CS 161 Project 2: Mini Cryptocurrency

Due on October 27 at 3:00 PM.

Introduction

In this project you will implement parts of a Bitcoin-like cryptocurrency. Your code will validate blocks and transactions, organize them into a blockchain, and compute the balances corresponding to accounts. You'll also explore how implementation errors such as the use of insecure random number generators can lead to insecurities.

For a conceptual description of cryptocurrency transactions and blocks, see the Bitcoin paper, particularly sections 2, 3, and 4:

https://bitcoin.org/bitcoin.pdf

The owner of a coin is an elliptic curve public key. Knowing the corresponding private key enables you to spend the coin. A chain of transactions tracks the ownership of each coin; a transaction is the public key of the new owner and a hash of the previous transaction, signed by the old owner. A block contains transaction and the hash of a previous block. A proof of work function causes there to be only one valid chain of blocks and prevents double-spending.

Some ways in which the project differs from Bitcoin:

- Coins are indivisible and each transaction moves exactly one coin from one public key to another. (In Bitcoin, the number of coins can be anything and there can be multiple inputs and outputs.)
- A block contains at most two transactions: the "reward" transaction `reward_tx` that goes to the miner of a block, and an optional “normal” transaction `normal_tx` that moves a coin from one address to another.
- The difficulty of mining a block is much lower. You need to brute-force only 24 bits.
- No network access: you'll be simulating everything on one computer.
- Bitcoin allows complicated predicates to decide when a transaction may occur; we simply hard-code the common case of requiring a public key and a signature.

For the implementation of the crypto operations you will use the OpenSSL library. The important operations are SHA256 for hashing and ECDSA for digital signatures. OpenSSL is already installed on Hive. Here are links to documentation of some of the functions you will be using:

https://wiki.openssl.org/index.php/Elliptic_Curve_Cryptography
https://www.openssl.org/docs/manmaster/crypto/bn.html
https://www.openssl.org/docs/manmaster/crypto/ec.html
https://www.openssl.org/docs/manmaster/crypto/ecdsa.html
https://www.openssl.org/docs/manmaster/crypto/sha.html

As in Project 1, the project source code comes with a Makefile. To build it, run:

```
make
```

Step 1. Organize the blockchain

The project source code builds two executable programs: “balances” and “genkey”. genkey is used in the next step; you can ignore it for now. In this step, you will write the “balances” program. As in Project 1, the places where you have to write code are marked with `/* TODO */` comments. You are free to modify anything in balances.c.

In the “blocks” subdirectory there are a bunch of .blk files. Your job is to read the files into memory, arrange them into a tree, and decide which blocks are valid and which are invalid. Then you will find the longest valid path from the root to a leaf (the “main chain”), and compute the final sum of coins held by each public key you see. Run the balances program like this:

```
./balances blocks/*.blk
```

You can use the `block_read_filename` function to read a block from a file. The provided `struct blockchain_node` is one way to organize the blocks in memory; but you can do whatever you think is most convenient. You will have to sort the blocks or make multiple passes over them, because they are not guaranteed to be provided to your program in order. (Hint: the `height` member defines how far from the root the block must appear.) Not all the .blk files represent valid blocks. Only valid blocks count for determining the main chain.

Rules for checking the validity of a block:

- If the block has height 0 (the “genesis block”), its SHA256 hash must be the hardcoded value 0000000e5ac98c789800702ad2a6f3ca510d409d6cca892ed1c75198e04bdeec. (Use the `byte32_cmp` function.) If a block has height ≥1, its parent must be a valid block with a height that is 1 smaller.
- The hash of the block must be smaller than `TARGET_HASH`; i.e., it must start with 24 zero bits. (Use the `hash_output_is_below_target` function.)
- The height of both of the block's transactions must be equal to the block's height.
- The `reward_tx.prev_transaction_hash`, `reward_tx.src_signature.r`, and `reward_tx.src_signature.s` members must be zero—reward transactions are not signed and do not come from another public key. (Use the `byte32_zero` function.)
- If `normal_tx.prev_transaction_hash` is zero, then there is no normal transaction in this block. But if it is not zero:
  - The transaction referenced by `normal_tx.prev_transaction_hash` must exist as either the `reward_tx` or `normal_tx` of an ancestor block. (Use the `transaction_hash` function.)

- The signature on `normal_tx` must be valid using the `dest_pubkey` of the previous transaction that has hash value `normal_tx.prev_transaction_hash`. (Use the `transaction_verify` function.)
- The coin must not have already been spent: there must be no ancestor block that has the same `normal_tx.prev_transaction_hash`.

You can compare these rules for the Bitcoin rules for transactions and blocks.

Once you have built the tree of blocks, the main chain is the path from the root to the valid block of maximum height. Walk along the path and track the number of coins held by each public key. Every block contains a reward transaction that adds a coin to a public key. It may also (if `normal_tx.prev_transaction_hash` is not zero) contain a normal transaction that removes a coin from one public key (the key found in a previous transaction having hash valud `normal_tx.prev_transaction_hash`) and adds a coin to another public key (`normal_tx.dest_pubkey`). Therefore in each block, you will do one increment for the reward transaction, and then optionally one decrement and one increment for the normal transaction. The total number of coins in circulation increases by 1 with every block.

The output of the balances program is a table of public key *x*-coordinates and their corresponding balances. Here is a test case. This is what the output of your program should be when it is run on the genesis block only:

```
./balances blocks/e04bdeec.blk
4f0236ab842a6b2d0eb01c17424d3a566c773b38b5628c55fbf60e08ec0c0d88 1
```

This output means that the public key with *x*-coordinate 4f0236… has 1 coin. That's because that is the key that mined the genesis block.

When we test your code, we will give it a different collection of block files.

This part of the project is a bit of a programming challenge. You will have to build a few of your own data structures and auxiliary functions inside balances.c. For example, you probably want a function that searches the block tree for a transaction that has a specific hash value, and a function that decides the validity of a block. It's all right if your algorithms are not efficient. We won't test your code on huge blockchains. (This is because typically you would use an external library for data structures such as linked lists and hash tables; implementing those is not the point of this project.)

Step 2. Take advantage of weak transactions

In this step you will show how it is possible to steal coins if you can guess the private key that corresponds to a public key.

Start by generating a key that will receive the stolen coins:

```
./genkey mykey.priv
```

You do not have to do anything to the genkey program. It is ready to run. You will submit the file mykey.priv but you will not submit genkey.c.

Implement the `block_mine` function in block.c. There is nothing tricky here. Just increment the block's nonce until `hash_output_is_below_target`. (Use the function `block_hash` to get the hash value of a block.) There is a debugging function `block_print` that may be useful while you are developing. It should not take longer than 2 minutes to mine a block.

The block at height 4 sends a coin to an address that was generated using a weak PRNG. In terms of the functions in genkey.c, it was generated like this:

```
unsigned char buf[32];
int i;
srand(1234);
for (i = 0; i < 32; i++) {
      buf[i] = rand() & 0xff;
}
return generate_key_from_buffer(buf);
```

Modify genkey.c and generate a such a key. Verify that its public key matches `normal_tx.dest_pubkey`. Note: You must run this code on a Hive machine. This is because the `rand` and `srand` functions may work differently on other systems. If you don't get a matching public key, stop and try again on Hive.

Now mine a new block that transfers a coin from the weak public key to your own mykey.priv. You can do this in balances.c if you want, or in a separate program. A code outline:

```
/* Build on top of the head of the main chain. */
block_init(&newblock, &headblock);
/* Give the reward to us. */
transaction_set_dest_privkey(&newblock.reward_tx, mykey);
/* The last transaction was in block 4. */
transaction_set_prev_transaction(&newblock.normal_tx,
      &block4.normal_tx);
/* Send it to us. */
transaction_set_dest_privkey(&newblock.normal_tx, mykey);
/* Sign it with the guessed private key. */
transaction_sign(&newblock.normal_tx, weakkey);
/* Mine the new block. */
block_mine(&newblock);
/* Save to a file. */
block_write_filename(&newblock, "myblock1.blk");
```

Mine a new block that contains a transaction that transfers the coin to the mykey.priv account. Save myblock1.blk; it is one of the files you will submit.

The block at height 5 was mined by a private key that used a time-based PRNG seed:

```
unsigned char buf[32];
int i;
```

```
srand(time());
for (i = 0; i < 32; i++) {
      key[i] = rand() & 0xff;
}
return generate_key_from_buffer(buf);
```

It was generated around the time 2015-10-01 12:00:00 UTC. Generate the same key (you will have to try many candidate time offsets near the target time). Mine a new block that transfers the `reward_tx` of block 5 to mykey.priv. Build off of myblock1.blk and save as myblock2.blk.

Concluding Questions

In a separate file called answers.txt, please put down the answers to the following questions:

1. What is the current Bitcoin mining difficulty? (You can find it online at https://bitcoinwisdom.com/bitcoin/difficulty.)

2. How long does your computer take to mine a block? We don't need a precise answer.

3. The quantity difficulty$\times 2^{32}$ is the expected number of hash iterations needed to mine a Bitcoin block. How long would it take for your computer to mine a Bitcoin block, if a Bitcoin hash operation takes the same amount of time as a hash operation in your code for this project?

4. The Bitcoin difficulty is dynamically tuned so that it takes on average 10 minutes to mine a block. A DES key is 56 bits long. How long does it take the current Bitcoin network to compute 2^{56} hash operations?

Submission instructions

Submit your files using the glookup system on a Hive machine. Enter your source code directory and run the command:

```
submit proj2
```

The files you should submit are:

answers.txt
balances.c
myblock1.blk
myblock2.blk
mykey.priv

Only one member per group needs to submit.

Submissions are due on October 27 at 3:00 PM.