Notes for Bitcoin Developer workshop

1. What we're going to talk about.

It is important to understand that Bitcoin, developed in 2009, is not a standalone invention. Many different technologies underpin the Bitcoin system. Many of these foundational software constructs have roots dating back 30 years or more. These inventions span a number of disciplines, including cryptography, data structure design, networking protocols, and more.

This workshop will start by discussing some of these technologies first, before delving into the actual internals of the Bitcoin system so that the students can gain some understanding of the surrounding principles of the Bitcoin architecture, before actually analyzing the Bitcoin system itself.

The technologies covered will consist of the following:

1. Hash functions.
2. Cryptographic hashing.
3. Balanced Binary trees, with particular respect to Merkle Trees.
4. Public Key Infrastructure and basic encryption systems.
5. Elliptic Curve encryption.

2. Hash functions

What is a hash function? Simply put, a hash function allows a program to take any arbitrary data and transform it into a much shorter fixed-length value. This value is usually called the *hash key.*

Why is this interesting? Because it allows us to access large amounts of data very quickly by using the hash key rather than having to search through the data itself for the specific things that you're looking for. It also has another purpose, which is especially interesting for Bitcoin. It allows us to encrypt and decrypt digital signatures. This provides for the ability to authenticate both message senders and receivers, which is required for Bitcoin transactions. A specific subset of hash functions called *Cryptographic hash functions* are used for this purpose. They are called this because it is effectively impossible to obtain the original data from the digest alone.

Cryptographic hash functions have four main properties.

1. It's relatively easy to compute a digest using a cryptographic hash algorithm.
2. It's nearly impossible to re-create the original data by using the digest as input.
3. It's nearly impossible to change the message without changing the digest.
4. It's extremely unlikely that two messages will create the same digest.

What sort of hash functions are available? Many different hashing functions have been created for use. Some of the more popular functions are as follows:

1. Message Digest 5 (MD5). MD5 is a hashing algorithm that can take as input any arbitrarily large amount of data and outputs a 128 bit hexadecimal key called *message digest*. Here is an example in pseudocode of an MD5 hash output.

Here is an example of passing as the input a 43 byte ASCII string and its MD5 output.

MD5(“The quick brown fox jumps over the lazy dog”) =
9e107d9d372bb6826bd81d3542a419d6

Note that adding something as trivial as a period to the end of the string produces a completely different output.

MD5(“The quick brown fox jumps over the lazy dog.”) =

e4d909c290d0fb1ca068ffaddf22cbd0

2. Secure Hash Algorithm (SHA). SHA is a group of cryptographic hash functions that were published by the U.S. National Institute of Standards and Technology (NIST).

Some of these algorithms include:

a. SHA-1. SHA-1 generates a 160 bit digest for any message. This algorithm is no longer considered “computationally safe” and is not recommended for any applications.

b. SHA-2. SHA-2 consists of two functions, SHA-256 and SHA-512. The main difference is that SHA-256 uses 32 bit words, whereas SHA-512 uses 64 bit words. The Bitcoin protocol uses SHA-256 internally to generate digests.

3. RACE Integrity Primitives Evaluation Message Digest (RIPEMD) are a family of cryptographic hashes designed in Europe by academic researchers. Whereas the SHA hashes were designed by the U.S. National Security Agency, RIPEMD were designed by the open academic community. The most commonly used RIPEMD algorithm is RIPEMD-160 which generates a 160 bit digest. This algorithm is also used by the Bitcoin protocol.

3. A primer on asymmetric public key infrastructure.

There are many instances when two people will want to exchange information in a secure fashion. Examples of this include financial transactions, where one party wants to keep financial information, such as credit card numbers, secret from the general public and only be available to the message receiver.

There are two fundamental problems when attempting to exchange data in a secure fashion between two people.

1. How do you keep your data private from malicious snoopers?

2. How do you ensure that the person sending you the data is not a third party masquerating as the sender?

Originally, the concept of *symmetric key* encryption was developed. The idea was that both the sender and receiver of a message had the same key. The sender would encrypt the data with that key, and the receiver would be able to decrypt that data with the same key. There is, however, a serious weakness in this strategy. If a malicious third party obtains this key by some method, then the encryption scheme becomes useless as the third party is now able to decrypt all the data being sent between the sender and the receiver.

Because of this problem, a new strategy, *asymmetric public key infrastructure* was developed.

In this strategy each user generates two keys, the *private key* and the *public key.* The public key is made available to anyone who wishes it. The private key is kept (in theory) in a secure location accessible only to the user. Because there is a mathematical relationship between a users private key and their public key, it is possible to use the private key to decrypt a message that has been encrypted with their public key. When a person (whom we'll call Alan) wants to send a message to another person (whom we'll call Bill), the following steps will be taken.

1. Alan obtains Bill's public key.

2. Alan encrypts the message using Bill's public key.

3. Alan sends the encrypted message to Bill.

4. Bill takes the encrypted message and applies his own private key to decrypt it.

Note that this prevents a third party from decrypting the message as he does not have access (again, in theory) to Bill's private key.

Also, note that while you can encrypt the message with the public key, decrypting it again requires the private key. This is an example of a *one-way* function.

When creating a private key, it is important to understand that this key is merely a random number. Generally private keys are a random number containing 256 characters. Bitcoin generally uses the random number generator function provided by the underlying operating system. Interestingly enough, the Android operating system had a flaw in it's underlying random number generator that would generate the same number multiple times, causing some people to lose money from their wallets when crackers exploited this weakness to empty users bitcoin wallets of their funds.

There are many examples of asymmetric public key encryption algorithms. We'll now look at the one used with Bitcoin, *Elliptic Curve Digital Signature Algorithm (ECDSA).*

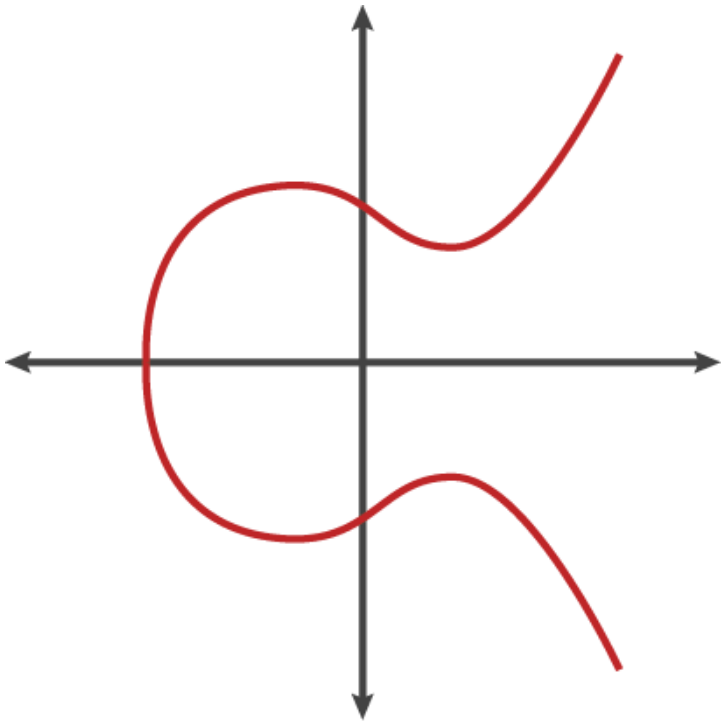
4. Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm was designed by the U.S. National Security Agency to provide a strong method to create a secure public and private key infrastructure for the internet (At least, that's what they claim).

The general format of an elliptic curve function is: $y^2 = x^3 + ax + b$

Here is an example of a specific elliptic curve that satisfies the function

$y^2 = x^3 + 7$ where a=0.

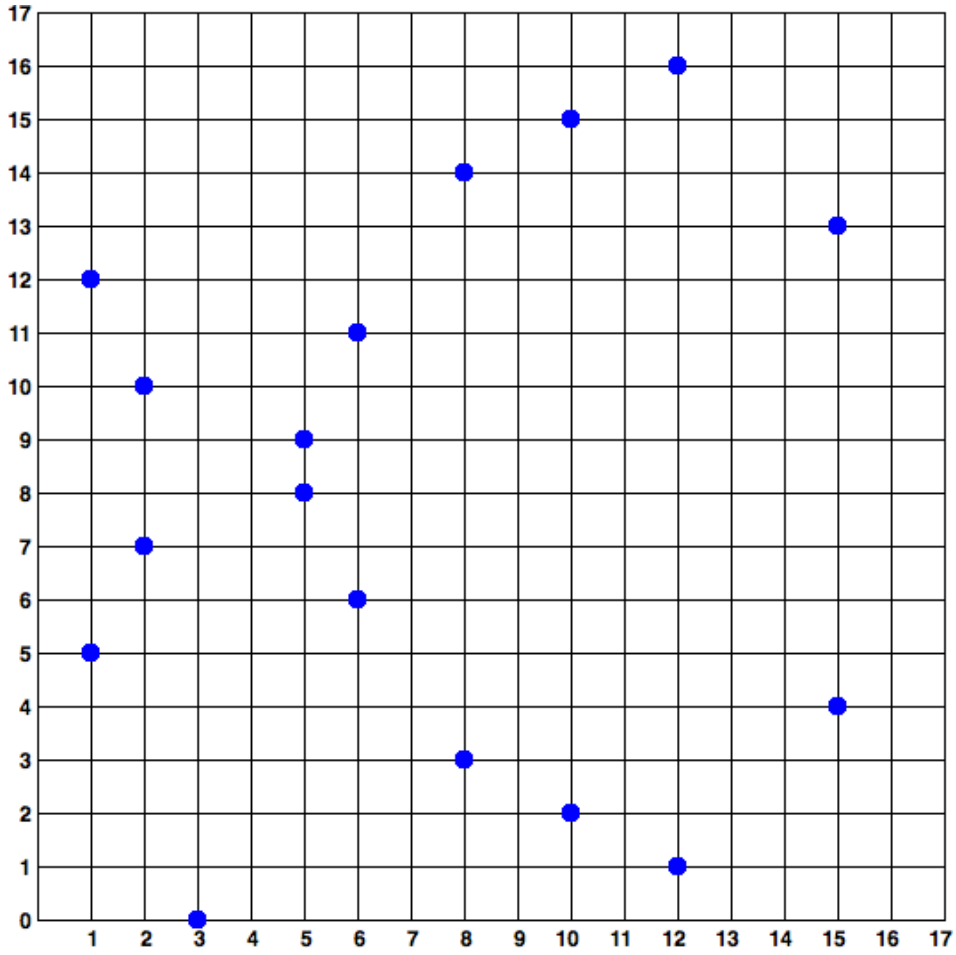


This particular elliptic curve function is known as *secp256k1.* Note that the graph above shows a mapping of the function to the real number set, whereas in reality, the function, when used, is actually mapped to a finite field of numbers of size F_p where

F_p = $2^{256}-2^{32}-2^9-2^8-2^7-2^6=2^4-1$

F_p is actually an extremely large prime number.

Here is an example of a a plot of secp256k1 mapped over a very small subset of this field with F_p =17 (again, a prime number).



Even though the graph shows discrete points rather than a smooth curve, you can still see the geometry of the curve in the plot. The mathematics are the same whether it is over the real number set or a field of prime order.

Note that we can draw a line between any two points on the graph and we find that it will intersect the graph in a third location. If we reflect that point around the X axis, we will get the *sum* of the coordinates of the first two points.

The three points (A,B, and C) are defined as follows:

A. The *generator* point. This is a constant number that is defined in the Bitcoin protocol.

B. A random 256 bit number multiplied by the generator point i.e the private key.

C. The *point of infinity* which is the sum of A and B. Note that once the sum is defined, then multiplication is just summing multiple times.

We can now calculate a public key P_k by multiplying the private key

by the generator point. $P^k = p_k * G$ Because of the concept of *prime factorization*, it turns out that we can generate the public key from the private key, but we can't realistically find the private key from the public key.

5. Base 58 and Base 58 check encoding

What is Base <x> encoding? There are many times when an application (or user) wants to transmit raw binary data across the network. This is generally a bad idea, however. Many applications will translate raw binary into control codes which have meaning to the application itself. Also, many protocols and media are designed to deal with textual data and don't properly process raw binary data. Encoding in a mechanism like Base64 allows binary data to be represented as text for easy transportation across the network.

Base 64 uses a set of 64 characters (A-Z, a-z,0-9,+ and /) to represent binary data in a text format.

The encoding process for Base 64 is as follows:

1. Divide the input stream into blocks of three bytes (8 bits/byte * 3 bytes = 24 bits).
2. Divide the 24 bits of each three byte block into four groups of six bits.
3. Map each group of six bits to a printable character in the Base 64 set as defined above.
4. If the last three byte block has < three bytes, pad with n bytes of zeros where 0 < n < 2.
5. Overwrite any final zeroes with the '=' so that the decoding process knows that these are padded zeroes.

As an example, Let's encode the following binary 0100000101000010 into Base 64.

Input data		
Input Bits	01000001 01000010	
Padding	01000001 01000010 00000000	

Note that we need three blocks of eight bytes, so we create a trailing byte with all zeroes.

Bit Groups	010000	010100	001000	000000	
Decimal conversion	16	20	8	0	
Base 64 Mapping	Q	U	I	A	
Overriding	Q	U	I	=	

However, Bitcoin doesn't use Base 64 encoding for converting binary to text. It uses a related system called *Base58 Check*. Why use a bespoke system like this? From the comments in the Bitcoin core...

// Why base-58 instead of standard base-64 encoding?

// - Don't want 0OIl characters that look the same in some fonts and

// could be used to create visually identical looking account numbers.

// A string with non-alphanumeric characters is not as easily accepted as an account number.

// - E-mail usually won't line-break if there's no punctuation to break at.

// - Doubleclicking selects the whole number as one word if it's all alphanumeric.

Base58 Check encoding is used to change a bitcoin address, usually output from SHA-256 and/or RIPEMD-160 into a text format that can be easily shared with anyone who wants to send you money.

The system is called Base58 *Check* rather than just Base58 because it also implements a checksum to allow the system to perform error-checking on the bitcoin address. Without this error-checking it would be possible to send an address that has been corrupted, but still seems to be a valid address, which means that any BTC sent to that address are effectively lost.

Here's how a Bitcoin address gets generated with Base58 Check.

We define the following:

1. The payload. In this case the payload is a public key generated from the secp256k1 ECDSA algorithm.

2. A prefix byte. For a bitcoin wallet, the designated prefix is 0x0.

We start the process by concatenating the prefix byte with the payload.

When then run this through SHA-256 *twice.*

Take the resulting output and append the first four bytes (32 bits) to the end of the prefix byte + payload. These four bytes are the checksum.

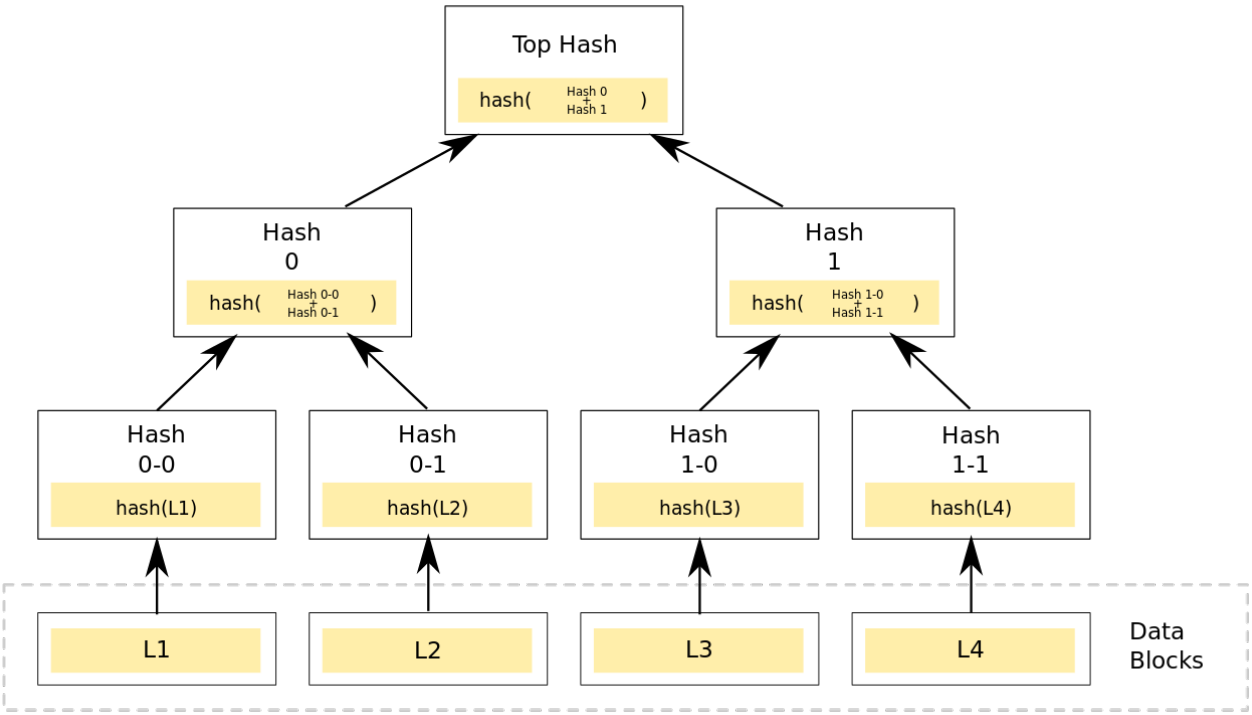
We now have data of the format “prefix byte + payload + checksum”.

We then run the data through the Base 58 encoder.

6. Merkle Trees.

Merkle trees are a type of a data structure called a *hash tree*. A hash tree is a specific implementation of a *balanced binary* search tree. Merkle trees are commonly used by peer to peer network applications such as Bittorrent to verify that data received by any arbitrary node is valid. The concept here is that in this tree, the leaves of the tree represent the data blocks. Subsequent nodes leading to the root are hashes of their children. The top of the tree is the root hash (Also known as the Merkle root in the case of Merkle trees).

Why is this interesting for us? We know that the stored blockchain of a full bitcoin node stores 10's of gigabytes of data. This makes it unfeasable for everyone to run a full node to be access and verify every transaction on the Bitcoin network. This has led to the creation of “Bitcoin-lite” nodes which only accesses transactions that matter to it, rather than verifying every transaction. This is where the Merkle tree becomes useful. With a Merkle tree, instead of having to go back through every transaction and it's ancestors to verify the transaction, it can simply calculate the hashes of a small number of transactions and verify it against the Merkle root which is present in every block header of the blockchain. Here is an example of a Merkle Tree.



Note that in the diagram, the leaves represent the hash of the datablocks, i.e. hash(0-0), hash (0-1) and so forth.. Each parent is then a combination of the hash of their children. This means that in order to verify data block 0, we need the following from a peer node:

hash 0-1

hash 1-1

The rest can be calculated by the local node. The local node then checks its calculation against the merkle root in the block header (Remember, the “lite” node only gets the block header information for each block rather than the entire block of data itself). Assuming that the calculated root matches the Merkle root in the header, then the data block is confirmed.

7. Bloom Filters

A bloom filter is a data structure that is designed to tell you rapidly whether an element is present in a set. To do this a Bloom filter is known as a *probabilistic data structure*. It will tell you that an element is either definitely not in the set or that it may be in the set. The basic structure of a bloom filter is a *bit vector*.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Each empty cell represents a bit, and the number above it represents its index in the vector. To add an element to the Bloom filter, we simply hash it a few times and set the bit s in the bit vector at the index of those hashes to 1.

For example to add the string “foo” to a sample bloom filter, we can hash the string with two simple hash functions, in this case *fmv* and *murmur*. The output of the fmv hash is “13” and the output of the murmur hash is “8”. Therefore the bloom filter would set the bits at index 8 and 13 to the value of 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	1	0	0	1	0	0	0

To test whether the string is in the filter, you simply hash the string you want to test with the hash functions specified and then see if the bit vectors are set. If not, then you know that the string isn't in the filter. Otherwise there is a high probability that the string is in the filter. Note that this isn't definite because other strings might also have set that bit vector.

It is strongly suggested that non-cryptographic hashes be used for the bloom filter hash functions, mainly due to performance issues. Examples of appropriate hash algorithms include:

1. fmv
2. murmur
3. Jenkins hashes

8. Wallets

What is a bitcoin wallet? This term is misleading when applied to bitcoin as it it implies that it is a container that stores a token of value (such as a physical wallet for fiat currency which contains paper bills). With bitcoin, the wallet stores the users *private keys*. No actual value tokens are associated with the wallet directly. Wallets are usually stored on a device (such as a laptop or a cell phone) as a simple structured file or a small database. Because the wallet contains the private keys, public keys can be generated from this. It is this private/public key combination that enables users to spend their bitcoin. Also, for increased anonymity, it is recommended to use new bitcoin addresses for each bitcoin transaction.

Wallet Types

As the bitcoin protocol has developed, different types of wallets have been invented for users. The first type, known as a *Type-0 non-deterministic* wallet, is simply a file containing a number of randomly generated private keys. The Bitcoin core client contains this type of wallet, which initially generates 100 private keys at random and creates more keys as necessary. It is not recommended to use this type of wallet, as it is difficult to manage. This type of wallet requires frequent backups if address re-use is not desired.

A more advanced wallet type is the d*eterministic* wallet. With this type of wallet, all the private keys are generated from a *seed*. A seed is a randomly generated number that, when combined with other data, can generate different private keys. This means that only the seed needs to be backed up. All other data can be then re-created from the backed up seed. Also, unlike the Type-0 wallet, it is easy to migrate to different wallet vendors as only the seed needs to be moved.

There is a Bitcoin Improvement Proposal (BIP 0039) to allow creation of seeds through a concept of *mnemonic phrases*. This translate the seed into human readable words that are easier for users to transcribe.

The most advanced wallet type is the *Heirarchical Deterministic* (HD) wallet. This type of wallet contains a heirarchy of private keys starting from a master key and implementing a tree of children (and grandchildren and so forth) keys.

HD wallets may be primarily used by entities and people who are transacting large volumes of Bitcoin and want to have multiple bitcoin addresses for organizational purposes. A good example of this would be a web site that wanted to generate new bitcoin addresses for every user when they are filling a shopping cart. Without this concept of HD addresses, the web site would have to generate new, unrelated, keys for every transaction which would quickly become unfeasible to manage.

9. Transactions

Transactions are the heart of Bitcoin. A transaction is the mechanism by which people can send you money, and you can send money to others. Everything else in the Bitcoin system is designed to allow transactions to be created, sent over the network, validated and recorded in the bitcoin ledger, also known as the blockchain. To begin with, we shall look at the concept of a bitcoin address. This is the part of the transaction that designates the sender and recipient.

A bitcoin address is simply a string of letters and numbers beginning with the number '1'. We generate this address by doing the following.

1. $A = RIPEMD160(SHA256(P_k))$ which simply means that we take the users public key, hash it with the SHA-256 hashing algorithm, and then hash it again with the RIPEMD160 algorithm. This double hashing is presumably for extra security.

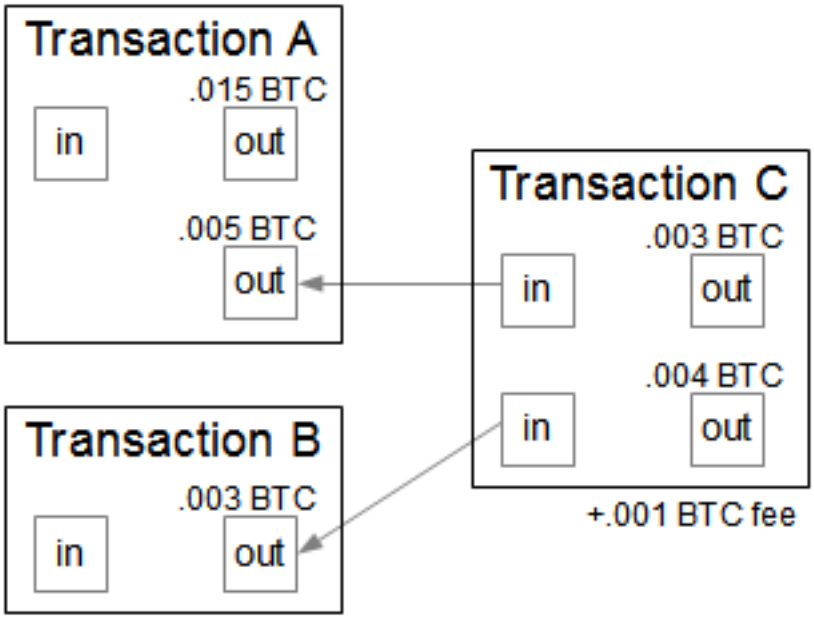
2. Prepend an '0x0 to the address and then run it through the Base58 check algorithm. Note that '0x0' converted into a Base58 character is '1'.

For our labs, we'll be using the bitcoin testnet (unless you're not really fond of any bitcoins you currently possess). In this instance, the bitcoin address begins with an 'm' or an 'n' which is converted from the value '0x6f'.

The Transaction Lifecycle

We start by creating a transaction. A transaction is more than just moving bitcoins from one address to another. What really happens is that the system moves bitcoins between one or more *inputs* and *outputs*. Each input is a data structure that contains a transaction and an address that supplies the bitcoin(s). Each output is a data structure containing a bitcoin address and the amount of bitcoins going to the address.

The following diagram shows an example of this.



What does 'unconfirmed' mean?

In this diagram Transaction C has two inputs, .005 BTC from Transaction A and .003 BTC from Transaction B. Note that the outputs for Transaction C are .004 BTC from Transaction A and .003 BTC from Transaction B. The extra .001 BTC is a fee that is paid to the miner to include this transaction in a newly generated block. Also, don't be confused by the direction of the arrows. The arrows are references to the previous outputs, so they point backward from the flow of the BTC.

Each input must be completely used when sending to an output. So, for example if you have an input of 100 BTC and you want to send 1 BTC to a new output, you would create another transaction that would send the output back to yourself.

While transaction fees are optional, be aware that miners may not place a high priority on a transactions confirmation and it may be hours or even days before a transaction without any attached fees is confirmed.

The structure of a transaction

Size	Field	Description
4 bytes	Version	Currently this field is set to the value of '1'
1-9 bytes	Input counter	How many inputs to this transaction are included
Variable	Inputs	Data structures containing the transaction inputs
1-9 bytes	Output counter	How many outputs to this transaction are included
Variable	Outputs	Data structures containing the transaction outputs
4 bytes	Locktime	Indicates when the transaction is supposed to be executed. A '0' indicates immediate execution. Other possibilities included a UNIX timestamp or a block number.

Here is an example from blockexplorer.com of the raw transaction in JSON format.


```
{
  "hash":"09ec1100c0c1dbbc33992815ccef458b2ac98697c3892d34a28066da3704c523",
  "ver":1,
  "vin_sz":2,
  "vout_sz":1,
  "lock_time":0,
  "size":339,
  "in":[
    {
      "prev_out":{

"hash":"a1abcb3cb222cac5eef308a82ede99a0b8ed03fbd8329256551db9b95fab21a2",
        "n":56
      },

"scriptSig":"304502210087d433cafa1a578f0481fc20aea887f400579980c73ad6518a600ac
4f772d05f02200b0c9888d981279542e1c924cd5e53fb8268891be7ac9c72a1797a3e20b58
0bc01 03b7c4ea1c28744c25e8106b725079b3055b9390311860bcb3b57428ec273ba93c"
    },
    {
      "prev_out":{

"hash":"cd71286a2af02335764f1f5c5dabc6d12784f827ea8dbaaf383ad6c89d35b025",
        "n":1
      },

"scriptSig":"3044022042e924f1ec93a77949f1d6d056e4ebbd38b5b1e7bcc66d08d395ccc0
c69330240220324ea0882e238161e5d5ab6b1216d3b5d7a70d5720158696bc1d7b61c958
7a2901
03b7c4ea1c28744c25e8106b725079b3055b9390311860bcb3b57428ec273ba93c"
```

```
    }
  ],
  "out":[
    {
      "value":"0.00158830",
      "scriptPubKey":"OP_DUP OP_HASH160
81580ad53d64229c60189f82c870a6305a78bf71 OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
```


Unspent Transaction Outputs

The basic unit of a transaction is an *Unspent Transaction Output* (UTXO). It is important to understand that there is no concept of an “account” or a “balance” tied to a bitcoin address. All the bitcoin that you own are recorded as UTXO's scattered across the blockchain. When a wallet displays your “balance”, what it is actually doing is scanning the blockchain for all of the UTXO's that are tied to your bitcoin address. Once a UTXO is recorded on the blockchain it can't be divided. If you wish to spend btc in a UTXO, you must do two transactions, the amount to spend and another transaction to take the remainder of the BTC and create a new UTXO. Note that if you are creating your own transactions and you fail to direct the “change” from the transaction to another UTXO, the change will be allocated to the miner's fee. This is probably not what you want.

The transaction inputs are the UTXO consumed by the transaction. The transaction outputs are the UTXO created by the transaction.

There is one exception to this rule, the *coinbase* transaction. The coinbase transaction is created when a miner successfully mines a new block on the blockchain. Since new BTC are created, there are no input transactions, only an output transaction.

Output transaction structure

UTXO's are stored in a database held in memory called the *UTXO memory pool*. These transactions are contained in a data structure with the following format:

1. A BTC amount denominated in Satoshis.
2. The size of the locking script in bytes.
2. The locking script itself.

Here is a JSON fragment that shows the structure of the output transaction


```
"out":[
   {
     "value":"0.16700000",
     "scriptPubKey":"OP_DUP OP_HASH160
b5f9d7aaf8c5be4241e343836daa1f2404ff417b OP_EQUALVERIFY OP_CHECKSIG"
   },
   {
     "value":"0.00230817",
     "scriptPubKey":"OP_DUP OP_HASH160
e42f5bde4fb8fc974bbd1eb50a8e7a33acbd35f1 OP_EQUALVERIFY OP_CHECKSIG"
   }
 ]
}
```


This fragment doesn't include the size of the script in bytes.

There is currently a debate that touches upon the current state of the memory pool size on a full bitcoin node. There is a worry that as the size of pool grows, assuming increased usage of the bitcoin network, it will become economically unfeasible to run systems with enough memory to store the entire list.

Input transaction structure

The structure of an input transaction is as follows:

Size	Field	Description
32bytes	Transaction Hash	Pointer to the previous output transaction containing the UXTO to be spend
4 bytes	Output index	Index number of the output transaction to be spent, starts at zero.
1-9 bytes	Unlocking script size	The length of the unlocking script, in bytes.
Variable	Unlocking script	The unlocking script
4 bytes	Sequence number	Disabled.

Here is an example of the input transaction structure in JSON format.

"in":[

{

"prev_out":{

"hash":"dbb49f1cfa1cd6f9f5f818dece0c395ec5f551c54564eed2ac2fe53b2b5c970b",

"n":0

},

"scriptSig":"3045022100ad2350b8c99c3e087fa386b85fdaf36c2a36278a87f4a4596502647990728ca60220037ddf960465294c8ac6f85f29f7352ed11386a641f469a6dcff441d8707792301 03267dac36155b720d92835c6c56a893c415f80ba68f04c76ed7ff47ebe39cf0a8"

},

{

"prev_out":{

"hash":"d844906bb27bfc1ec15786654ebbe75c01eed87b97072fec7bd61491cd066d0a",

"n":1

},

"scriptSig":"304502205678aefab6d1024ef8675fe69e63668a1091e36c9ef6cdb1072a5b1f543badf3022100ded729060011a35874c4e421028dedfa5f4449528bc4890329bd79fb08c0a7c201 03b4a45727c6f816d2bf41f37a97a521e6c99e5dd14c8d1ee72135b02d0c4f6ed6"

}

],

Bitcoin scripts

It is important to understand that Bitcoins are not just digital money, they are *programmable* money. Bitcoin uses a scripting langague (usually just called *script* or *Bitcoin script)* to facilitate this. Script is a *stack*-based language similar to Forth. Stack based languages use something called *Reverse Polish Notation* (RPN) for placing operators and operations on the stack. So, for example:

3 + 4 using Reverse Polish Notation would be:

3 4 +

And the output would be seven.

Reverse Polish Notation allows us to define away ambiguity like so.

3 + 4 * 5 can be ambiguous. Do we want (3 + 4) * 5? or 3 + 4 * 5? The rules of arithmetic say that the output of these two expressions are quite different (i.e. 35 vs. 23).

with RPN, we can do this:

3 4 + 5 * which gives us the first expression

or,

3 4 5 * + which gives us the second expression

Now let's do a similar operation with bitcoin script.

3 4 OP_ADD 7 OP_TRUE

This script pushes the 3 and then the 4 onto the stack and adds them up. It then pushes 7 on to the stack and tests whether the results of the OP_ADD are equal to 7. If they are equal, then a TRUE value is returned, otherwise a FALSE (0) value is returned.

As mentioned previously, the Bitcoin protocol supports two types of scripts, the locking script (scriptPubKey) and the unlocking script (scriptSig). The output transaction supplies the locking script, that is, only if the resulting value of the output is TRUE can the output be spent. The input transaction supplies the data that is acted upon by the output script.

A couple of notes on Bitcoin script.

1. Bitcoin script is *Turing incomplete*. That is to say, it is a limited language that doesn't support looping constructs or complicated flow control. This is specifically designed so that a malicious entity can't write a program that can run forever and/or consume resources on bitcoin nodes.

2. Bitcoin script is *stateless*. There is no state prior to or after the execution of the script. All inputs and outputs are contained within the script itself. This means that the script will behave in a completely predictable way on every node in the bitcoin network. This ensure

that properly formed and authorized transactions will be validated in the same way on all bitcoin nodes.

At this time, only five types of Bitcoin scripts are supported. It is expected that in the future this restriction will be lifted to allow programmatic scripts of most any kind. Some of the scripts supported are:

Pay to public key hash (P2PKH)

The overwhelming majority of transactions are of this type. The output (locking) script looks like this.

OP_DUP OP_HASH160 <recipients public key hash> OP_EQUAL OP_CHECKSIG

The input (unlocking) script would look like this.

<Recipients digital signature> <Recipients public key>

The final script (unlocking + locking) looks like this:

<Recipients digital signature> <Recipients public key> OP_DUP OP_HASH160 <recipients public key hash> OP_EQUALVERIFY OP_CHECKSIG

So what does the execution of this script look like:

Step 1. The recipients digital signature is pushed onto the stack

Step 2. The recipients public key is pushed onto the stack

Step 3. The recipients public key is duplicated onto the stack.

The stack now looks like this...

Recipients Public Key

Recipients Public Key

Digital Signature

Step 4. Execute OP_HASH160. Hashes the most recent copy of the Recipients public key.

Step 5. Push the recipients public key hash onto the stack.

The stack now looks like this:

Recipients public key hash

Recipients public key hash

Recipients public key

Recipients digital signature

Step 6. OP_EQUALVERIFY checks to see that both of the Recipients public key hashes match. If so, both public key hashes are removed from the stack, otherwise a FALSE value is returned

Step 7. OP_CHECKSIG checks that the recipients public key matches the digital signature. If they match, then a final TRUE value is returned and the output(s) can now be spent.

MULTISIG

This type of transaction specifies that the outputs can only be spent if some number of signatures are supplied as part of the locking script. Currently a maximum of 15 signatures can be passed in the locking script. A multisig script looks like this:

2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG

This script sets a 2 out of 3 key condition, where at least two out of the three keys must be present in the unlocking script for the output to be consumed.

The unlocking script might look like this:

OP_0 <Signature A> <Signature C>

Note that OP_0 is the mull statement. This is needed because of a bug in CHECKMULTISIG where it popped one too many arguments off of the stack. Pushing OP_0 onto the stack first is the workaround.

Pay to script hash (P2SH)

While the multisig scripting function is quite useful, it has its own set of problems. One issue is that the script can get very large with multiple public keys as part of the script. This can get somewhat expensive given that miner fees are based on the size of the transaction, not its value. It also makes it more complicated for customers to be able to send money to the merchant if they have to worry about how the locking script.

With P2SH, a third script element is introduced, the *redeeming* script. When using P2SH, the receiver creates a special bitcoin wallet with a prefix of '5'. This turns into a Base58 check value of '3'. The bitcoin addess now contains the following:

Prefix value of 3

20 byte hashed value of the redeem script

This is the address that is used by customers when they send BTC to the merchant. This address can be used by any standard wallet. Therefore there is no specific burden on the sender in terms of complexity or fees.

The recipient then provides the transaction with the redeeming script as part of the unlocking script which is then run on the stack. Assuming that a TRUE value is the final output of the redeem script, the recipient can then consume the BTC in the transaction.

When P2SH was first implemented in Bitcoin Core the redeem script was limited to a standard script such as P2PKH or Multisig. However as of version 0.92, the redeem script can be any valid script.

Warning. If you provide a hash of a script that turns out to be invalid, any bitcoin sent to that output will be lost as the script will never return a TRUE value.

OP_Return

The OP_Return op code provides the ability to store non-bitcoin related data inside the blockchain. This allows for applications to use the Bitcoin blockchain for non-bitcoin related use. The locking script looks like:

OP_Return <data> where data can be up to 40 bytes long. Generally, this data is the output of a SHA256 hash, although it can be anything at all. For example, here is a partial list of transaction outputs for a specific transaction.


```
"vout" : [
     {
          "value" : 0.00000000,
          "n" : 0,
```

```
            "scriptPubKey" : {
                  "asm" : "OP_RETURN 636861726c6579206c6f766573206865696469",
                  "hex" : "6a13636861726c6579206c6f766573206865696469",
                  "type" : "nulldata"
            }
      },
```

Note that the “asm” portion shows the following.

The Op code (OP_RETURN).

The size of the message in bytes (0x13, which is decimal 19)

and the message itself “charley loves heidi”.

The concept of OP_Return and the ability to store arbitrary data on the blockchain itself may, in the end, be a bigger market than the bitcoin currency itself for an example, see the Open Assets Protocol specification for Colored Coins.

Decentralized Consensus

The real power of Bitcoin is the concept of *Decentralized Consensus.* This means that all transactions can be verified without having to rely on a central authority (which can be subverted or shut down) to do so. Decentralized Consensus is enabled by the following properties:

1. Each transaction is verified independently by each node on the Bitcoin network based on a list of criteria. Note that this list can be modified in response to any new potential exploits discovered.

2. Each transaction can be aggregated into a block by independent mining nodes, where each block is verified using a computation proof of work algorithm.

3. Each block is verified into the blockchain independently by every full node on the network.

4. Independent verification of the correct blockchain by independent nodes, based on an highest amount of work generated algorithm.

Transaction Validation

When a block is mined, all of the transactions (minus the coinbase transaction) are validated by every full node on the network. This ensure that invalid transactions will not be propagated across the network. The validation checklist is somewhat long and convoluted and won't be listed here.

Aggregating transactions into a block.

Once transactions have been validated, special nodes called *mining nodes,* will aggregate these transactions into a candidate block. Note that at this time, the candidate block is not yet part of the blockchain (and may never be). The mining node will then attempt to solve the proof of work puzzle. If successful, the candidate block will be propagated out onto the network and be added to the blockchain if verified by all other nodes. The decision on which transactions to add to the block are based on the following formula:

$$Priority = \sum (V * I)/T$$

Where the sum is calculated over all of the inputs of the transaction.

V = Value of Input in satoshis I = Input Age and T = Transaction Size.

10. The Blockchain

The blockchain is basically a (back) linked list of blocks that store blocks of transactions. The bitcoin core software stores this blockchain using the Google LevelDB database. LevelDB is a key/value storage system that is accessible through their API. It isn't relational in nature and there is no SQL support for it. Each block in the blockchain is identified by a SHA256 hash. As well, each block contains a hash to the previous block in the chain. This creates a chain going all the way back to the first block, also known as the *genesis block*.

Each block in the blockchain can also (temporarily) have multiple children. This happens when two miners or mining pools discover a block at nearly the same time, causing a *fork* in the blockchain. Eventually the fork gets resolved as the chain with the most *work*, as defined by the difficulty level in each block of the fork. The fork with the highest difficulty becomes the permament structure.

The structure of each block in the blockchain is as follows:

Size	**Field**	**Description**
4 bytes	Block Size	The size of the block in bytes (not including this field).
80 bytes	Block Header	Structure containing the block meta-data.
1-9 bytes	Transaction counter	The number of transactions in this block
Variable	Transactions	The transactions encoded in the block

The block meta data contains the following fields:

Size	**Field**	**Description**
4 bytes	Version	Bitcoin version protocol number (currently set to 1)
32 bytes	Previous Block hash	A reference to the hash of the previous (parent) block.
32 bytes	Merkle Root	A hash of the root of the Merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block in seconds from January 1, 1970 (Unix epoch)

4 bytes	Difficulty target	The proof of work difficulty target for this block. (Used to resolve forks in the blockchain)
4 bytes	Nonce	A counter used for the proof of work algorithm.

Proof of Work algorithm

A new block is created on the blockchain through a concept called *proof of work*. A challenge string consisting of the hash of the most recent block header is hashed with the nonce value. The result must be numerically smaller than the value in the target field. A prospective miner must iterate through the values of the nonce (a 32 bit number) and for each iteration hash the nonce with the hash of the block header. If that value is smaller than then target, then the block is created. The miner inserts a *coinbase* transaction, which is always the first transaction encoded in the block. Currently, the miner will be awarded 25 BTC per block (At the time of this writing, one BTC is worth approximately 240 USD which is approximately 6000 USD. Sometime next year, the value of each block will halve to 12.5 BTC. This halving will continue until the year 2140, when the last bitcoin will be mined.

The bitcoin core protocol generates a new target for each block to be mined. The goal is to make sure that the average block generation time doesn't exceed ten minutes. Note that it doesn't mean that *all* blocks will be generated within ten minutes of the last one, merely that the running average doesn't exceed 10 minutes. If the running average over the last 2016 blocks has exceeded 10 minutes, then the target is recomputed using the following algorithm.

$$T = T_{prev} * t_{actual} / 2016 * 10\, minutes$$

Where T_{prev} is the old target value and t_{actual} is the time span of how long it took to generate the last 2016 blocks.

Note that there isn't necessarily only one solution to any given proof of work problem. Any number will do as long as that number is less than the target.

Additionally, miners will also earn *transaction fees*. Transaction fees are the remainder of the inputs – outputs. How the miners incorporate the transactions is left to them. It is therefore possible that a transaction without any transaction fees attached may not be incorporated at all. It is up to the sending wallet to determine whether the transaction has been confirmed and re-send it if some time limit has been exceeded.

11. The bitcoin network protocol.

When a new full node boots up, the first thing it must do is discover other peers on the network. Since the bitcoin network is not defined geographically, any existing bitcoin node can be chosen at random. Note that the bitcoin-core has some hard-coded “seed” nodes which should always be available. The new node can choose one of them to begin the discovery process.

Nodes will connect through TCP port 8333 (or an alternative if one is provided). Once a connection is established to a peer node, the new node will send a *handshake*. This handshake packet contains the following:

Field	Description
PROTOCOL_VERSION	Defines the network version of the P2P protocol spoken by Bitcoin.
nLocalServices	The services provided by this node. Currently only defined to support NODE_NETWORK
nTime	The current time
addrYou	The IP address of the remote peer node
addrMe	The IP address of the local node
subver	The software version of the bitcoin-core running on the local node.
BestHeight	The blockheight of the local node's blockchain.

Once the remote peer node receives the packet, it will respond with a *verack* packet to acknowledge receipt of the original packet and establish the network connection. The remote node may send it's own version packet if it wants to connect to the local network as a peer.

Once the node is connected, it will send an *addr* message to other nodes which will, in turn, broadcasts them to its connected nodes. This means that eventually the local node will become better known and better connected on the bitcoin network.

Once the full node is connected, it will try and construct and store a complete copy of the blockchain. If the node is brand new, it will only know about the *genesis* node, which is statically coded into the bitcoin core. A new full node will have to download all of the other blocks in the full blockchain in order to synchronize with its peers. This can take some period of time (24 hours or more). A node communicates with other nodes to retrieve the latest blocks in the blockchain by sending a *getblocks* message with the local nodes BestHeight value. If other nodes have a BestHeight value that is greater than the one

sent, then it will return the difference of all blocks that are higher up in the blockchain via an *inv* message containing the new blocks.

Appendix A. Miscellaneous Topics

Bitcoin Block sizes

There is currently a discussion between developers, miners and companies in the bitcoin space regarding the current bitcoin blocksize. The current blocksize is limited to a maximum size of one megabyte. This means that the maximum number of transactions per second is around 7 tps. Compare this with other major payment transaction systems...

System	TPS
Visa	Average of 2000 tps (can peak out at approximately 56000 tps)
Paypal	Average of 115 tps
Bitcoin	7 tps

The concern is that the Bitcoin network will become overwhelmed as more and more transations hit the network. Because of the small block size, unconfirmed transactions will pile up and the length of confirmation time for transactions will increase.

Because Bitcoin has no centralized maintainer, unlike, for example, Linux, where Linus Torvalds has final say over what gets committed to the code base, major upgrades (called *hard forks*) like the one suggested for the increase in block size become difficult to implement due to lack of strategic agreement amongst the stakeholders.

Given that Bitcoin is rising in popularity and that the number of transactions are increasing,

what is the way forward? The obvious answer may be to simply increase the block size (an initial new maximum limit was 20 MB), however, there are a number of people who object to this. A major objection is that many of the miners are based in China, where internet infrastructure is not as robust as in Europe and America, so having a larger block size increases internet usage, which may make it more difficult for miners in China.

Other proposals include a smaller increase to 8 MB (This is favored by many of the Chinese based miners).

Objections to the block size increase include the fact that the increased internet usage and costs will favor larger and more well funded mining operations, which will increase mining centralization.

Other arguments include the fact that making a change this drastic may result in

unforeseen events, including breaking other parts of the system.

Replace By Fee

Replace by Fee is another approach to the scalability problem mentioned in the previous topic. A fundamental rule of the Bitcoin core is that if two transactions or blocks that build off of the same dependency are submitted on the Bitcoin network, then the one that a given node sees first wins. This is the *first seen* rule. This incentivizes miners to build on top of the new block rather than wait and see if there is a better block available. The RBF rule says that a transaction can be undone if the second transaction has a higher transaction fee associated with it. In other words, blocks no longer get confirmed by the order that they are received, but by the amount of transaction fees included. Note that the first seen rule is not enforceable by the network, so miners are free to use RBF, however, no wallets (as of this writing) support RBF.