CS 251 Assignment 1

Luis A. Perez

Hash functions and proofs of work

Solution: We assume D is fixed in the following proof. Let $k = \log_2 D$.

As per the hint, we let $H : X \times Y \to \{0, 1, \cdots, 2^n - 1\}$ be a collision resistant hash function. We now define $H' : X \times Y \to \{0, 1, \cdots, 2^m - 1\}$ with $m \geq n + k$ as:

$$H'(x, y) = 0^k || H(x, y)$$

where 0^k is simply $000 \cdots 000$ k times and $||$ is the concatenation operator such that the output of H' is always m bits long but has k leading 0s.

We now prove that H' is collision resistant. Let us suppose that H' is not collision resistant. Then that means an adversary exists who can efficiently find $(x_1, y_1) \neq (x_2, y_2)$ such that $H'(x_1, y_1) = H'(x_2, y_2)$. However, in order for the output to be equal, it must be the case that $H(x_1, y_1) = H(x_2, y_2)$, and so (x_1, y_1) and (x_2, y_2) also serve to produce a collision in H, implying that H is not collision-resistent. This contradicts our assumption. Therefore, it must be the case that H' is collision resistent.

However, we now show that H' is not proof-of-work secure. To see this, suppose we are given a puzzle $x \in X$. Then we can pick any $y \in Y$ and we will have the following:

$$\begin{aligned}
H'(x, y) &= 0^k || H(x, y) && \text{(Definition of } H') \\
&< 2^n && \text{(Definition of } H) \\
&< 2^{m-k} && (n \leq m - k) \\
&< \frac{2^m}{2^k} && \text{(Properties of exponents)} \\
&< \frac{2^m}{D} && \text{(Defintion of } D)
\end{aligned}$$

As such, we have that H' is not proof-of-work secure.

Beyond binary Merkle trees

Solution:

a. Let $S = (T_1, \cdots, T_9)$. Let H be our hash function, and let $h_i = H(T_i)$ for $1 \leq i \leq 9$. Alice will then compute $h_{10} = H(h_1||h_2||h_3), h_{11} = H(h_4||h_5||h_6), h_{12} = H(h_7||h_8||h_9)$ (interior nodes of the ternary Merkel tree) and finally compute $h_{13} = H(h_{10}||h_{11}||h_{12})$. She will then commit to S by giving Bob h_{13}.

To later prove to Bob that T_4 is in S, Alice need to provide Bob with h_5, h_6, h_{10} and h_{12}. To verify that T_4 is indeed in S, Bob need to simply compare $H(h_{10}||H(H(T_4)||h_5||h_6)||h_{12})$ to h_{13} (the commit from earlier).

b. Generalizing to the case where $S = (T_1, \cdots, T_n)$ and where we use a k-ary Merkel tree, we can see that in order to proof to Bob that $T_i \in S$, we must provide every sibling hash along the path from the root to T_i. This is the only way that Bob can compute the correct hash for the node, without having access to the other T_i values.

Since a k-ary tree has height $\log_k n$ and we must provide $(k-1)$ hashes for each level, we have that the total proof size must be:

$$(k-1)\log_k n = \frac{k-1}{\log_2 k}\log_2 n$$

Note that this works for the case where $k = 2$ (binary Merkel tree proof size is $\log_2 n$) and also works for the example above ($k = 3, n = 9$) since we have $2 * \log_3 9 = 4$ which is exactly the proof we provided.

c. For large n, it is better to use a binary tree. This is because:

$$\frac{2}{\log_2 3}\log_2 n \approx 1.2618 * \log_2 n > \log_2 n \text{always}$$

so the proof size of a ternary tree will always be larger than the proof size needed for a binary tree. As such, if we want to minimize proof size, it's always better to use a binary tree.

Bitcoin Script

Solution:

a. To redeem the transaction, we need the program `ScriptSig` || `ScriptPubKey` to end with 1 on the stack. Given the `password` and the provided `SriptPubKey`, the `ScriptSig` script needs to simply push `password` onto the stack.

```
<password>
```

b. If she choses an 8 character password, her bitcoins can easily be stolen. The UTXO is readable by anyone, so everyone will be able to see the hash of her password. As such, an attacker simply need to compute the hashes of all possible 8 character passwords (which we assume is do-able in a reasonable amount of time). In this way, the attacker can discover `password`, and can then generate a valid `ScriptSig` (using the plain-text password) to spend the bitcoins.

c. Even if Alice choses a 20 character passphrase, her bitcoins are still not safe. When she tries to redeem her bitcoins, Alice will have to reveal her 20 character passphrase when she submits `ScriptSig` to the network. She must do this since this is the only script she knows that hashes to the address she set in `ScriptPubKey`. Other scripts might exist, but it's infeasible to find them given that SHA256 is collision-resistent.

However, the `ScriptSig` script is trivial and does not validate Alice's identity. As such, any attacker can simply take the `ScriptSig` Alice submits to the bitcoin network and create his/her own transaction transferring the funds to him/her. Both transactions will be valid, so the network will accept one at random. The attacher can increase the probability of stealing the funds by submitting many such transactions (only one will be accepted) and thereby steal Alice's bitcoins.

BitcoinLotto

Solution: We describe in a bit more detail the original design, to make sure everyone is on the same page. Basically, the lottery exists as a UTXO on the bitcoin network, with the `ScriptPK` = `P2PKH` where the address of the UTXO is the publicized public key, pk_L, (so everyone knows the funds exist). The winning ticket for the loterry will then contain the *private key* (sk_L) (corresponding to pk_L) *required* to sign a transaction which has this UTXO as input.

As such, the loterry winner will be able to spend the UTXO.

a. We can modify the above slightly to make sure that the winner can only redeem the bitcoin in one week's time (as opposed to as soon as the winner receives the private key). The simplest way to do this is to instead use `ScriptPK` = `P2SH` (Pay to script hash). Then the UTXO on the bitcoin network is published with `ScriptPK` given by:

```
OP_SHA256
<Hash of Redeem Script>
OP_EQUAL
```

where we use the following redeem script:

```
<1 week from start of lottery>
OP_CHECKLOCKTIMEVERIFY
OP_DROP
<pk_L>
OP_CHECKSIG
```

where pk_L is the public key for the lottery. As such, the only way this UTXO can be spent is if one week has elapsed (otherwise CHECKLOCKTIMEVERIFY will fail) and the following `ScriptSig` is provided:

```
<sig_L>
<redeem script>
```

This means that the lottery winner can only spend the UTXO (using s_L to compute sig_L) after 1 week has passed. Otherwise, the UTXO cannot be spent. Note that the winner will still have complete ownership of the secret key (he can use it to sign messages that other can verify using p_L). However, the spending can only occur once this weeks lottery is over.

b. As per the hint provided in Piazza, we assume that the lottery needs to be run for a maximum of N weeks (which is some finate number). We will once again make use of `ScriptPK` = `P2SH`, with a more complicated reedem script. For $1 \leq i \leq N$. Let pk_{Li} be the public key associated with the winnings from week i and sk_{Li} the associated secret key (which is printed by the trusted scratch-off printing factory and released only during week i).

Now, let UTXO_i be the unspent transaction published on week i (containing the bitcoins that will be won). UTXO_i will have the following for its `ScriptPK`:

```
OP_SHA256
<Hash of Redeem Script>
OP_EQUAL
```

where the redeem script for week i is a combination of a $1-of-i$ redeem script and a frozen funds script. To be more precise, the script is as follows (where '//' begins a comment line and is ignored)

```
// The intent here is to use this to verify that this prize
// cannot be redeemed until week i has passed.
<i weeks from start of lottery>
OP_CHECKLOCKTIMEVERIFY
OP_DROP

// This is a 1-of-i script which verifies the winner has the
// secret key from week i OR from ANY of the FOLLOWING weeks.
// This allows a subsequent winner to redeem previous lotteries,
// as long as UTXO_i hasn't already been spent.
//
// Note that this mean that if a lottery winner finds his ticket
// too late, the money may have already been claimed by a
// subquent winner.
<1>
<pk_(N - i + 1)>
...
<pk_(N-2)>
<pk_(N-1)>
<pk_N>
<i>
OP_CHECKMULTISIG
```

The comments inline above explain how the script works. As such, the only possible way to spend UTXO_i is to provide the `ScriptSig` like the following:

```
<0>
<sig_X>
<redeem_script>
```

The very first thing the redeem script will do is verify that the transaction is being redeemed at some time after week i (this prevents winners from claiming their loot too early). It will fail otherwise. Next, it will verify that the provided signature (sig_X) is valid under at least one of $\{pk_{N-i+1}, \cdots, pk_N\}$. By construction, the only people capable of generating a verifiable sig_X would be those with a secret key from week i or later (so either they win and redeem in Week i, or they win in a later week and redeem a prize from a previous week).

Note that this design does not require the involvement of the adminstrator to roll-over funds, and as such the administrators cannot embezzle any of the funds.

Similary, earlier winner cannot user their keys to win later prizes. By the design of the blockchain, later week winner cannot redeem previous week prizes that have already been redeemed (since a UTXO can only be spent once).

In conclusion, the lottery design above satisfies all of the required properties.

Lightweight clients

Solution:

a. This question is simply asking how to verify that a particular transaction (Tx) is in the block chain. We assume Bob has the Tx already. In order to proof to Bob that the transaction is in the block chain, Alice must send Bob the following:

- The sibling hashes along the Merkle branch containing Tx (labeled).
- The block headers of every block between the block containing Tx and the block header (includes the block header of the block containing Tx) in the sequence in which they appear in the block-chain.

With the above, Bob can be sure that the Tx is in the chain ending with his trusted header. He simply computes the Merkel root (using the sibling hashes provided by Alice) of Tx, and verifies this is included in the block header which Alice claims contains Tx. This first step verifies that Tx is embedded in the specified block.

Next, Bob computes the hash of the header, and verifies this is the next header (as the prevhash) provided by Alice. He continues doing this with the chain of headers, until finally, he verifies that the hash of the header immediately before the current head of the block-chain is the same as the hash specified by his trusted header.

If this is the case, Bob can rest assured that the transaction has been written into the block chain (as long as he trusts his currnet header).

b. The proof requires $(k-1)$ block headers (the block including the transaction up to (but not including) the current head header). Each block header in Bitcoin is 80 bytes. It also requires approximately $\log_2 n$ sibling hashes, where each hash is 256 bits or 32 bytes. As such, the total size of the proof should be approximately:

$$80 * (k-1) + 32 * \log_2 n$$

Plugging in $k = 6$ and $n = 1024$, we have:

$$80 * (k-1) + 32 * \log_2 n = 80 * (6-1) + 32 * \log_2 1024 = 400 + 320 = 720 \text{ bytes}$$

So overall, the proof size is less than $1kB$.

c. The process described is better understood in terms of bits. To reiterate, if we're at block header number n, we first find the largest power of 2 (2^i) that divides n, and then compute $m = n - 2^i$. This block header number m is the one that n points to.

We know that any number n can be written in base 2, as the sum of powers of 2. Written in this form, the largest power of 2 that divides n would simply be the smallest power of 2 in the representation. To be more precises, we write n in binary.

For clarity we provide an running example, suppose $n = 78$. Then we can write $n = 64+8+4$, which means we can write $n = 1001100_2$. From the way we've written n, we can immediately see that the largest power of 2 dividing n is the smallest power of 2 in the sum. So $n = 78$ would point to $n = 64 + 8 = 72$ (we subtracted 4 off). In binary, we have $m_1 = 1001000_2$ (the least significant 1 is converted to 0). We can repeat this process, since now the largest power of 2 dividing 72 is clearly 8, so we have $m_2 = 64 = 1000000_2$ (again, we flip the least significant 1 to 0).

From the above, we see that from any block n, if we follow the new pointers, we'll need at most $\log_n$ intermediate blocks to verify a transaction in the genesis block (block 0) (we just need to verify these intermediate chains).

However, from the above, it's not immediately clear that we can actually provide much shorter proofs for arbitrary header block n and transaction block $n-k$ (a block k steps before us). To see this, suppose we start with $n = 32 = 100000_2$ and $k = 31$ (so we want to verify a transaction in block 1). Block 32 has the hash of the genesis block (Block 0), but we can't use this to verify the validity of Block 1. Is the best we can do in this case then to revert back to our previous strategy and verify all intermediate blocks between Block 32 and Block 1.

Thankfully, the answer is no! From the above, we know that the "new" pointers points to a block whose binary representation is the same as the current block, except where the least significant 1 has been turned to 0. What happens if we're at a block with many low-significnat 0s, and follow the "regular" hash pointer? This will point us to a block whose binary representations is the same as our, except *the most significant 1 is now 0* and *a low* 0*s are now* 1*s*.

For our above example, the "normal" pointer points from block 32 to 31.

```
100000 (32)
011111 (31)
```

We now have sufficient information to provide an algorithm for Alice to know which block headers she must send to Bob in order for Bob to verify his transactions. In fact, Alice must provide the following:

- The sibling hashes along the Merkle branch containing Tx (labeled).
- The block headers as specified by the algorithm described below (each block header is 80 + 32 = 112 bytes).

The algorithm to determine which block headers are needed is as follows. Let n_h the the block number of the current head, and n_{tx} the block number of the transaction block.

- Find m_h corresponding to n_h (eg, $m_h = n_h - 2^i$ where 2^i is the larger power of two that divides n_h. This can be done quickly by simply flipping the least-significant 1 to a 0.

- If $m_h < n_{tx}$ (eg, following the "new" pointer takes us past the tx block), set $n_h = n_h - 1$ and include the header for block $n_h - 1$ in the proof.
- Otherwise, follow the "new" pointer. This means we set $n_h = m_h$ and we include the header for block m_h in the proof.
- Continue this process until n_h is equal to n_{tx}.

The above set of headers, by construction, will be sufficient for Bob to verify that the transaction is indeed included in the block-chain since he now has a path from n_{tx} to n_h (the trusted head).

The question now arises – how large is this proof, in terms of n (the number of transactions in a block) and k (the number of blocks before the current head where the tx is included). The easiest way to see how many block headers are included is to think about the bit operations that we defined above ("new" and "normal") and their effect. n and $n-k$ can differ **at most** in $\log_2 k$ bits. Starting with the most significant bit in which n and $n-k$ differ and using at most a single "normal" operation and at most $\log_2 k$ "new" operation, we can always make this most significant differing bit **match**. We provide an example below, where we have $n = 63 = 111111_2$ and $n - k = 1 = 000001_2$. Note that n and $n - k$ differ on $\lfloor \log 62 \rfloor = 5$ bits.

```
111111 (63)
111110 (62, follow new pointer)
111100 (60, ...)
111000 (56, ...)
110000 (48, ...)
100000 (32, ...)
011111 (31, follow normal pointer)
```

We now have a value $n = 31$ that differs from $n - k = 1$ by only 4 bits (the most significant bit now matches). This process took only $\log_2 k$ steps.

As such, if in $\log_2 k$ we can force one bit to match (in the worst case), and there are at most $\log_2 n$ bits that differ, we have that the total number of steps in the algorithm is $O(\log_2^2 k)$. As such, the worst-case proof size in bytes is:

$$\underbrace{112 * O(\log_2^2 k)}_{\text{header}} + \underbrace{32 * \log_2 n}_{\text{Merkel tree proof}}$$