (for $k < n$), she sends Bob the value $x_k = H^{(n-k)}(x)$. Of course, Alice can send these k units one at a time, by sending $x_1 = H^{(n-1)}(x)$ to Bob, then $x_2 = H^{(n-2)}(x)$, and so on.

a. Write an implementation of this payment channel contract in Solidity, using the `sha3` hash function. Alice and Bob's addresses are hard-coded into the contract, as is the value y and the timeout when the channel expires and the contract calls `selfdestruct`. Your contract should support two methods `withdrawBob` and `withdrawAlice`.

Note: unlike the Bitcoin payment channel, Alice only sends a hash value to Bob to spend a token. She does not need to compute a signature per token, which makes this scheme well suited for Alice running on a very low-power device. As an aside, we point out that one can implement Alice quite efficiently – using only $O(\log n)$ storage she need only evaluate H twice per token (amortized). See this paper `https://eprint.iacr.org/2002/001` if you are curious to see how.

b. What security property should the hash function H satisfy to ensure that Bob cannot steal more money than Alice intended to give him?

c. As a function of n and k, how many hashes does the contract need to compute before distributing funds? How much data will Bob need to store?

d. In practice, we want to minimize the resources consumed by the contract as gas is expensive. Since the above scheme is a linear chain, you might guess that it can be improved using a tree structure. Describe in code or pseudocode an improved scheme using a Merkle tree that reduces the gas cost to only $O(\log n)$ hashes. What are the storage and computation costs (in terms of n and k) for Alice, Bob?

Problem 3. Vulnerable 3-party payment channel. Three parties, A, B, and C, are constantly making pairwise payments and thus design a 3-party payment channel based on the revocable hashed timelock contracts we saw in class. At each step, A gets a revokable commitment that it can sign and submit with three outputs, one for B, one for C, and one that A can spend 48-hours after the transaction is mined, but either B or C can spend immediately given a hash preimage initially known only to A (and released by A to invalidate the transaction). Similarly, B and C each gets a corresponding commitment transaction with an output that either of the other two parties can claim given a hash preimage. Explain how two colluding parties may be able to steal funds from the third.

Problem 4. Auditability. Two parties, A and B, share a 2-of-2 multisig cold storage address with a large sum of Bitcoin. Each wants to be able to access the funds if the other becomes unavailable, so they come up with the following scheme to stop hackers. They jointly sign two transactions, t_1 and t_2, with the following properties: t_2 spends the funds in the 2-of-2 cold storage address and makes them available under a 1-of-2 multisig scheme (so either party can spend them). However, t_2 is not valid until 48 hours after t_1 has been submitted to the blockchain. During those 48 hours, if either A or B sees t_1 on the blockchain and decides the other party has been hacked, that party can unilaterally invalidate t_2, leaving the funds in the 2-of-2 multisig transaction.

Explain how to implement such a scheme in Bitcoin. Assume you have segwit and the input sequence field can be used for relative timelocks. Also assume transaction fees are bounded, so that t_1 and t_2 can be created in advance with sufficient fees to be mined. Hint: t_1 should spend only a tiny sum—its only purpose is to declare the intent to access cold storage funds.

Problem 5. Penalty-free payment channels. Design a payment channel in which there is no penalty (or only a negligible cost) for submitting a revoked commitment—if one party attempts to close the channel using an old transaction, the other party simply disables that obsolete transaction and then submits the most recent commitment transaction to close the channel properly. Assume the same conditions as the Auditability problem (namely segwit, relative timelocks, and bounded transaction fees). Hint: use a similar technique to the Auditability problem.

Problem 6. Correct 3-party payment channel. How would you design a correct payment channel between three parties, A, B, and C? Assume the same conditions as the Auditability problem and use your design for a penalty-free payment channel.

Problem 7. Ethereum mixing. Let's implement a CoinJoin-like protocol in Ethereum that does not rely on any external anonymity infrastructure like Tor. Assume that three parties (call them Alice, Bob, and Carol), have established the following: Alice has a random `uint160` array k_a that only she knows, Bob has a random `unit160` array k_b that only he knows, and Carol has a random `unit160` array k_c that only she knows. These arrays are all the same length and satisfy $k_a[i] \oplus k_b[i] \oplus k_c[i] = 0$ for $i = 0, 1, 2, \ldots$ (the $\oplus$ operator is a bitwise exclusive-or). You may assume that these arrays are as long as you need them to be. There are several cryptographic protocols that could establish these arrays, including by means of an Ethereum contract, but you do not need to implement that part.

a. Write Solidity code to implement a mix contract (analogous to CoinJoin) between Alice, Bob, and Carol using these random arrays. The challenge is to enable each of the three users to specify their 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 to an observer.

Your contract should require only one message from each of Alice, Bob, and Carol. You should not assume any other infrastructure beyond the Ethereum contract and the parties should not communicate with one another. 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). Concretely, your contact should contain initialization code and support two methods `receiveFunds` and `abort`. Once all three users call `receiveFunds` the funds should be disbursed to the specified output accounts.

Hint: user Alice will choose a random number i between 0 and 7 and send to the contract the vector $\text{msg}_{\text{alice}} = \big(k_a[0],\ k_a[1],\ \ldots,\ k_a[i-1],\ (k_a[i] \oplus out_{alice}),\ k_a[i+1],\ \ldots,\ k_a[7]\big)$ along with the funds. Here out_{alice} is a `uint160` that is Alice's desired output address. Bob and Carol will do the same. In case of failure, the contract will refund the funds to the parties, and they can try again if they wish.

b. Assuming all users honestly participate in the protocol, calculate the probability of failure where the funds have to be refunded to the addresses from which they were sent.

c. The proposed protocol is insecure as is. Suppose Carol is the last user to call `receiveFunds`. Show that she can alter the output addresses provided by the first two users and have all the funds sent to her. We note that this attack can be prevented with appropriate use of zero-knowledge, but you are not expected to do that here.