CS 251: Bitcoin and Crypto Currencies Fall 2015

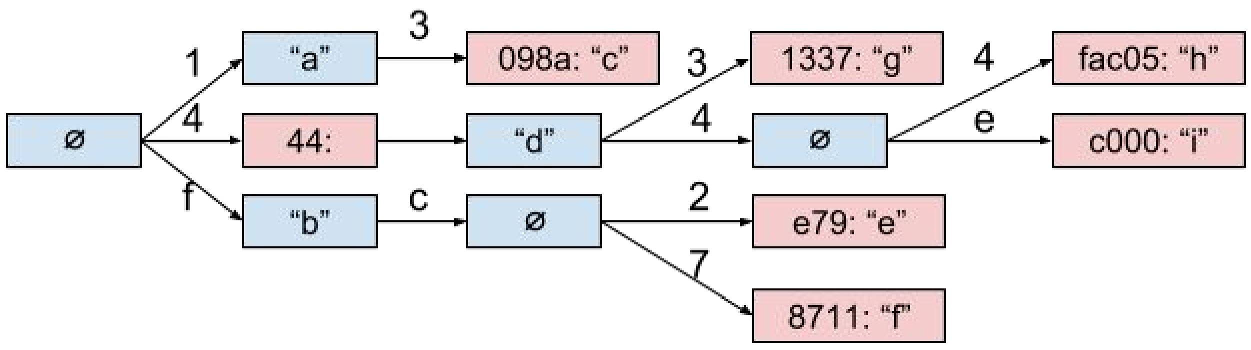
Homework 3

due: November 17, 11:59pm via email to cs251.stanford.fall.2015@gmail.com

Question 1: (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 mapped to "d":



A. Explain why in the above example there is no label on the arrow coming out of the intermediate node with the path 44.
B. 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"?
C. 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 *k* that requires proofs of length O(*k*).
D. 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.
E. 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?

Question 2: (Alternate proof-of-work) For a hash function H: $\{0,1\}^* \rightarrow \{1,\ldots,2^{256}\}$, consider the following alternate proof-of-work: given a challenge *c* and a difficulty *d*, find nonces $n_1 \neq n_2$ such that:

$$H(c,n_1) = H(c,n_2) \pmod{d}$$

That is, the miner must find *two* nonces that collide under H modulo d. Clearly, puzzle solutions are easy to verify and the difficulty can be adjusted granularly.

A. A simple algorithm to solve this problem is to repeatedly choose a random nonce *n*, and add (*n*, H(*c*,*n*)) to a set L stored to allow efficient search by n (such as a hash map). The algorithm terminates when the last hash value added to L collides with some hash value already in L. For given values of *d*, approximatey how many invocations of H are required in expectation to generate a solution n_1,n_2? How much memory does this use?
Hint: it will be helpful to familiarize yourself with the birthday paradox.

B. Consider an algorithm with limited memory that chooses one random nonce n_1 and then repeatedly chooses a random nonce n_2 until it finds an n_2 that collides with n_1. How many invocations of H will this algorithm use in expectation? We note that there is a clever algorithm that finds a solution in the same asymptotic time as part (A), but using only *constant* memory.

C. Recall that a proof-of-work is *progress-free* if for all *h,k* (where $h{\cdot}k < B$ for some large bound *B)* the probability of finding a solution after generating $h{\cdot}k$ hashes is *k* times higher than the probability of finding a solution after just *h* hashes. Is this puzzle progress-free? Explain.

Question 3: (Ethereum micropayments) Recall from class that 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?

Question 4: (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.