SESUG Paper 107-2018

Mining Bitcoins: A step-by-Data Step Simulation

Seth W. Hoffman, GEICO

ABSTRACT

Have you ever thought about investing in a crypto-currency, but first wanted to understand more about how they work? The best way to understand a computer program is to read its software code and see it run. This paper walks through a Base SAS® simulation of a Bitcoin [1] miner to demonstrate all the parts needed to implement the Bitcoin payment system.

INTRODUCTION

Bitcoin is one particular example of what is known as "Blockchain technology". A blockchain is a type of secure distributed ledger. The ledger is made up of a chain of blocks. Each block contains transaction information and a link to the previous block. Bitcoin miners maintain the integrity of the system by making sure transactions follow all the rules, and only including those that do in blocks.

Three main concepts provide the security and functionality of the ledger: hash functions [2] (not to be confused with the Hash Object), signing [3], and proof-of-work [4]. By implementing these ideas in the same exact way, the network of Bitcoin miners create the Bitcoin network. An overview of these topics is provided, but in-depth discussions are beyond the scope of this paper.

A hash function is an algorithm that translates an input message of any length into an output message of a fixed length. Several additional properties are required for these functions to be useful for cryptography applications. One is that the same input should always produce the same output. The next is that a small change in the input should produce a large change in the output. Finally, given the output, it should be impossible to determine the original input. Bitcoin makes extensive use of hashes for both data reduction and for linking transactions to blocks and linking each block to the last one.

Signing messages with public key cryptography is another important idea made use of by Bitcoin. A user of a public key system will have two encryption keys. One key is kept private and the other is broadcast to the public. Messages encrypted with either key can be decrypted by the other. In this way, a message is said to be "signed" if it is encrypted with the private key as it can be decrypted with the public key that is known to belong to the signer. Every Bitcoin transaction has its own "address". This address is related to the public key of the coins in that transaction.

Miners do their jobs for the possibility of winning the race to mine the newest block, earning them Bitcoins. The rule for creating new blocks is based on the idea called proof-of-work. Simply, this means every miner spends time working to solve a hard problem, and finding the solution is the proof that a miner worked to solve it. Bitcoin's hard problem is finding a hash value of the block it is currently mining such that the value is less than a certain number called the "difficulty target".

IMPORTANT CONCEPTS – SAS EXAMPLES

Before getting to the miner simulation, the preceding concepts will be illustrated with basic SAS code.

HASHING

As of release 9.4 M1, Base SAS includes the common cryptographic hash function "SHA 256" [5]:

```
DATA secure_hash_256bits;
   FORMAT hash_output $hex64.;
   hash_output = sha256("SAStoshi PROCamoto");
RUN;
```

The output dataset now contains the value:

7AB4C32981BE22D1EE0C95287F8E0B8146DE60B6DD4BDE1F0A2F8DDC9B74406A

The random number generator, which users may be more familiar with, also has the properties needed for a hashing function:

```
DATA  NULL ;
   Max message = 2147483647; /*Max input to stream init, 2^31 - 1*/
   message = 836583; /*ASCII: A=65, S=83 --> 83 || 65 || 83*/
   CALL STREAMINIT(MOD(message, max message));
   hash output = ROUND(RAND("Uniform") * 1000000000, 1);
   PUT hash_output; /*164427464*/
RUN;
```

Translating every character into its ASCII value allows them to be concatenated into a large number. This number is then used as the seed value for the random number generator. Every time the code is ran, it will produce the output. If the input is changed slightly, to 836582 for example, then the output is vastly different: 510979032377. Given the output 991631893208, it would be pretty surprising if anyone could figure out the original input.

PUBLIC KEYS - SIGNING

Bitcoin uses complicated math involving elliptic curves as the basis of its public key system. The original, and much more easy to understand, public key cryptography system is based on some interesting math involving prime numbers and modular arithmetic:

```
DATA public key example;
   modulus = 3131; /* = 31 x 101, small primes are not secure */
   private key = 1663; /*derived by interesting math from [31,101] */
   public key = 727;  /*derived by more interesting math from [31, 101]*/
   plain text = 2723;
   cypher text = plain text;
   DO i = 1 TO (private key - 1); /*some modular arithmetic to encrypt*/
      cypher_text = MOD((plain_text * cypher_text), modulus);
      END;
   PUT cypher text; /*2909, what could this really mean ??? */
   recovered text = cypher text;
   DO i = 1 to (public key - 1); /*some modular arithmetic to decrypt*/
      recovered_text = MOD((cypher_text * recovered_text), modulus);
      END;
   PUT recovered_text; /*2723 matches the original message*/
RUN;
```

The address from which coins are sent is related to the public key of the sender. An example address could be constructed by concatenating the modulus, a separator (e.g. “-”), and the public key. Following the above example, the address would then be 3131-727. To prove ownership, along with the address, both the original plain text and the encrypted cypher text would be sent. Since only the holder of the private key could have encrypted this message, the act of encrypting is equivalent to putting their notarized signature on the message. Anyone on the network can use the public key and the modulus to decrypt the cypher text and see if the recovered text matches the original plain text. In a real application the private key would be over one thousand digits long, making guessing the private key a much harder job than in the above example.

PROOF-OF-WORK

Bitcoin defines a hard limit for the size of each block. Once a miner has enough verified transactions to approach that limit, it will create a block consisting of a header and the body. The header for the block contains information about the body, some technical information, and a value called “the nonce”. The

miner will keep trying different nonce values till it find a hash value of the header that is less than some difficulty target:

```
DATA proof of work;
   block header_hash = 836583;
   nonce = 0;
   difficulty target = 1000;
   hash output = difficulty target  + 1; /* loop runs at least once */
   DO UNTIL (hash output <= difficulty target);
      CALL STREAMINIT(block header hash + nonce);
      hash output = ROUND(RAND("Uniform") * 1000000000000, 1);
      nonce = nonce + 1;
      END;
RUN;
```

For this particular header, the nonce turned out to be 111,731,892 giving `hash_output` the value "000000000698". The log shows how much work it took the computer to find this value:

```
NOTE: The data set WORK.PROOF_OF_WORK has 1 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           7.10 seconds
      cpu time            7.07 seconds
```

Increasing the difficulty by changing the target to 500 required 39.69 seconds of CPU time. Moving the difficulty target down to 200, it took longer than the system policy's allowed runtime of 30 minutes! To summarize, it takes many CPU cycles to find an acceptable answer, and that answer is proof of the work done to find it.

SIMULATED MINER

A miner's job is to create valid blocks. To be valid, the blocks must contain valid transactions. The only fool-proof way to do this is to download the entire blockchain and validate every transaction of every block. Validation requires making sure the sender owns the coins they are spending, and that they are not spending more coins than they own. In addition, every miner must operate in the same way. If one miner doesn't follow all the requirements, its blocks may be rejected by others and not included in their copies of the blockchain.

BLOCK STRUCTURE

Besides the previously discussed concepts, it is also very important to know how a block is structured. Each block contains three parts: validated transactions, the coinbase transaction, and the header.

Validated transactions are made up of lists of coin inputs and coin outputs. Coins are referenced by their address, which in this example includes the public key information. The other important part of a transaction is the message. Bitcoin has a full scripting language to define the conditions for when a coin is spent. In this example, the message is simply the number of coins that can be spent. To show that the coins are being spent by the valid owner, inputs include the signature as well as how many of the available coins are to be spent.

Every block contains one coinbase transaction. The coinbase assigns coins to whoever mined the block. The reward for mining each block is determined by unanimous consensus of all the miners. Besides the automatic reward, any difference between the sum of coins in the inputs and the sum of the outputs is considered a fee paid to the miner. With no inputs, coinbase transactions are considered verified because they are part of a valid block.

Packaging everything together is the job of the header. Blocks are chained together by including the ID of the previous block in the current header. A value called the Merkle root [2] summarizes all the transactions to ensure that transaction data cannot be changed once included in a block. In Bitcoin, the difficulty target is calculated such that the average time to mine a block should be ten minutes. Finally, the nonce is a

random number that the miner increments every attempt to find a hash of the header that is less than the difficulty target.

The example blockchain in this paper includes three blocks. The first block is called the “origin block” and is special in that is contains no validated transactions and there was no previous block to be linked to. The second block’s transaction has one input and two outputs. The third block demonstrates the reward fee as the output has less coins than the input:

```
DATA blockchain;
   INPUT block_data $;
   DATALINES;
1c-3131-727-650-50
1h-164427464-468548638-1000-248008-650
2i1-3131-727-50-2773-50
2o1-5141-5-20
2o2-7373-3571-30
2c-4141-369-235-50
2h-650-628352786-1000-1960319-235
3i1-4141-369-50-2715-49
3o1-4343-1103-49
3c-4399-367-761-51
3h-235-628053884-1000-1497779-761
;
RUN;
```

WHICH COINS ARE UNSPENT?

Before SAS ® Hash Objects, having different variables for different types of transactions in meant a very wide data set. Concatenating all the variables for each type of input allows the program to parse a single input variable and break it into the needed variables for each Hash Object. The following code will “download” the existing blockchain and parse it to create a list of unspent coins:

```
DATA  NULL ;
   If  N  = 1 THEN DO; /*Define all the Hash Objects*/
      LENGTH h index h prevHash h merkle h difficulty h nonce
             h blockId c index c blockId c address c message
             i index i address i message i signature i_amount
             o index o address o message $20.;
      DECLARE HASH h(ordered:"a");
      DECLARE HITER hi("h");
         rc = h.defineKey("h index");
         rc = h.defineData("h prevHash", "h merkle",
                           "h_difficulty", "h_nonce", "h_blockId");
         h.defineDone();
      DECLARE HASH c(ordered:"a");
      DECLARE HITER ci("c");
         rc = c.defineKey("c index");
         rc = c.defineData("c_blockId", "c_address", "c_message");
         c.defineDone();
      DECLARE HASH i(ordered:"a");
      DECLARE HITER ii("i");
         rc = i.defineKey("i index");
         rc = i.defineData("i address","i_message", "i_signature",
                           "i_amount");
         i.defineDone();
      DECLARE HASH o(ordered:"a");
      DECLARE HITER oi("o");
         rc = o.defineKey("o index");
         rc = o.defineData("o_address","o_message");
```

```
      o.defineDone();
   DECLARE HASH u(ordered:"a");
      rc = u.defineKey("u address");
      rc = u.defineData("u_message")
      u.defineDone();
   CALL MISSING(h index, h prevHash, h merkle, h difficulty,
                h nonce, h blockId, c index, c blockId,
                c address, c message, i index, i address,
                i message, i signature, i amount, o index,
                o_address, o_message, u_address, u_message);
   END;

SET blockchain END=eof; /*Read in blockchain*/
/*Regular Expressions make for easy parsing of strings*/
/*header row*/
re = PRXPARSE('/h(\d)-(\d+)-(\d+)-1000-(\d+)-(\d+)/');
IF PRXMATCH(re, block data) THEN
   h.add(key:PRXPOSN(re,1,block data),
         data:PRXPOSN(re, 2, block data),
         data:PRXPOSN(re, 3, block data), data:"1000",
         data:PRXPOSN(re, 4, block data),
         data:PRXPOSN(re, 5, block_data));
/*coinbase row*/
re = PRXPARSE('/c(\d)-(\d+)-(\d+-\d+)-(\d+)/');
IF PRXMATCH(re, block data) THEN
   c.add(key:PRXPOSN(re, 1, block data),
         data:PRXPOSN(re,2,block data),
         data:PRXPOSN(re, 3, block data),
         data:PRXPOSN(re, 4, block_data));
/*inputs row*/
re = PRXPARSE('/i(\d{2})-(\d+-\d+)-(\d+)-(\d+)-(\d+)/');
IF PRXMATCH(re, block data) THEN
   i.add(key:PRXPOSN(re, 1, block data),
         data:PRXPOSN(re, 2, block data),
         data:PRXPOSN(re, 3, block data),
         data:PRXPOSN(re, 4, block data),
         data:PRXPOSN(re, 5, block_data));
/*outputs row*/
re = PRXPARSE('/o(\d{2})-(\d+-\d+)-(\d+)/');
IF PRXMATCH(re, block data) THEN
   o.add(key:PRXPOSN(re, 1, block data),
         data:PRXPOSN(re, 2, block data),
         data:PRXPOSN(re, 3, block_data));

IF eof;
/*First block is special*/
rc = hi.first(); /*start with the first block*/
rc = c.find(key:i);
rc = u.add(key:c address, data:c_message);
/*automate other blocks*/
DO x = 2 TO 3;
   rc = hi.next();
   /*Example only has 1 input per block, so no loop for inputs*/
   rc = i.find(key:COMPRESS(x || "1")); /*load the input*/
   rc = u.find(key:i address); /*find it's unspent transaction*/
   unspent coins = i message;
   DO y - 1 TO 2; /*example has 1 block with 2 outputs*/
```

```
            rc = o.find(key:COMPRESS(x || y); /*load the output*/
            IF (rc EQ 0) THEN DO; /*if the output exists*/
               unspent coins = unspent coins - o.message;
               rc = u.add(key:o_address, data:o_message);
               END;
            END;
         rc = u.remove(key:i_address); /*coins are now spent*/
         rc = c.find(key:x);
         miner reward = PUT(INPUT(CATS(c message +
                                    unspent coins),4.),$20.));
         rc = u.add(key:c_address, data:miner_reward);
         END;
      rc = u.output(dataset:"unspent_coin_transactions");
   RUN;
```

WILL VALIDATE FOR COINS

Now that the miner has a list of unspent coins, it can validate incoming transactions. A Bitcoin miner will include transactions until they have reached a certain bit limit. This example only includes one input transaction per block. The policy of which valid transactions to include are left up to the individual miner. The following example is greedy and uses the transaction that pays the highest fee:

```
   /*Results from the previous code example*/
   DATA unspent db;
      INPUT u id $9. u_message 4.0;
      DATALINES;
   5141-5    20
   7373-3571 30
   4399-367  51
   4343-1103 49
   ;
   RUN;

   DATA transactions;
      INPUT in modulus in public in message in_signature in_amount
            out_modulus out_public 4.0;
      DATALINES;
   5141 5 20 2212 20 4183 267
   4399 367 51 777 48 2201 851
   ;
   RUN;

   %LET max message = 2147483647;
   DATA mine block(KEEP=block data);
      If  N  = 1 THEN DO; /*Define Hash Objects*/
         LENGTH u id $ u message v id v message 8.0 block data $100.;
         DECLARE HASH u(dataset:"unspent_db", ordered:"a");
            rc = u.defineKey("u id");
            rc = u.defineData("u_message");
            u.defineDone();
         DECLARE HASH v(ordered:"a"); /*Validated. New outputs*/
            rc = v.defineKey("v id");
            rc = v.defineData("v_message");
            v.defineDone();
            CALL MISSING(u_id, u_message, v_id, v_message);
         END;

      SET transactions END=eof;
```

```
RETAIN best fee best_fee_id best_data;
IF  N  EQ 1 THEN DO;
   best fee = -1;
   best feeId = "";
   best_inputData = "";
   END;
/*Make sure transaction input is in the list of unspent coins*/
IF (u.FIND(key:COMPRESS(in modulus || "-" || in public)) = 0);
/*Decrypt signature to see if it matches the plaintext message*/
recovered text = in signature;
DO i = 1 to (in public - 1);
   recovered text = MOD((in signature * recovered_text),
                        in_modulus);
   END;
IF recovered text EQ in message THEN DO;
   id key = COMPRESS(out modulus||"-"||out public);
   /*only for data needed for output message*/
   v.add(key:id key, data:in amount);
   IF (in message - in amount) GT best fee THEN DO;
      best fee = in message - in_amount;
      best feeId = id key;
      best inputData = COMPRESS(in modulus || "-" || in public
                               || "-" || in message || "-" ||
                               in_signature || "=" ||in_amount);
      END;
   END;

IF eof THEN DO; /*Output the new block*/
   block data = best_inputData;
   OUTPUT;
   /*Time is money, process the best tippers*/
   rc = v.FIND(key:best fee key);
   block data = COMPRESS(v_id || "-" || v_message);
   OUTPUT;
   c address = "6497-1691"; /*Miner makes keys for its reward*/
   c message = 50 + best fee; /*2nd transaction: 51 - 48 = 3*/
   block data = COMPRESS(c_address || "-" || c_message);
   OUTPUT;

   h prevHash = 761;
   h difficulty = 1000;
   h nonce = 0;
   h blockId = 0;
   /*Merkle Root is a binary tree of hashes*/
   h merkle = 616564270;
   CALL STREAMINIT(4399+367+51+778+48, &max message);
   h h1 = ROUND(RAND("Uniform") * 1000000000, 1); /*806931926*/
   CALL STREAMINIT(2201+851+50, &max message);
   h h2 = ROUND(RAND("Uniform") * 1000000000, 1); /*421654199*/
   CALL STREAMINIT(6497+1691+53, &max message);
   h h3 = ROUND(RAND("Uniform") * 1000000000, 1); /*188708526*/
   h h4 = h h3; /*Balanced binary tree must have 2^n leaves*/
   CALL STREAMINIT(h h1 + h h2, &max message); /*2nd row*/
   h h5 = ROUND(RAND("Uniform") * 1000000000, 1); /*144325126*/
   CALL STREAMINIT(h h3 + h h4, &max message);
   h h6 = ROUND(RAND("Uniform") * 1000000000, 1); /*79633978*/
   CALL STREAMINIT(h_h5 + h_h6, &max_message); /*root*/
```

```
        h_merkle = ROUND(RAND("Uniform") * 1000000000, 1);
        /*616564270*/

        /*Block information is built. Time to mine the block!*/
        hash_output= 1001;
        DO UNTIL((hash_output < 1000) OR (h_nonce = &max_message));
           proposed_header = h_prevHash + h_blockId + h_merkle +
                             h_difficulty + h_nonce;
           CALL STREAMINIT(MOD(proposed_header, &max_message));
           hash_output = ROUND(RAND("Uniform") * 1000000000000, 1);
           nonce = nonce + 1;
           END; /*nonce = 975452*/
        block_data = COMPRESS(h_prevHash || "-" || h_blockId || "-"
               || h_merkle || "-" || h_difficulty || "-" || h_nonce);
        OUTPUT;
        rc = u.remove(key:best_fee_id);
        rc = u.output(dataset:"updated_unspent");
     END;
RUN;
```

The miner has now run once and output a new block (ID = 147). All other miners on the network will receive a copy of the new block and verify that it was properly created by checking that the header do, in fact, produce a hash in the valid target range using the provided nonce. If a valid block is received during the mining process, the miner updates the unspent coin list, refreshes the transactions included in its proposed new block and starts over trying to mine the next new block.

CONCLUSION

Many details go into the implementation of code to run a blockchain. But, designing a blockchain based crypto-currency system requires a few simple concepts. Linking of blocks is accomplished by including the previous block's ID in the header used to create the current block. Bitcoin values anonymity, coins are only referenced by the public key information in which they are stored. Creating a signature, that when decrypted by the public key, shows that the coins can be legitimately spent. Spending computer time finding making sure all the above has been done correctly can earn the miner some Bitcoins. Hopefully the rewards will be more than the cost of the electricity!

ACKNOWLEDGMENTS

I would like to thank SAS and this year's academic conference chair, Lind Sullivan for organizing and supporting another year of the Southeast SAS Users Group. I would also like to thank the section chairs, John Cohen and Chuck Kincaid for accepting my paper proposal.

REFERENCES

[1] Nakamoto, Satoshi, "Bitcoin: A Peer-to-Peer Electronic Cash System " Oct 2008. Available at https://bitcoin.org/bitcoin.pdf

[2] Mykletun, Einar, Maithili Narasimha, and Gene Tsudik. "Providing authentication and integrity in outsourced databases using Merkle hash trees." *UCI-SCONCE Technical Report* (2003). Available at https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkleodb.pdf

[3] Rivest, R.L, Shamir, A., and Adelman L. Feb 1978. "A method for obtaining digital signatures and public-key cryptosystems". Communications of the ACM, volume 21 Issue 2.:120-126. Available at https://people.csail.mit.edu/rivest/Rsapaper.pdf

[4] Dwork, Cynthia and Naor, Moni. 1993. "Pricing via Processing, Or, Combatting Junk Mail, Advances in Cryptology". *CRYPTO'92: Lecture Notes in Computer Science No. 740*. Springer: 139–147. Available at http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.ps

[5] Hemedinger, Chris. "A fresh helping of hash: the SHA256 function in SAS 9.4m1." SAS. Jan 18 2014. Available at https://blogs.sas.com/content/sasdummy/2014/01/18/sha256-function-sas94/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Seth Hoffman
GEICO
Seth.W.Hoffman@gmail.com