Course Home
Home

Administration
Overview
Course email
Anonymous feedback

Assignments
Home virtual machines
Homework dropbox
Class GoPost forum
Class Gradebook

Calendar
Course calendar
HW list

Online Resources
bitcoin/Tor Links

CSE461 Project 4: Bitcoins

Out: Thursday March 6, 2014
Due: Friday March 14, 2014 by **11:59pm**.
Teams Allowed: Yes
Teams Encouraged: Yes
Ideal Team Size: 2

Summary

You'll implement some of the crucial mechanisms used by Bitcoins. There isn't much networking involved - possibly none. But, Bitcoins employ a number of cryptographic techniques that are crucial to what the Internet is today. This project provides exposure to those techniques (not to mention Bitcoins!).

Bitcoins are a digital currency. They're digital in the sense that they can be represented by bit strings, which means that it's easy to send them over a network. They're currency in the sense that you can't just create some. Instead, you obtain them by trading something for them, for example, a pack of gum for .0001 Bitcoin, or 1 Bitcoin for 600 US dollars.

One problem Bitcoins must solve is how to keep you from minting new ones in an uncontrolled fashion. They're just bit strings, so what keeps you from creating whatever bit string represents a Bitcoin and sending it to someone? The answer to that is that a record of every exchange involving Bitcoins (called a *transaction*) is sent to everyone. So, if Alice sends two Bitcoins to Bob, Bob can verify that Alice has two Bitcoins to send by checking the ledger of transactions. Even though you can make up any bit string you want, you can't spend bit strings that represent Bitcoins unless there's a record of how you obtained them in the ledger.

There are still a couple of problems, though. One is that Charlie knows that Alice gave two Bitcoins to Bob, because Charlie receives a record of the transaction. What keeps Charlie from spending the Bitcoins by pretending to be Bob? The answer to that is familiar - when you try to spend those two Bitcoins you have to prove you're Bob by demonstrating that you know some secret information that (presumably) only Bob knows. In fact, Bitcoins doesn't really have a notion of individual people at all. Instead it has the notion of "the person who knows this secret key." A transaction sends Bitcoins to a secret key, rather than a person. For that reason Bitcoins is also anonymous, much like (traditional) cash - you don't reveal who you are when you spend it, you simply demonstrate that you know the required secret.

The final major problem for Bitcoins is how to keep Alice from spending her two Bitcoins twice. Suppose she nearly simultaneously trades them to both Bob and Charlie. Each of them consult the ledger and finds that Alice has the Bitcoins to spend, so everything looks fine, but it isn't. To handle this, we need to make a decision that one of the two transactions occurred first and that the other is therefore invalid. So, in this case, Bob may end up with the two Bitcoins and Charlie with nothing. The technical problem now is that everyone has to agree on the ordering. The entire scheme works only if the ledgers held by all participants are consistent with each other.

To handle this, Bitcoins relies on *miners* to *verify* transactions. A miner collects a group of unverified transactions into a *block*. It then solves a hard, cryptographic problem that has the block as input. Finally, it sends the block and the solution to all other participants. Each participant accepts the first solution it receives as "the truth". If the miner has seen both of Alice's transaction spending the same two Bitcoins, it will have made some (arbitrary) decision about which one came first and which was invalid. All other nodes adopt that decision. (There's still the problem that two miners may solve the problem at about the same time, leading to confusion about which was first. To address that requires details we'll see in a bit.)

Because mining is expensive (you have to pay for the computing hardware to run on and the electricity to power the hardware), Bitcoins needs to incent people to perform the mining. It does that by giving them Bitcoins -- every successfully mined block creates some number of new Bitcoins, ones that come into existence with that block. The miner is free to award these to herself.

Crypto Operations Background

Bitcoins relies on public-private key pair encryption, cryptographic hashing, and cryptographic signatures.

Public-private key encryption

Public-private key pair encryption uses a pair of keys, K_{priv} and K_{pub}. The public key can be known by anyone; it's public. The private key is known only by the secret holder.

Let `E` and `D` be the encryption and decryption functions. Then $E(K_{pub}, text)$ is the result of encrypting `text` using the public key. It is thought to be computationally infeasible to recover the text given just the encrypted form and the public key. However, $D(K_{priv}, E(K_{pub}, text)) = text$, meaning that the holder of the private key can successfully decrypt. Surprisingly, for some public-private key pair schemes, if we reverse the use of the keys we can still decrypt: $D(K_{pub}, E(K_{priv}, text)) = text$.

Finally, the scheme has other important properties that you might expect. For example, given a text and an encrypted version of the text, it isn't feasible to determine the key used to encrypt. Editing the encrypted text has unpredictable effects on the result of decrypting it.

> Our implementation of Bitcoins uses RSA-1024.

Cryptographic hashing

A cryptographic hash is a *one way function*, `H(data) = h`. Like other hash functions, it takes data of arbitrary size and reduces it to a fixed size. It's one-way in the sense that evaluating `H(data)` is cheap, but it is believed that there is no computationally efficient way to find `data` that hashes to a particular value `h`.

Our implementation of Bitcoins uses SHA-256, which maps arbitrary data to a 256-bit (32-byte) value. Following the actual Bitcoins implementation, we apply SHA-256 twice whenever we need to hash.

> We define the function `dHash(data)` to mean `SHA256(SHA256(data))` in the rest of this description..

Cryptographic signatures

A cryptographic signature shows that the holder of some private key produced a message. Given a message, `data`, the signature is $E(K_{priv}, H(data))$. Someone receiving the `data` and the signature can verify that the holder of K_{priv} signed it by computing the hash of the `data`, then decrypting the signature using the public key. Those two values should be the same.

> We implement signatures ourselves; we do not use the signature methods available in most languages. Our signature of some byte string `data` is $E(K_{priv}, dHash(data))$.

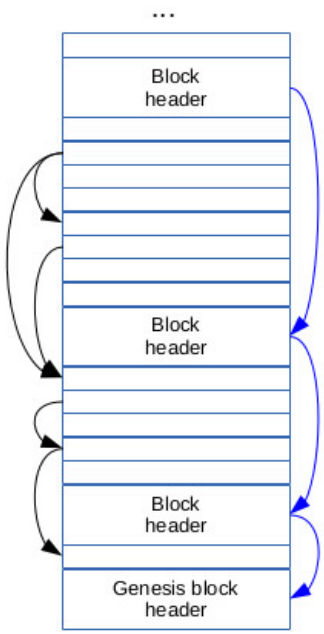
Bitcoin transactions and block chain

A Bitcoin transaction consists of references to a number of inputs and declarations of a number of outputs. The inputs name a set of earlier transactions in which Bitcoins were delivered to the entity creating this transaction. Those Bitcoins are being spent. The outputs indicate to whom those Bitcoins are being delivered. The sum of the input Bitcoins must be at least as large as the sum of the outputs.

In a typical case, there are two outputs. For example, Alice may create a transaction whose inputs refer to three outputs that delivered 2, 2, and 3 Bitcoins to Alice. She wants to send 6 Bitcoins to Bob. One of the outputs of her transactions gives those 6 Bitcoins to him. The other output gives one Bitcoin to Alice (her "change").

The Bitcoin transaction ledger is called a block chain. It consists of a sequence of blocks, each of which contains a block header, a 4-byte transaction count, and a vector of transactions. The first block is called the genesis block. It contains a single transaction that spontaneously creates a number of Bitcoins. Succeeding blocks after that contain sets of transactions, including a single *coinbase* transaction that has no inputs (and so generates new Bitcoins) that appears first in the block. (Only coinbase transactions are allowed to have no inputs.)

This image depicts a block chain. We show references from transactions spending Bitcoins to the transactions that awarded those Bitcoins for only a few sample transactions.



Each block header contains a reference to the previous block header, which is what makes this a block chain.

> The name of a transaction is the 32-byte string obtained by applying `dHash` to its complete binary representation. Similarly, the name of a block is the `dHash` of the complete binary representation of its header. The references shown as arrows in the figure above are simply copies of the names of the things being referenced.

Transaction format

A transaction is a vector of input specifiers and a vector of output specifiers. A two-byte unsigned integer gives the length of each vector.

#Inputs	<Input specifier vector>	#Outputs	<Output specifier vector>
2	var	2	var

> We follow the Bitcoins convention of representing integers in **little endian order** (not network order).

Input specifiers name a particular output specifier of some earlier transaction. Each input specifier looks like this:

Prev tx ref	Prev tx output index	Signature	Length(Public key)	Public key
32	2	128	2	var

The previous transaction reference is the name of the previous transaction, that is, its dHash value. The index field indicates which of that previous transaction's outputs is the input we're using. The signature field is the signature of the byte string representing this transaction. You form the signature using dHash to the entire transaction (not just this input specifier), except that you elide (completely leave out) the signature fields of all the input specifiers. (You only elide the signature fields when forming signatures.) You compute the signature using the private key that corresponds to the public key referred to in the previous transaction's output specifier. The remaining two fields provide that public key.

> We transmit public keys as PKCS1 format strings. For example:
> -----BEGIN RSA PUBLIC KEY-----
> MIGJAoGBALZt89/Z8YqoNQcljH4/oKbun+prJ/pEQ/CLGMXpCh44bRQNYVyjDn/Y
> 93dHAD4d2W8Xnno0lg7ovEY6eOm6+FU0qR8WBusaEM55/eCg9PUMncbqxxi2Azg7
> vNWFbYGnXpdYYTZWpMTEutuZss8RYbltURwXGe1RiF+gC+JJbw0BAgMBAAE=
> -----END RSA PUBLIC KEY-----

Each output specifier looks like this:

Value	Hashed public key
4	32

The value field is an unsigned integer indicating the number of Bitcoins to transfer. The recipient is a public key, which is represented as the dHash value of its PKCS1 format representation.

Block header format and mining

Blocks are composed of a block header and a vector of previously unverified transactions, arbitrarily chosen by the miner who creates the block. In our implementation, the first transaction in the block has no inputs, and is either the genesis transaction (in the genesis block) or a coinbase transaction (awarding payment, presumably to the miner) in all the other blocks. The miner must construct and insert the coinbase transaction. The block header looks like this:

Version number (1)	Previous block ref	Merkle root	Creation time	Difficulty (3)	Nonce
4	32	32	4	2	8

The version number is the integer 1. The previous block reference is the dHash of the binary representation of the immediately preceding block header. The Merkle root is, basically, a dHash of all transactions in the block, but done in a particular way. We'll discuss that in its own section. The creation time is a Unix time that is "pretty close" to when the block was created.

The nonce and difficulty fields are related to mining.

> To mine, you try to find a value for the nonce field so that the dHash of the block header has some number of high order 0's.

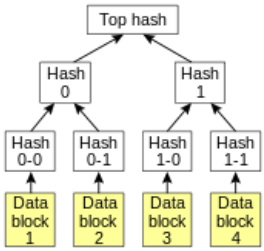
The difficulty field tells you how many leading 0's are required. In this project, the difficulty field is expressed in bytes - the value 3 means that the miner needs to find a nonce so that the dHash of the block header is 0x000000... (24-bits of leading 0's). Any nonce value that produces that will do. The only way to find one that works, though, is brute force. That is, you guess a value, assign it to the nonce field, evaluate the dHash of the header, and see if the result has enough leading 0's. How to guess is up to you.

We always use the value 3 for difficulty. The actual Bitcoins implementation dynamically adjusts the difficulty in response to changing amounts of computing power worldwide being devoted to mining, with the goal of achieving an average successful mining rate of 1 per 10 minutes. Computing single threaded (in perl!) on a PC with a 3.4GHz i7, I've experienced mining times at difficulty 3 of between about 20 seconds and 4 minutes for a single block. (I've never successfully mined a block at difficulty 4, as I give up after about 30 minutes. I believe real Bitcoins is currently operating at a difficulty level corresponding to about 60 bits of leading 0's. It's estimated that about 362,000,000,000 GigaFLOPs are currently devoted to mining.)

Merkle tree

A Merkle tree is a binary tree of hashes. The leaf nodes are hashes of data items. Each internal node is the hash of the value formed by concatenating the hashes of its two child nodes.

http://en.wikipedia.org/wiki/File:Hash_Tree.svg



A Bitcoins block header stores the hash from the root node of the Merkle tree formed by using the binary representations of all the transactions in that block as the data items. If any level of the tree has an odd number of elements, the value of the last node is concatenated with itself to use as input to the hash function when computing the parent node's value. That is, if there were only three data blocks in the figure above, the value of "Hash 1" would be computed by concatenating the value of "Hash 1-0" with itself, and then hashing that result.

As always, Bitcoins uses `dHash` as the hash function when computing the Merkle tree.

What to Do

We'll provide you with binary data that has, in order:

1. The genesis block
2. The genesis block transaction count (1)
3. The genesis transaction
4. A 4-byte transaction count
5. That many transactions

We make this data available in two ways:

- Via TCP. When you connect to `cse461.cs.washington.edu:46114` the data will immediately start streaming at you. There is no protocol - the data just starts arriving. Anything you might send on that connection will never be read. The connection will be closed by the sender when all the data has been sent.
- As a file: `/projects/instr/14wi/cse461/transactionData-10000-3.bin`.

Your job is to *verify* the transactions that follow the genesis block. That is, you must create one or more blocks that, together, contain all *valid* transactions, and for each block find a nonce that satisfies difficulty level 3. You need to write a binary file with the block chain you've formed, starting with the genesis block. You'll submit that binary file, and we'll verify that it correctly verifies the original transactions.

You should award yourself ten Bitcoins for each block you verify, by inserting a coinbase transaction as the first transaction in the block. Your identity is the public key:

```
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBAN3MxXHcbc1VNKTOgdm7W+i/dVnjv8vYGlbkdaTKzYgi8rQm126Sri87
702UBNzmkkZyKbRKL/Bfc4EG8/Mt9Pd2xQlRyXCL9FnIFWHyhfIQtW+oBsGI5UhG
I8B8MiPOMfb6d/PdK+vd4riUxHAvCkHW5Lw0szAD1RVGbkG/7qnzAgMBAAE=
-----END RSA PUBLIC KEY-----
```

There are no characters after the final hyphen in that string. The `dhash` of that key is 1f5a0200bc94ae4264642855786d9c2bb436b9e129ef95e6416136c03f339581.

You should also keep track of how many Bitcoins each public key has, and print that information when you've processed all the transactions. (Because the transaction outputs specify the `dHash` of the public key of the recipients, you should use the `dHash` of public keys to name the owners of Bitcoins.) Put that list of current Bitcoin balances in the file `balances.txt` for turn-in.

Valid Transactions

You should simply drop any transaction that isn't valid. This Bitcoins wiki article lists many criteria that a transaction must satisfy to be valid. We'll enforce only a few.

To be valid, a transaction must:

- have total output value not larger than its total input value
- have valid output references: the previous outputs used as inputs must exist; for each referenced output, the public key supplied in the input specifier must hash to the value given in that output; the signature given in the input specifier must have been produced by the private key corresponding to that public key.
- no previous output can have been already used as an input by any other transaction

We guarantee that when you go through the transactions all references are backwards in the stream -- that is, if a transaction's inputs aren't available when you encounter that transaction in the stream, they never will be. So, you don't have to worry about making multiple passes over the transaction stream (as you would in real Bitcoins).

We also guarantee that all transactions have at least one input, except for the genesis and coinbase transactions. Some transactions will award you a transaction fee, by having a total output value that is smaller than the sum of the input values, and some won't. We don't reject transactions that award no transaction fee.

The final "detail" is the signature that is contained in the input specifier. Its purpose is to prove that the transaction is created by the

owner of the public key specified in the previous output being used. It does that by signing something using the corresponding private key. Basically, what it signs is the entire transaction it is part of. (That prevents an adversary from reusing the signature in some other transaction, which might be possible if the signature were over any data other than the transaction itself.) But, there's a problem: the signature is part of the transaction it is trying to sign, so you can't sign until you know the signature.

To get around this, Bitcoins (and we) define the signature to be over the binary representation of the transaction but omitting the signatures contained in all the input specifiers.

Online verifier

We've installed an online verifier: http://courses.cs.washington.edu/courses/cse461/14wi/projects/proj4/verifier/verify.cgi . You can upload the binary file output of your program and it will, eventually, tell you something. Here's what it says if you upload the file we provided to you as input:

```
[2014-03-09 11:20:25 (0)] Looking for block 0
[2014-03-09 11:20:25 (0)] Read block 0
[2014-03-09 11:20:25 (0)]   Verifying block header
[2014-03-09 11:20:25 (0)]     Version number: match
[2014-03-09 11:20:25 (0)]     Predecessor block: match
[2014-03-09 11:20:25 (0)]     Merkle root: match
[2014-03-09 11:20:25 (0)]     Difficulty: match
[2014-03-09 11:20:25 (0)]     Nonce verification: match
[2014-03-09 11:20:25 (0)]     Tx count: match
[2014-03-09 11:20:25 (0)]     Block header verifies
[2014-03-09 11:20:25 (0)]   Verifying 1 transactions
[2014-03-09 11:20:25 (0)]   Done with transaction verification
[2014-03-09 11:20:25 (0)] Looking for block 1
[2014-03-09 11:20:25 (0)] Error: TCP::readBytes: reached EOF in middle of request for 28467 bytes
```

This means it managed to read block 0 (the genesis block), the block passed a sequence of verification tests, it managed to read the sole transaction in that block, and that transaction passed verification. After that, it tries to read block 1, which isn't in that file. It gets end of file in the middle of trying to read block 1, which is a fail.

Here's what it says when I send it the output of my assignment solution:

```
[2014-03-09 11:28:36 (0)] Looking for block 0
[2014-03-09 11:28:36 (0)] Read block 0
[2014-03-09 11:28:36 (0)]   Verifying block header
[2014-03-09 11:28:36 (0)]     Version number: match
[2014-03-09 11:28:36 (0)]     Predecessor block: match
[2014-03-09 11:28:36 (0)]     Merkle root: match
[2014-03-09 11:28:36 (0)]     Difficulty: match
[2014-03-09 11:28:36 (0)]     Nonce verification: match
[2014-03-09 11:28:37 (0)]     Tx count: match
[2014-03-09 11:28:37 (0)]     Block header verifies
[2014-03-09 11:28:37 (0)]   Verifying 1 transactions
[2014-03-09 11:28:37 (0)]   Done with transaction verification
[2014-03-09 11:28:37 (0)] Looking for block 1
[2014-03-09 11:28:40 (0)] Read block 1
[2014-03-09 11:28:40 (0)]   Verifying block header
[2014-03-09 11:28:40 (0)]     Version number: match
[2014-03-09 11:28:40 (0)]     Predecessor block: match
[2014-03-09 11:28:40 (0)]     Merkle root: match
[2014-03-09 11:28:40 (0)]     Difficulty: match
[2014-03-09 11:28:40 (0)]     Nonce verification: match
[2014-03-09 11:28:40 (0)]     Tx count: match
[2014-03-09 11:28:40 (0)]     Block header verifies
[2014-03-09 11:28:40 (0)]   Verifying 9985 transactions
[2014-03-09 11:28:41 (0)]     500 transactions processed
[2014-03-09 11:28:41 (0)]     1000 transactions processed
[2014-03-09 11:28:41 (0)]     1500 transactions processed
[2014-03-09 11:28:41 (0)]     2000 transactions processed
[2014-03-09 11:28:41 (0)]     2500 transactions processed
[2014-03-09 11:28:41 (0)]     3000 transactions processed
[2014-03-09 11:28:41 (0)]     3500 transactions processed
[2014-03-09 11:28:41 (0)]     4000 transactions processed
[2014-03-09 11:28:42 (0)]     4500 transactions processed
[2014-03-09 11:28:42 (0)]     5000 transactions processed
[2014-03-09 11:28:42 (0)]     5500 transactions processed
[2014-03-09 11:28:42 (0)]     6000 transactions processed
[2014-03-09 11:28:42 (0)]     6500 transactions processed
[2014-03-09 11:28:42 (0)]     7000 transactions processed
[2014-03-09 11:28:42 (0)]     7500 transactions processed
[2014-03-09 11:28:43 (0)]     8000 transactions processed
[2014-03-09 11:28:43 (0)]     8500 transactions processed
[2014-03-09 11:28:43 (0)]     9000 transactions processed
[2014-03-09 11:28:43 (0)]     9500 transactions processed
[2014-03-09 11:28:43 (0)]   Done with transaction verification
[2014-03-09 11:28:43 (0)] Looking for block 2
[2014-03-09 11:28:43 (0)] Reached EOF
[2014-03-09 11:28:43 (0)] All done
```

I don't have a convenient way to generate useful incorrect assignment solutions, so I'm not sure what the coverage of this verifier might be, or how it will react to unusual cases. It's intended to be useful, but comes with no guarantees.

Additional information

We mostly follow the Bitcoins specification, so it might be useful to look at Bitcoins documentation if anything here is confusing. (The major changes are to simplify Bitcoins by implementing only one choice in places where Bitcoins allows many choices.)

The Bitcoins wiki contains a technical articles section that provides lots of details.

Competitive mining

In real Bitcoins, mining is competitive: only the miner that first finds a valid nonce gets to collect the reward for that block. That online competition has motivated people to do things like build special hardware just for the purpose of mining.

This project has an optional, offline competitive element as well: we'll award to whichever team ends up with the most Bitcoins the right to name this project when next used in CSE 461. (Great, eh?) You can choose any name you want, subject to the highly excitable nature of state law.

The simplest scheme to satisfy the assignment requirements is to put all the transactions in a single block. That means you have to solve for only one nonce, so runs are relatively quick. However, you earn only one reward. The more blocks you solve, the more rewards you earn, but the longer it takes to run. I've sized the input so that producing one block per transaction (earning the maximum reward possible on the input) is estimated to take at least 3.5 CPU days. If you want to enter this competition, the trick is to (a) get a working version quickly, then (b) estimate about how many blocks you can verify in the time you have left and work towards that.

Workload

This project is far smaller than the previous one. The main difficulty is in understanding all the pieces. The major pieces are: computing the Merkle tree; finding a nonce that satisfies the difficulty criterion; verifying that a transaction is valid; converting between program representations of block headers and transactions and their binary representations. None of these is very big -- double digits lines of code, I'd guess. Hashing and encryption are just library calls. The code to produce a signature (as we do it) is only a couple of lines.

The project does not need to be multi-threaded.

Your `run` script

We're not likely to run your code, but if we do we'd like it be invokable like this:

```
$ ./run <hostname> <port> <in-filename> <out-filename>
```

That is, we'll supply you with the location of a TCP source for the data and a file source, and we'll specify the name of the binary file you should create. You're free to use either input source -- just ignore the specifiers for the source you don't use, but you must accept them as arguments because we're going to supply them. Write your binary output to the file we name. Print your balance ledger to `stdout`.

Turn in

When you're ready to turn in your assignment, do the following:

1. The files you submit should be placed in a directory named `proj4`. There should be no other files in that directory.
2. That directory should contain your project source, a `run` script, your `balances.txt` file, the binary file containing the block chain you generated, and a README.TXT file that contains the names, student numbers, and UW email addresses of the member(s) of your team.
3. Tar up the directory as `tar czf proj4.tar.gz proj4` and submit.

Appendix A: Example block header

Here are text and binary representations of the genesis block and transaction. They can be helpful to verify that your encoding of blocks and transactions agrees with that used in the distributed data.

```
Block: dHash (name) = 000000ad6f0e881c462f84ecb9dbfaa64ed4d81e6f638513dbe77d40e16c1122
   Previous block hash:  0000000000000000000000000000000000000000000000000000000000000000
   Merkle root:  b6023ace162dfdddbd0c663996f470c8f77abfce08f32102b96293bc9d0a2d09
   Creation time:  1393987703  [2014-03-04 18:48:23]
   Difficulty:  3
   Nonce:  3060210
   Header packedBytes:
010000000000000000000000000000000000000000000000000000000000000000000000b6023ace162dfdddbd0c66399
6f470c8f77abfce08f32102b96293bc9d0a2d097790165303 00f2b12e0000000000
   Transaction: dHash (name) = b6023ace162dfdddbd0c663996f470c8f77abfce08f32102b96293bc9d0a2d09
      nInputs: 0
      nOutputs: 1
      TxOut
         value: 100000
         dhashPublicKey: 2307375d0b54d125294bbbd855a9eaf2a4b26f88b4b339a502e14b5cf90eaebc
      Packed bytes:
00000100a08601002307375d0b54d125294bbbd855a9eaf2a4b26f88b4b339a502e14b5cf90eaebc
```

The "packed bytes" fields are hex representations of the binary encoding of the object. One is shown for the block header (which excludes the transaction count and transaction), the other for (just) the transaction.

Appendix B: Merkle root example computation

In this example our data is five ASCII strings. (In your code, the data is the binary representation of transactions.) We show the results as we compute up to the root. All those values are 256-bit binary strings, shown here in hex.

```
[ "a", "five", "word", "input", "example" ]

[ bf5d3affb73efd2ec6c36ad3112dd933efed63c4e1cbffcfa88e2759c144f2d8,
  785ec2cf1199779cb1593eac85d7465db3603834e9edd6d1420f2b369bb01dc3,
  9c16c96d992ee3c53299e9c497a4b1c16fd2b177214c5abb5dc958c57366a48b,
  b3c8485039102bd67a91e9ae23c9f8c7b4b7f9907f2866f22ad14f9894d5070f,
  f0a7447cc7c8ab136c4c253e224377ac108af790d55cd9a9dd372bf2a7a3e737 ]

[ b32e2584f780f0cb09e150721ceac6b94a7c7c672ae31109d245692aa5f2b92e,
  048f642abfda573feab46610eeaa73f5cb6bb718993b47456ded94cbd914260c,
  268279451b322c1acdd60833093eff05e3b8f8b705c2cef1b3a7e01321dae4c1 ]
```

```
[ 211e42a0c483cc5010d6fc35173ca4be34d1a485c033052b0e4c7b56d6ae8ef5,
  a067ab5cbb397fd0724a9508089573f0a51c0d33fa0a8f3afaf88be7194299f6 ]

[ 2f45af2d1caa2da0a03226923a5cf88d734e2e194e3a1638d807d4d03fc3c67e ]
```

Appendix C: Signature example

```
Public key:
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBANu2X9ijlIhDbaua5+x9BK/vrbntU6HQc1lO1RRCpRfK9DWhkzzJwIAB
Bm1NEWpTN4DhSv04qcbMpSzqSDYMxz9/x3lg6zmhRWwq5T7qa1hXDOB6ffhFpxV0
k1X5J0FC/YiVPg+8SgwUy5G9K4t9iPLVedoPddbYy07wpDrnPH1hAgMBAAE=
-----END RSA PUBLIC KEY-----

Private key:
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDbtl/Yo5SIQ22rmufsfQSv76257VOh0HNZTtUUQqUXyvQ1oZM8
ycCAAQZtTRFqUzeA4Ur9OKnGzKUs6kg2DMc/f8d5YOs5oUVsKuU+6mtYVwzgen34
RacVdJNV+SdBQv2IlT4PvEoMFMuRvSuLfYjy1XnaD3XW2MtO8KQ65zx9YQIDAQAB
AoGAeBtAVftGTR8fKroponvNPig1vffgygpbpCyWCtdLzK/jxBWpmYdothDZZJLG
vGr1YnzGM5rwJH7mpKEGDJX7rNVufTrcRIjquR2GFvhogNLr/I49XT2fehvgwjD1
7IxaQYU43wFazCyW5iKrdeAlVQ0luKJjawWofBYmRSHRWUkCQQDufDjcWYFMzJam
8CbCk6ZyM6jxcUOGfpzomHrK9NrCo/aryQ8Wuf0ka6IHaEJkX7CwSbGiRBfGtEex
HDdz+AofAkEA69kz5z1rhSOhDONTpZEdnI6tYThpnD1EQnHqnffhjCYUTzj6OnNs
BcaDnRz89QKFOaXR2V1hxPaeEvCd2lGIfwJAPFGvEAyTZ5lXgWG8a/psXvYyBN9g
9OORTENEy5CixBg0i76O0nC4Vj3i/XyhTkHlrrD0/NW8LcXrXCCG5g4WgQJAUqig
WUYcfeAb3MF7moZ+o1UaDP3RfdG3L7ZvLQgog48BBTcJ9Bxp2qhVjmYPfetxN+AW
6SCiWH66rhaorFBxDwJBAMoGgm9cxUCYsl2FQugr+wL0Kx8ECI5727TEzCmuVFUx
X+kXCiTBgiO0WtTnd/uQmtqcLi3w19ko2her4Ctm8/k=
-----END RSA PRIVATE KEY-----

sign("hello world") =
8c079f46bf5d0f3dc26991993b48e121c4af991b5078f96ddb8c95a9f00c0c2236f2d9001ebda3d629ee64729d1b29ec8
027238ce4f37b5e55535068a4e86f88a01823ff5545b144de159648af60c6cd0cbe9e8daa16e099eaa174e9b43798065b
76e4ce200b46b65a70eb91991006a59fb4ea6c285e937d4f9625f366c4b3c1
```

Appendix D: Binary file format

If you don't already have a favorite tool for looking at the contents of binary files, you could consider using `hexdump`. (I use `emacs`.)

Here is a bit of the data file. The highlighted portion is the first public key in the file, presumably there as part of the genesis transaction's output specifier.