Joseph Puzzo
University of New Hampshire
November 4, 2015

The Bitcoin Protocol

Introduction:

Money and the way people exchange goods and services has changed drastically over the years. Though at its core, the idea of money has been very consistent. A physical median for exchange, typically backed by a governing authority. Until recently, this definition has remained static. That is, until Satoshi Nakamoto introduced Bitcoin and the modern day digital currency.

This document describes bitcoin ("b" for coin) and the Bitcoin protocol ("B" for protocol). Details about the currency itself will be defined first on a high level. After a core understanding of the coin, I will dive down into details of the Protocol itself. I assume no understanding of other application layer protocols for the readers of this document. That being said, it will be beneficial for the reader to have some understanding of data packets, and JSON format. JSON text will be used to describe structure formats, and provide readable examples.

What is Bitcoin?

The big question that people will ask is "what is bitcoin?". At first glance, the idea of a cryptocurrency seems pretty cool; but its hard for people to understand what is actually going on. There is much more to Bitcoin than meets the eye. In this section, I will go over what it is, and some of its hidden powers.

Bitcoin is a peer to peer "digital currency", or more appropriately its a "cryptocurrency". Digital currency simply means its a currency that is represented by bits in a computer, or in the case of bitcoin, many computers. Cryptocurrency by definition is a "medium of exchange using cryptography to secure the transactions and to control the creation of new units". In other words, the security and trust of the currency is implemented using cryptography. Finally, peer to peer means this currency is distributed from one user to another without an intermediate like a bank.

In order to understand this lets break Bitcoin down into a simple analogy. Suppose I have a piece of paper called the “public ledger”. This “ledger” is publicly on the internet, and therefore can be viewed by anyone in the world. But, unlike the traditional client server architecture that we might be familiar with, this ledger is not stored in one location(i.e on a server). Rather, the ledger is stored locally on many computers within the peer to peer network.

Now suppose we have a girl, Alice, that is storing this ledger on her computer. For doing so lets say she is rewarded 1 bitcoin (I will go into mining bitcoin later). So a new line is added to the ledger that states (1Bitcoin —> Alice). Soon after, a new person Bob decides to get in on bitcoin so he downloads the ledger from all of the peers, in this case only Alice, and now Bob has a copy of the ledger. This process continues for a while, and soon we have thousands of users that have this ledger on there computers.

Next, Alice decides she wants to send 1/2 a bitcoin to Bob. So, she announces to the bitcoin network that “I Alice have 1/2 bitcoin that I want to send to Bob” or more simply put (Alice 1/2 Bitcoin —> Bob). So because this is a public announcement that everyone can see, people can look at that announcement, and **as a community,** decide if that transaction is valid. Because everyone has access to the ledger that states (1Bitcoin —> Alice), everyone knows that Alice does indeed have a bitcoin to spend. So, after some extra secure verification (mining) this **transaction** will be written down on the public ledger on everyones computers.

So why is this so cool?! Because, we have just built a currency that is run by the community i.e has no governing body other than a “majority rules network” (Decentralized). This currency can be used by anyone that has access to a computer (Don’t even need internet, details later). And finally, the currency is secure i.e no counterfeiting (lying). The idea is that if everyone knows everything, then the only way someone can get away with lying is if over 50% of the network agrees with them. This is a revolutionary idea and actually has potential to solve many modern day network security issues. Just think, what if domain names were distributed like this!

Bitcoin Mining

If you are reading this document you have heard of Bitcoin Mining. Its a “magical” thing that you can do with your computer to make money from nothing (aside from using electricity). My goal in this section is to, on a high level, show you the man behind the curtain. I aim to expose the secrets of this magic in a simple understandable way.

So, what do we know? We know that there is a public ledger that everyone has access to, and we know that every time someone "spends" a bitcoin, it will get written down on this ledger. But we still have issues. How do we stop people from writing whatever they want? and where do bitcoins come from? What if JoShmo, just sends out an illegal copy of the ledger to everyone?

In order to solve these issues, Bitcoin does something very cool. To show this off, I will again start with an analogy. Suppose I give Alice a box with some bitcoin inside. The box is locked and can be opened with a key. In addition to the box, I give Alice a 1000 keys, where one of those keys will open the box. It may take some time and effort but after some time, its guaranteed that she will be able to unlock the box. Now suppose that the box also contains a few of the most recent transactions that have occurred on the network.
(Alice 1/2 Bitcoin —> Bob) (Bob 1/4 Bitcoin —> Eve)(Eve 1/4 Bitcoin —> Joe) ………
Finally, lets say that in order for any of the transactions in the box to be valid, the box must first be unlocked by someone.

So, what have we just done here!? We told Alice she would get a reward if she unlocked the box, and we stated that by unlocking that box the transactions inside would be validated. The first AhHa moment that I want to get across is the idea that by doing this work you keep the Bitcoin Network operational, and you get a reward. AhHa! So we have given incentive to helping the community! Why is this so powerful? Because it incentives people to be good and not bad. People cannot spend bitcoin without this **work,** and you get paid to do the work.

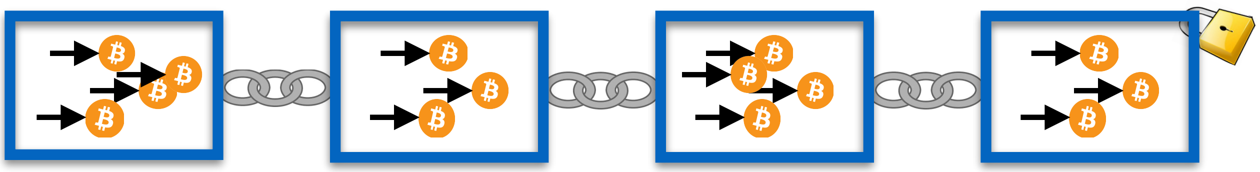
This process of finding this key and unlocking the box is called Mining. Similar to how you must mine gold in order to get get the reward. The next AhHa moment, this is where bitcoin comes from! By doing work to verify transactions, the miner that finds the key will be rewarded with some new bitcoin (1Bitcoin —> Alice). This transaction is formally known as a coinbase transaction because there is no sender, only a receiver.

Cool, so by trying these keys I can get a reward. Now lets say 1000 more people decide they want a piece of the pie and try to unlock the lock themselves. Problem, now it has become very easy to unlock the lock, i.e each person just tries there own key all at once. How does Bitcoin solve this problem? Staying with this analogy suppose that when more people join the mining community we simply increase the amount of keys. Rather than 1000 keys just for Alice, we now throw out 100,000 keys for 1000 people. By doing this, we are stabilizing the network such that bitcoins are mined at a steady rate.

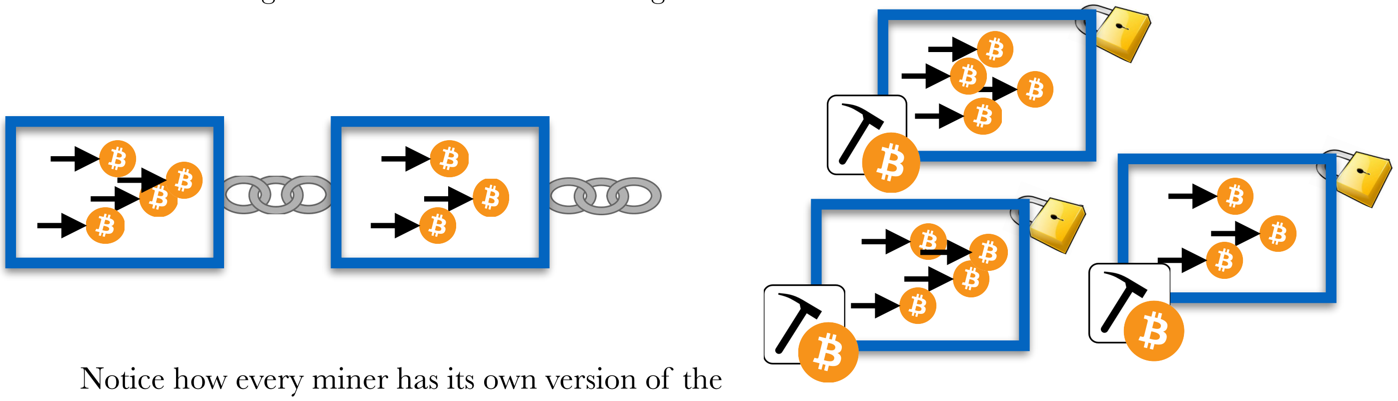
In reality there are no physical keys. Instead, the protocol uses cryptographic hashing but I will get to those details later. Though its more complex and involves some unique math, the idea is essentially the same. The box is locked and to unlock it you need a key, to find the key you need to do some work, and if you do the work you get paid. How Simple!

The Block Chain

Another element that you may have heard about is the BlockChain. From our analogy, the block chain is a linked chain of unlocked boxes that contain transaction records. As a whole the block chain is what I previously referred to as, the public ledger. Each “block” in the chain is really just a block header, and a list of transactions (tx’s). This chain starts with the first block ever mined known as the Genesis Block. This is the only block in the chain that does not have a previous block. The chain continues until it reaches the tail which is the most recent block.

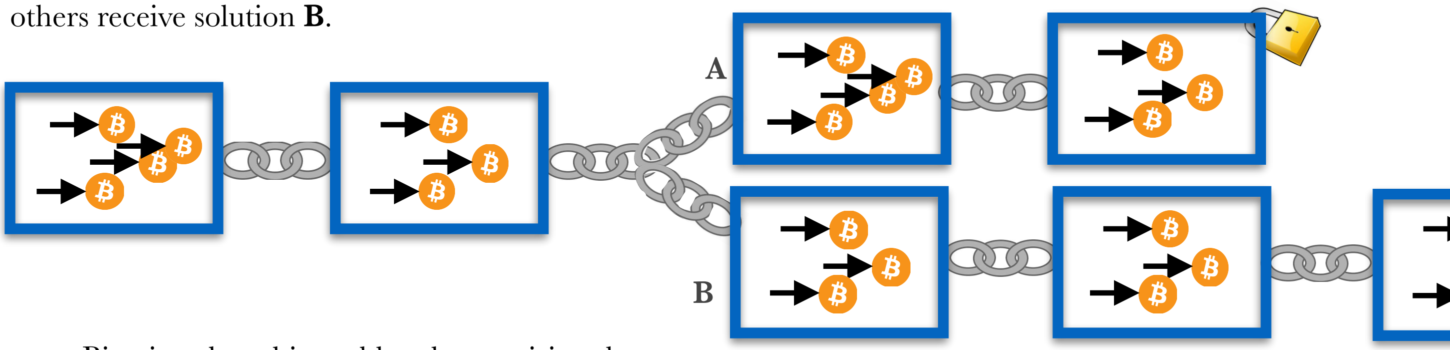


The tail of this chain is locked, i.e has the key has not yet been found. If a transaction has just been initiated it will be broadcast to the network and may end up in this block. The term “may” is used because that broadcast must have made it into the block of the winning miner. In reality there are many different versions of the current block and only the version that the winning miner holds (i.e the miner that finds the key) will be added to the chain. In order to get a better understanding of this I have created the diagram below.



Notice how every miner has its own version of the current block. The first miner to unlock its block will add a reward to their block. and broadcast the solution to the network. By "add the reward" I mean the miner that wins will add a transaction to the new block that sends the specified reward amount (new bitcoins mined) to there own bitcoin address.

One issue that must be noted is the creation of a "fork" in the chain. This may occur if two people come up with valid keys at the same time. Suppose we have a miner **A** and a miner **B** that are both trying to unlock the block. At some point in time **A** finds the key to the lock, and only seconds later, so does **B**. Both **A** and **B** then broadcast there solutions to the network. Due to the random propagation of the network, some of the nodes on the network receive solution **A** and others receive solution **B**.



Bitcoin solves this problem by requiring that miners work on the longest chain. After some time is passed, the next block in one of the two forks will be solved. At this point all of the miners working on the shorter fork will switch to the longer fork and life will go on. So in the above case, the miners working of fork **A** will switch over to fork **B**.

Proof of Work:

So far I have eluded to a very important and powerful part of the Bitcoin Protocol. It turns out, this idea was already explained in the Bitcoin Mining section. In order to prove that a bitcoin transaction is valid, we now know that you need to include it in a block. We also know that a certain amount of work is required to unlock that block. This is called proof of work, because you are required to perform a difficult (costly, time-consuming) problem and get a verifiable result. In other words the problem is very difficult to solve but once solved its very easy to prove that the solution is correct.

In Bitcoin, miners use an algorithm know as the hashcash. The idea is that given a new block to work on, you calculate the SHA256(SHA256(Block_Header)). This will result in some hash "x". The goal is to hash a value x that is <= to a set target "t" (00000FFFFFFFFF000….).

In order to achieve this magic value, the miner will rehash their block over and over. At each iteration, the miner will increment a nonce value within the header, thus, resulting in a different hash "x2". The search for this magic value "v" where v < t is the work that was mentioned before. The following is a description of this process.

Suppose We had the following: y = H(x) where H is a hash function, x is the key, and y is the result. Given x and H, it is very simple to compute y, but given only y, its computationally infeasible to compute x. The security is defined by O(2^k), where k is the hash size (k=256 for bitcoin). As previously described, the goal is to produce this value y that meets a specific requirement. In order to better describe this requirement, let me first introduce the concept of a **preimage.**

Given a function f(x) -> y, the image of x is defined to be f(x) or y (so the result). The preimage of y is then defined to be $f^{-1}(y) = \{x \mid f(x) = y\}$ or any x whose image is y. In other words the preimage is any value x that will result in the value y.

f(x) = y   x = preimage   y = image   f = hash function

Given a a preimage x our goal is to find a second preimage x`. It is very hard and time consuming to do this, so this works perfectly for our computationally difficult problem. But we have an issue with this process. We have no way of controlling how difficult the problem is. In other words it may take 10 minutes to 10 hours to solve this problem. To solve this problem, Bitcoin uses the idea of a **partial preimage**. Rather than finding a value x` that also hashes to y, we find a value x` that hashes to a value close enough to y. Now we can control how hard it is to solve the problem by varying the "closeness" of H(x`) to H(x).

This is where bitcoins difficulty comes from! Similar to adding more keys for alice and friends to try, varying this value will make mining bitcoin harder or easier. Bitcoin will vary this value every two weeks in order to keep the network stable. A stable Bitcoin network, is one where a block is mined approximately every 10 minutes.

Standards and Structures:

Now that you have a basic understanding of how bitcoin works, I will now dive into the protocol itself. In order to do this, I follow closely with the format and order of the Protocol Documentation on the bitcoin wiki. This wiki was the main source used for this document and should be referenced for additional details.

To better understand the different parts of the protocol, it's important to first get an understanding of some of the main standards and structures used within the protocol. By standards and structures, I am referring to unique data structures and formats used. One of these, A SHA-256 hash, has already been mentioned in a previous section, and is simply a hashing function that returns a 256 bit value from some given input.

In order to describe the protocol, I will be using tables directly from the wiki, and JSON strings for examples. Below is an **example** that details what this will look like.

Person

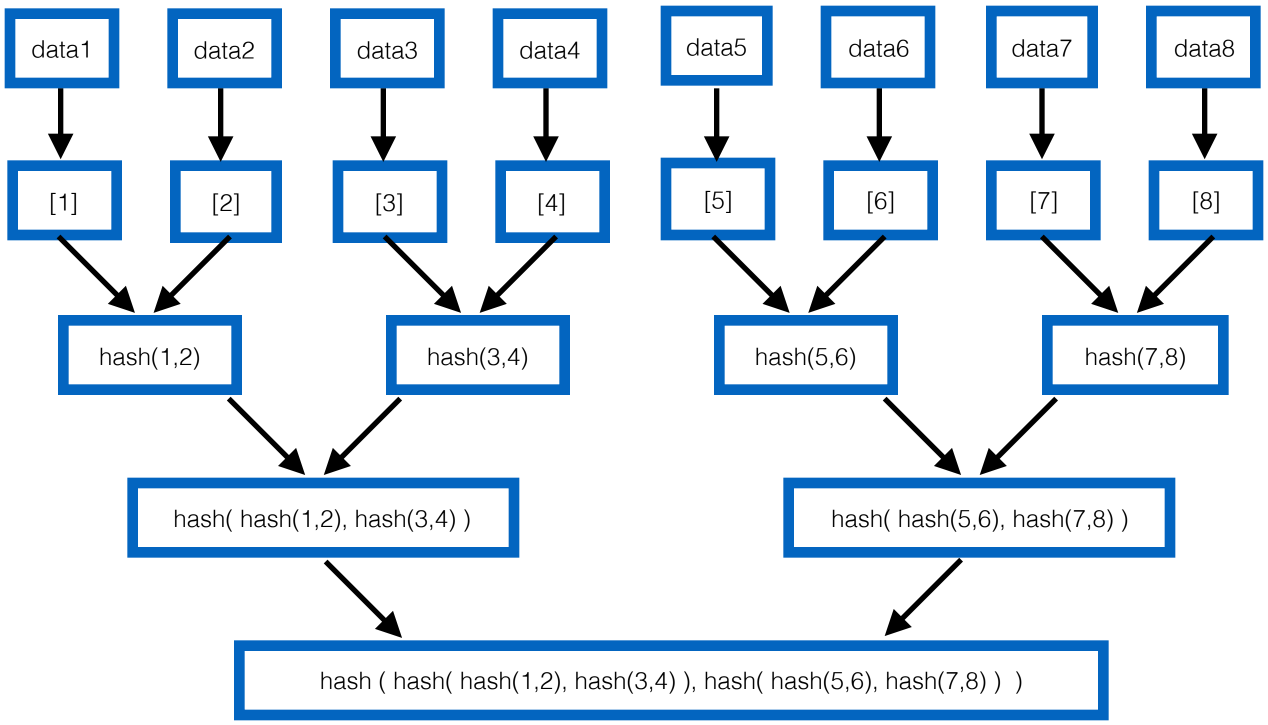
Field Size	Description	Data type	Comments
4	Age	uint32	age in years
8	Name	char[16]	name stored in a 16 byte value

```
{"Person": {
        "Age": 20,
        "Name": "Joseph Puzzo"
}}
```

Merkle Trees

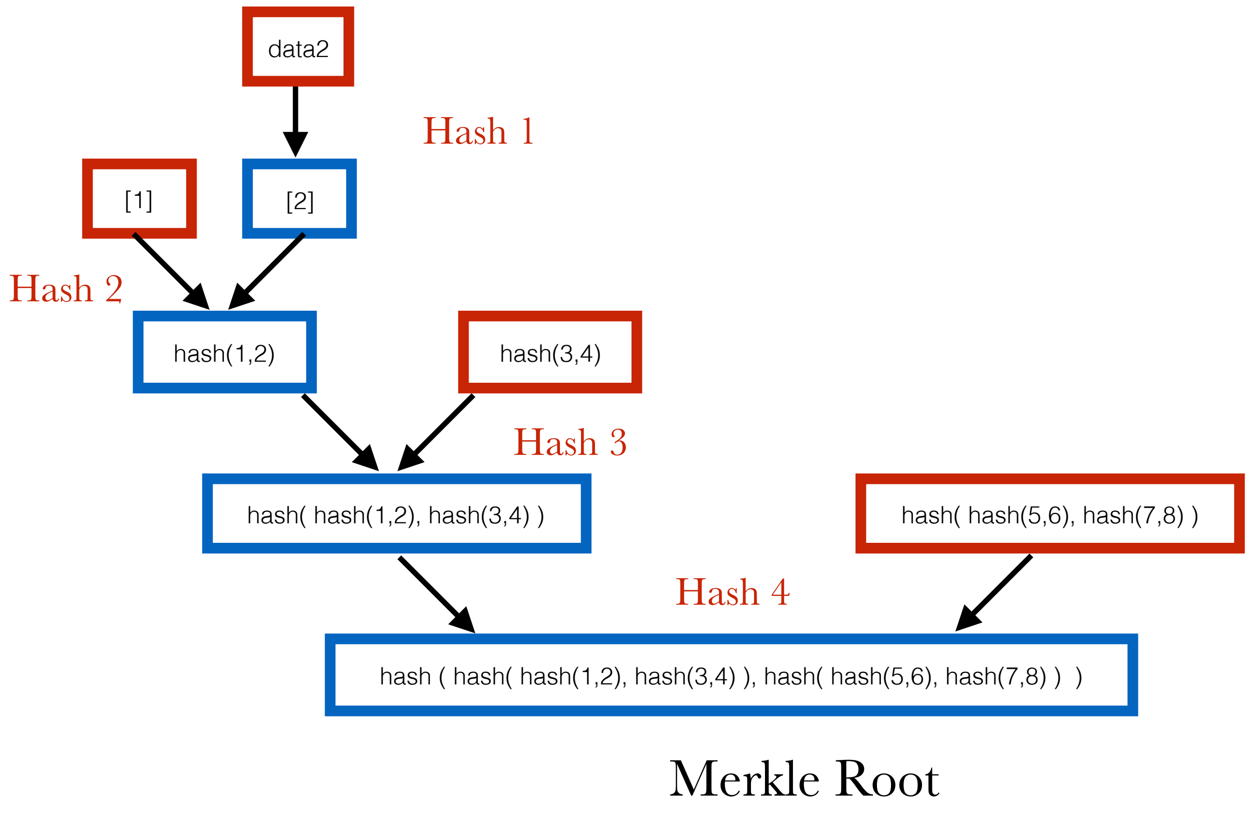
One of the data structures used in bitcoin is known as a Merkle Tree. Simply put, a Merkle tree is a binary tree of hashes. The idea is that given some set of data, [1,2,3,4,6,7,8,], we can build a tree from this data that allows us to quickly and efficiently verify that part of this data is correct. In other words if a tree exists that was built from this unique dataset, we can then share a piece of this data with someone and they can grab part of the tree and verify that the data provided is correct. The following describes this in more detail.

Suppose we had a set of data, S1 = [1,2,3,4,5,6,7,8]. We then split this data up into individual blocks [1] [2] [3] [4] [5] [6] [7] [8]. Lets denote each of these blocks as a leaf node: a node at the bottom of the merkle tree that contains the actual data. Now starting from the left and going right [1] [2] … [7] [8] we hash data together in pairs, i.e hash([1]+[2]) … hash([7]+[8]). The result from these hashes are then put into a new set, S2 = [hash(1,2), hash(3,4), hash(5,6), hash(7,8)]. We continue this process until we reach a set Sn who's length is = 1. Essentially we have done the following:


data1
data2
data3
data4
data5
data6
data7
data8
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
hash(1,2)
hash(3,4)
hash(5,6)
hash(7,8)
hash(hash(1,2), hash(3,4))
hash(hash(5,6), hash(7,8))
hash (hash(hash(1,2), hash(3,4)), hash(hash(5,6), hash(7,8)))

Merkle Root

By using this structure we can validate a piece of data, say for example data2, by obtaining the hash of data1 ([1]) , hash(3,4), and the hash(hash(5,6), hash(7,8)). This is much more efficient then downloading the entire tree because we simply have one piece of data, and 3 SHA256 hashes. In other words we are only working with the (size of data2) + (3 * 256 bit values) and we are only hashing 4 times to calculate the Merkle root. The following diagram shows the power of this structure.


data2
Hash 1
[1]
[2]
Hash 2
hash(1,2)
hash(3,4)
Hash 3
hash(hash(1,2), hash(3,4))
hash(hash(5,6), hash(7,8))
Hash 4
hash (hash(hash(1,2), hash(3,4)), hash(hash(5,6), hash(7,8)))
Merkle Root

Bitcoin Address:

Bitcoin utilizes public private key cryptography. This provides bitcoin with anonymity and security. Bitcoin specifically uses this technology to verify and sign transactions that occur in the network. A private key in the context of bitcoin, as defined by the wiki is “a secret number that allows bitcoins to be spent”(Protocol Documentation). Private keys are typically encrypted and stored by the bitcoin client. These keys are kept secret, as they are the “tickets that allow bitcoin to be spent”.

In bitcoin a private key is generally 256 bits long and can be represented in different formats. Here is an example from the wiki, of a bitcoin private address in hexadecimal format:

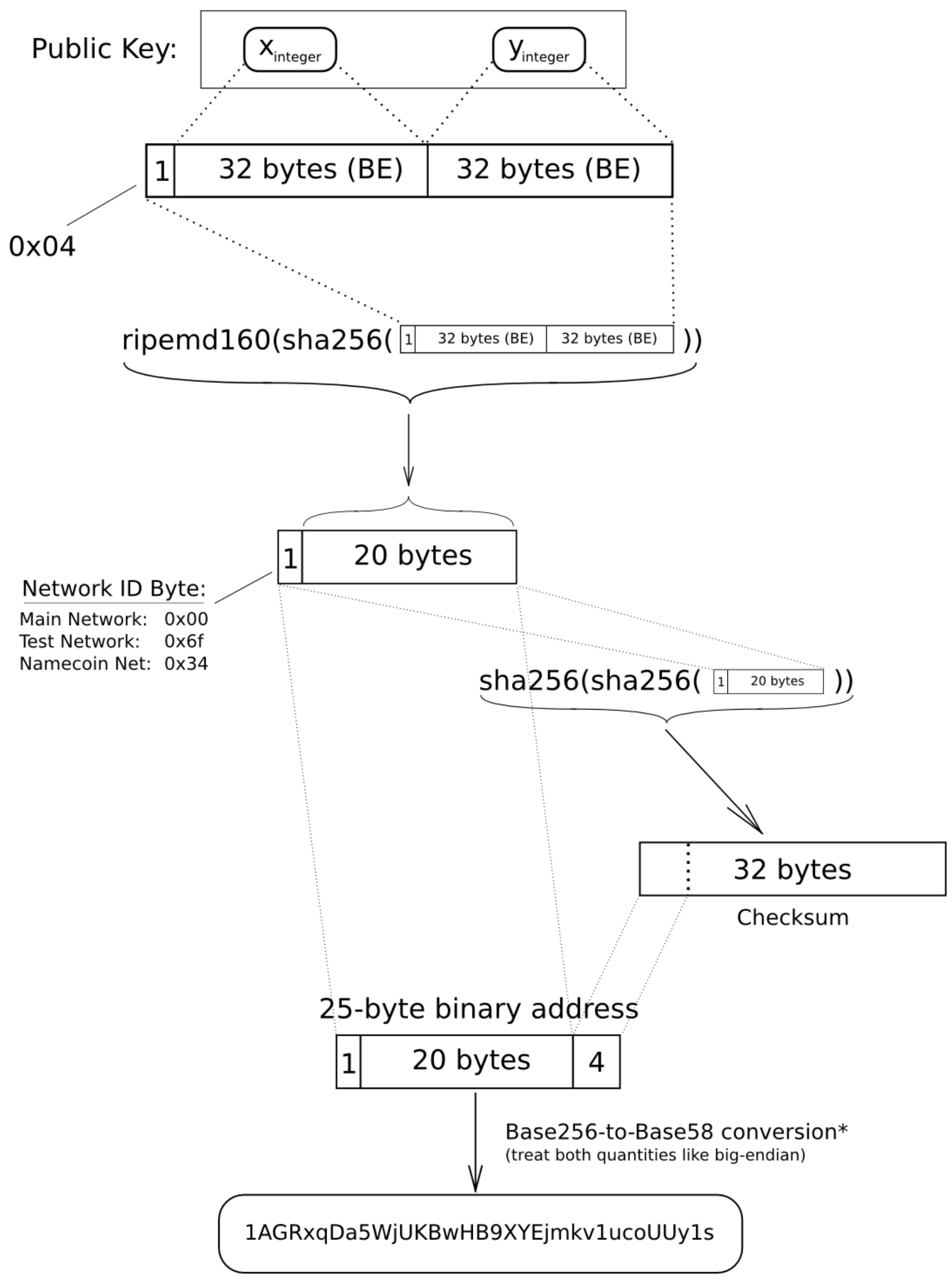
`E9 87 3D 79 C6 D8 7D C0 FB 6A 57 78 63 33 89 F4 45 32 13 30 3D A6 1F 20 BD 67 FC 23 3A A3 32 62`. A private address is used to sign transactions and generate public keys.

Public keys are the keys you are allowed to share (thats why its called public!). A public key is generated by performing some elliptic curve cryptography on a private address. This makes it easy generate a public key from a private address and nearly impossible to generate the private key given the public key. Essentially a public key is created in the following way: ElipticCurveCrypto(public-key) —> (x: 256 bits, y: 256 bits) resulting in a 512 bit key (Uncompressed). Compressed keys simplify this design by only using the x coordinate and adding a 1 byte flag that states what side of the curve the point lies on. Thus a private key is 256 bits + 1 byte = 264 bits. In order to tell the difference the additional byte is used, see diagram below [bobalot].

0x02 <32 byte x>	Compressed Even Keys
0x03 <32 byte x>	Compressed Odd Keys
0x04 <32 byte x> <32 byte y>	Uncompressed Keys

When you want to send a bitcoin to someone, you first obtain their bitcoin address, and then generate a new transaction that sends bitcoin two this address. As a side note, try not to confuse public **keys** and bitcoin **addresses**, they are two different things! Public keys are used to create addresses that you send bitcoin too, i.e some_function(public-key) —> bitcoin address. Stated very nicely in a reddit thread, "your address is a prefix byte of 0x00, the ripemd160(sha256(public key)) hash and then a checksum postfix". This process is depicted in the diagram on the following page. An example of a bitcoin public address is 3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy. This is represented in base 58 format, meaning [a-Z 0-9] but excluding uppercase letter "O", uppercase letter "I", lowercase letter "l", and the number "0" in order to prevent visual ambiguity. By viewing and storing addresses in this way, it makes dealing with bitcoin simpler.

Elliptic-Curve Public Key to BTC Address conversion

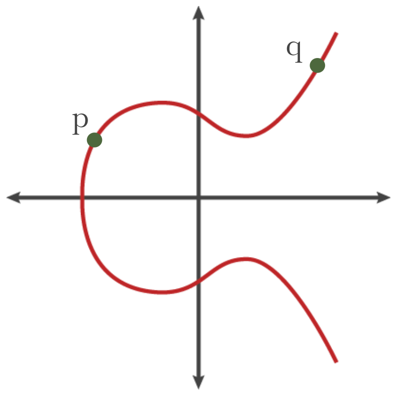


https://en.bitcoin.it/w/images/en/9/9b/PubKeyToAddr.png

Elliptic Curve Cryptography:

The diagram on the previous page is great for describing address generation, but where did this public key come from? Introducing elliptic curve cryptography. For bitcoin to be successful, we need a secure method of key generation, and ECC is the perfect tool for the job. ECC is pretty complex but can be explained simply on a high level.

Suppose we had a elliptic curve (see figure below). This curve is defined by some known equation f. Suppose we also define an arbitrary point on this curve p. Lets define p to be the start point. Now, lets take the point p, and perform a **unique** multiplication on it n times. This multiplication will result in some point q . The point q is our public key! Cool, so we have come up with a method to calculate some point q given f, p and n. The power behind ECC is that given only f, p, and q, its nearly impossible to find n. If it has not become apparent, n is the private key!



In reality this process is slightly more complex but the principle is the same. ECC is one of the most secure methods of encryption to date, and is used widely. The diagram below details this algorithm in more visually.

f = some_function

f (x1) = y1

p = (x1, y1)

n = private key

UniqueMultiply(p, n) = q

q = (x2, y2) = public key

Network Address:

In addition to having Bitcoin addresses, the protocol also makes use of Network addresses. A network address in the context of bitcoin, is simply a structure that holds some basic network information. The format of these addresses is described below.

Field Size	Description	Data type	Comments
4	time	uint32	the Time (version >= 31402). **Not present in version message.**
8	services	uint64_t	same service(s) listed in version
16	IPv6/4	char[16]	IPv6 address. Network byte order. The original client only supported IPv4 and only read the last 4 bytes to get the IPv4 address. However, the IPv4 address is written into the message as a 16 byte IPv4-mapped IPv6 address (12 bytes *00 00 00 00 00 00 00 00 00 00 FF FF*, followed by the 4 bytes of the IPv4 address).
2	port	uint16_t	port number, network byte order

```
{ “Network Address": {
        “services”: “NODE_NETWORK”,
        “Ipv6/4”: “10.10.10.1”,
        “port”: 9000
}}
```

Inventory Vectors:

In order to share object information, the Bitcoin protocol uses a structure called an inventory vector. An inventory vector is a simple structure that allows nodes to inform other nodes of data that they may have or data that may have been requested.

Field Size	Description	Data type	Comments
4	type	uint32_t	Identifies the object type linked to this inventory
32	hash	char[32]	Hash of the object

```
{ “Inventory Vector": {
        “type”: “MSG_BLOCK”,
        “hash”: “6bdcc421be9045d3ffff47b62cfd0c654dad6da916b60b8e010395f3bd956d”
}}
```

Block Headers:

One of the more important structures used in the protocol are Block Headers. This structure details block information and is used to share this information between nodes. I previously stated, the proof of work involved calculating the SHA256(SHA256(Block_Header)). This header format is detailed below and includes the varying nonce field.

Field Size	Description	Data type	Comments
4	version	uint32_t	Block version information, based upon the software version creating this block
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A timestamp recording when this block was created (Will overflow in 2106[2])
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block… to allow variations of the header and compute different hashes
1	txn_count	var_int	Number of transaction entries, this value is always 0

Below is the actual block header for block 354022 taken from **`blockexplorer.com`**. Note that block explorer will also include the hash of the block itself: "hash", the actual size: "size", and actual number of transactions: "n_tx".

{"Block Header": {
~~"hash":"00000000000000000b650e883088dcb1b042dd080ae71228d59124c5d3b30358",~~
"ver":2,
"prev_block":"000000000000000003eb574fc3e1da8f04a1074c58449ad1403c83c5bff47",
"mrkl_root":"6bdcc421be9045d3ffff47b62cfd0c654dad6da916b60b8e010395f3bd956d",
"time":1430193209,
"bits":404166640,
"nonce":3137234611,
"n_tx":~~509~~,
~~"size":450966,~~
}}

Message Header:

Each bitcoin packet contains a message header. The header is very basic and contains information that is needed to decode the packets.

Message Header

Field Size	Description	Data type	Comments
4	magic	uint32_t	Magic value indicating message origin network, and used to seek to next message when stream state is unknown
12	command	char[12]	ASCII string identifying the packet content, NULL padded (non-NULL padding results in packet rejected)
4	length	uint32_t	Length of payload in number of bytes
4	checksum	uint32_t	First 4 bytes of sha256(sha256(payload))
?	payload	uchar[]	The actual data

```
{“Message Header”: {
        “magic”: “main”,
        “command”: “version”,
        “length”: 85,
        “checksum”: 0x017d0000,
        “payload”: {“Version Message”{
                “Message Data”: …..
            }}
}}
```

Magic Values

Network	Magic value	Sent over wire as
main	0xD9B4BEF9	F9 BE B4 D9
testnet	0xDAB5BFFA	FA BF B5 DA
testnet3	0x0709110B	0B 11 09 07
namecoin	0xFEB4BEF9	F9 BE B4 FE

In the example above you will notice a magic value of “main”. This value simply means that this packet is on the main network as opposed to a test network. The “command field simply tells us that this packet will contain a Version Message. This field will be different for each type of predefined message type. The header also contains a length and checksum field to help process the data and detect errors. And finally, the payload field is where the rest of the packet will go.

Message Types:

There are many different types of messages in the Bitcoin protocol, but in order to keep this document small and concise, only a few will be covered. Specifically I will be covering messages that ask for, and exchange important information. Additional details will be provided that help reenforce the topics previously covered.

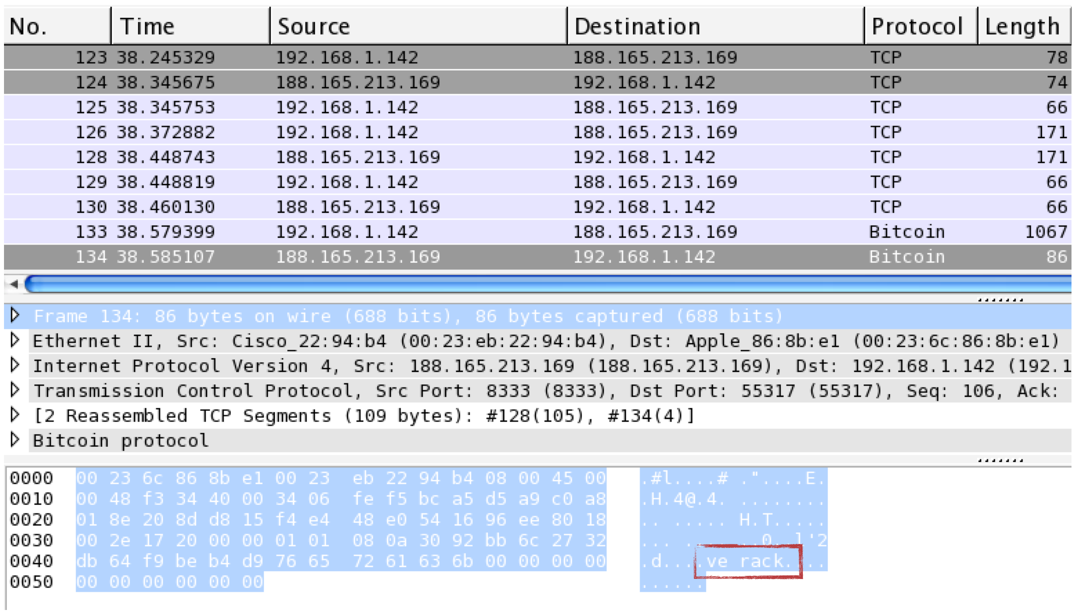
Version:

The version message is a fairly simple message that is used to initiate a connection. When two nodes wish too connect, they must first exchange versions. Any additional communications between the nodes must occur after the two nodes have agreed on a version.

Field Size	Description	Data type	Comments
4	version	int32_t	Identifies protocol version being used by the node
8	services	uint64_t	bitfield of features to be enabled for this connection
8	timestamp	int64_t	standard UNIX timestamp in seconds
26	addr_recv	net_addr	The network address of the node receiving this message
version >= 106			
26	addr_from	net_addr	The network address of the node emitting this message
8	nonce	uint64_t	Node random nonce, randomly generated every time a version packet is sent. This nonce is used to detect connections to self.
?	user_agent	var_str	User Agent (0x00 if string is 0 bytes long)
4	start_height	int32_t	The last block received by the emitting node
1	relay	bool	Whether the remote peer should announce relayed transactions or not, see BIP 0037, since version >= 70001

```
{“Version Message”: {
        “version”: 60002,
        “services”: 0x01 00 00 00 00 00 00 00, ( NODE_NETWORK )
        “length”: “Tue May 6 10:12:33 PST 2015”,
        “addr_recv”: {Network Address{….}},
        “addr_from”: {Network Address{….}},
        “nonce”:3137234611,
        “user_agent”: “/Satoshi:0.7.2/“,
        “start_height”: 9864,
        “relay”: false,
}}
```

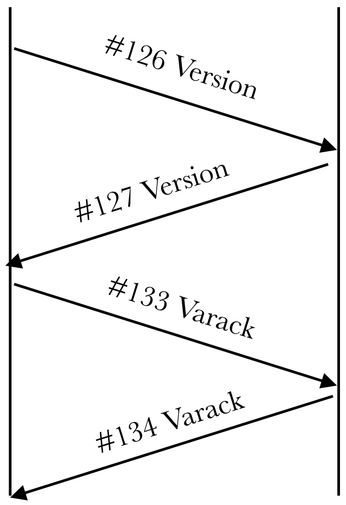
In response to this message, a node must send a “varack”. This is a simple acknowledgement that lets the sender know the recipients status. So An initial connection between two noes will look like the following:



No.	Time	Source	Destination	Protocol	Length
123	38.245329	192.168.1.142	188.165.213.169	TCP	78
124	38.345675	188.165.213.169	192.168.1.142	TCP	74
125	38.345753	192.168.1.142	188.165.213.169	TCP	66
126	38.372882	192.168.1.142	188.165.213.169	TCP	171
128	38.448743	188.165.213.169	192.168.1.142	TCP	171
129	38.448819	192.168.1.142	188.165.213.169	TCP	66
130	38.460130	188.165.213.169	192.168.1.142	TCP	66
133	38.579399	192.168.1.142	188.165.213.169	Bitcoin	1067
134	38.585107	188.165.213.169	192.168.1.142	Bitcoin	86

```
▷ Frame 134: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
▷ Ethernet II, Src: Cisco_22:94:b4 (00:23:eb:22:94:b4), Dst: Apple_86:8b:e1 (00:23:6c:86:8b:e1)
▷ Internet Protocol Version 4, Src: 188.165.213.169 (188.165.213.169), Dst: 192.168.1.142 (192.1
▷ Transmission Control Protocol, Src Port: 8333 (8333), Dst Port: 55317 (55317), Seq: 106, Ack:
▷ [2 Reassembled TCP Segments (109 bytes): #128(105), #134(4)]
▷ Bitcoin protocol
```

```
0000  00 23 6c 86 8b e1 00 23  eb 22 94 b4 08 00 45 00   .#l....# ."....E.
0010  00 48 f3 34 40 00 34 06  fe f5 bc a5 d5 a9 c0 a8   .H.4@.4. ........
0020  01 8e 20 8d d8 15 f4 e4  48 e0 54 16 96 ee 80 18   .. ..... H.T.....
0030  00 2e 17 20 00 00 01 01  08 0a 30 92 bb 6c 27 32   ... .... ..0..l'2
0040  db 64 f9 be b4 d9 76 65  72 61 63 6b 00 00 00 00   .d....ve rack....
0050  00 00 00 00 00 00                                  ......
```



In the capture above, packets 126 and 127 contain version messages, these are soon followed by packets 133 and 134 that contain the varack packets (see highlighted section in figure).

Addr:

In order to distribute information throughout the network, nodes must share information about other nodes. In order to do so, nodes must know about other nodes in the network, and must have the ability to communicate with them. Bitcoin distributes this using information using “addr” messages. Address messages are very simple, they only contain the amount of addresses and an array of net_addr objects.

Field Size	Description	Data type	Comments
1+	count	var_int	Number of address entries (max: 1000)
30x?	addr_list	(uint32_t + net_addr)[]	Address of other nodes on the network. version < 209 will only read the first one. The uint32_t is a timestamp (see note below).

```
{"Address Message": {
        "count": 20,
        "addr_list": [ net_addr{....},
                       net_addr{....},
                       net_addr{....}
                       .... ],
}}
```

Getter Messages:

Bitcoin uses various "getter" messages to request data from other nodes. These getter messages exist in two different formats.

Format 1

Field Size	Description	Data type	Comments
?	count	var_int	Number of inventory entries
36x?	inventory	inv_vect[]	Inventory vectors

Format 2

Field Size	Description	Data type	Comments
4	version	uint32_t	the protocol version
1+	hash count	var_int	number of block locator hash entries
32+	block locator hashes	char[32]	block locator object; newest back to genesis block (dense to start, but then sparse)
32	hash_stop	char[32]	hash of the last desired block; set to zero to get as many blocks as possible (500 for get blocks, 2000 for getheaders)

The following is a table that shows these various types and what they are used for. Refer to the formats detailed above for additional details.

Message Type	Format	Description
inv	1	Allows a node to advertise its knowledge of one or more objects. It can be received unsolicited, or in reply to *getblocks*. (Protocol Documentation)
getdata	1	getdata is used in response to inv, to retrieve the content of a specific object, and is usually sent after receiving an *inv* packet, after filtering known elements. (Protocol Documentation)
notfound	1	notfound is a response to a getdata, sent if any requested data items could not be relayed (Protocol Documentation)
getblocks	2	This packet is used to request block hashes from its peers (Can request up to 500) Get blocks simply asks for an "inv" packet in return.
getheaders	2	This packet is used to request block header data from its peers (Can request up to 2000)

As you can see, the last two messages utilize format 2 because they are specifically requesting block data, where the first three simply advertise or ask for any other data.

Block Message:

When a get data message requests for transaction information within a specific block, a block message will be returned by the receiver. A block message is crucial, because it contains some of the most valuable information within Bitcoin, the transactions. The requesting node can verify this message by taking the SHA256(SHA256(Block_Header)) i.e the first 6 fields of this message, and comparing it against the hash that it had locally.

Field Size	Description	Data type	Comments
4	version	uint32_t	Block version information, based upon the software version creating this block
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A Unix timestamp recording when this block was created (Currently limited to dates before the year 2106!)
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block… to allow variations of the header and compute different hashes

Field Size	Description	Data type	Comments
?	txn_count	var_int	Number of transaction entries
?	txns	tx[]	Block transactions, in format of "tx" command

Again for this example I used block 354022 taken from `blockexplorer.com`. Although, in this case I have matched the formatting of this message exactly.

```
{"Block Header": {
        "ver":2,
        "prev_block":"000000000000000003eb574fc3e1da8f04a1074c58449ad1403c83c5bff47",
        "mrkl_root":"6bdcc421be9045d3ffff47b62cfd0c654dad6da916b60b8e010395f3bd956d",
        "timestamp":1430193209,
         "bits":404166640,
        "nonce":3137234611,
        "txn_count":509,
        "txns": [ tx{….}, tx{….},tx{….},tx{….},tx{….},tx{….},…],
}}
```

Headers Message:

Not to be confused with a block message, a headers message is sent in response to a getheaders packet. Rather than returning a specific block and all of its data, a headers message returns an array of block_headers.

Field Size	Description	Data type	Comments
?	count	var_int	Number of block headers
81x?	headers	block_header[]	Block headers

```
{"Headers Message": {
        "count": 20,
        "headers": [ block_header{….},
                     block_header{….},
                     block_header{….},
                      …. ],
}}
```

Transactions:

Finally one of the most important parts to the Bitcoin protocol is a transaction, or “tx" for short. A transaction is basically some data that describes the transfer of bitcoin from one address to another. So lets build one!

Suppose through three previous transactions, Alice has acquired 1.8 bitcoin. Lets describe these previous transactions as previous_out1, previous_out2 and previous_out3.

Bob —— previous_out1: .5 bitcoin ——>

Joe —— previous_out2: 1 bitcoin ——> Alice (now has 1.8 bitcoin)

Bill —— previous_out3: .3 bitcoin ——>

Now suppose Alice wants to send .6 bitcoins to Eve. What is she to do? Currently she has a previous_out that states Bob gave her .5 bitcoin, a previous_out that states Joe gave her 1 bitcoin and a previous_out that states that Bill gave her .3. So Alice has three things to work with. At first glance everyone will probably assume the following as a solution:

Alice —— .5 + .3 bitcoin —> Eve

Eve —— .2 bitcoin ————> Alice (Change .. right!?) … NO!

The problem with this solution is that Alice would have to trust that Eve would send .2 bitcoin back in a separate transaction. This just isn't practical for an anonymous network. Also, you may be confused by how Eve was able to split .3 or .5 into the .2 in change. Lets describe the real solution to clear all of this up.

In reality, in order to get change, Alice will send change back to herself:

Alice —— .5 and .1 bitcoin —> Eve

Alice —— .2 bitcoin —> Alice

Yes this is strange, but its how it works and it solves a lot of problems. But how did she split that money up? Simple, Alice just “wrote down” two new transactions that totaled up to .6 and .2. We, the bitcoin community, know that she had .5 and .3 bitcoin to spend from her previous dealings so she can now split it up however she would like! In fact, Alice could have also done the following:

Alice —— .3 and .3 bitcoin —> Eve

Alice —— .2 bitcoin —> Alice

Where in this case, she took her change out of the previous_out1(.5) instead of the (.3)

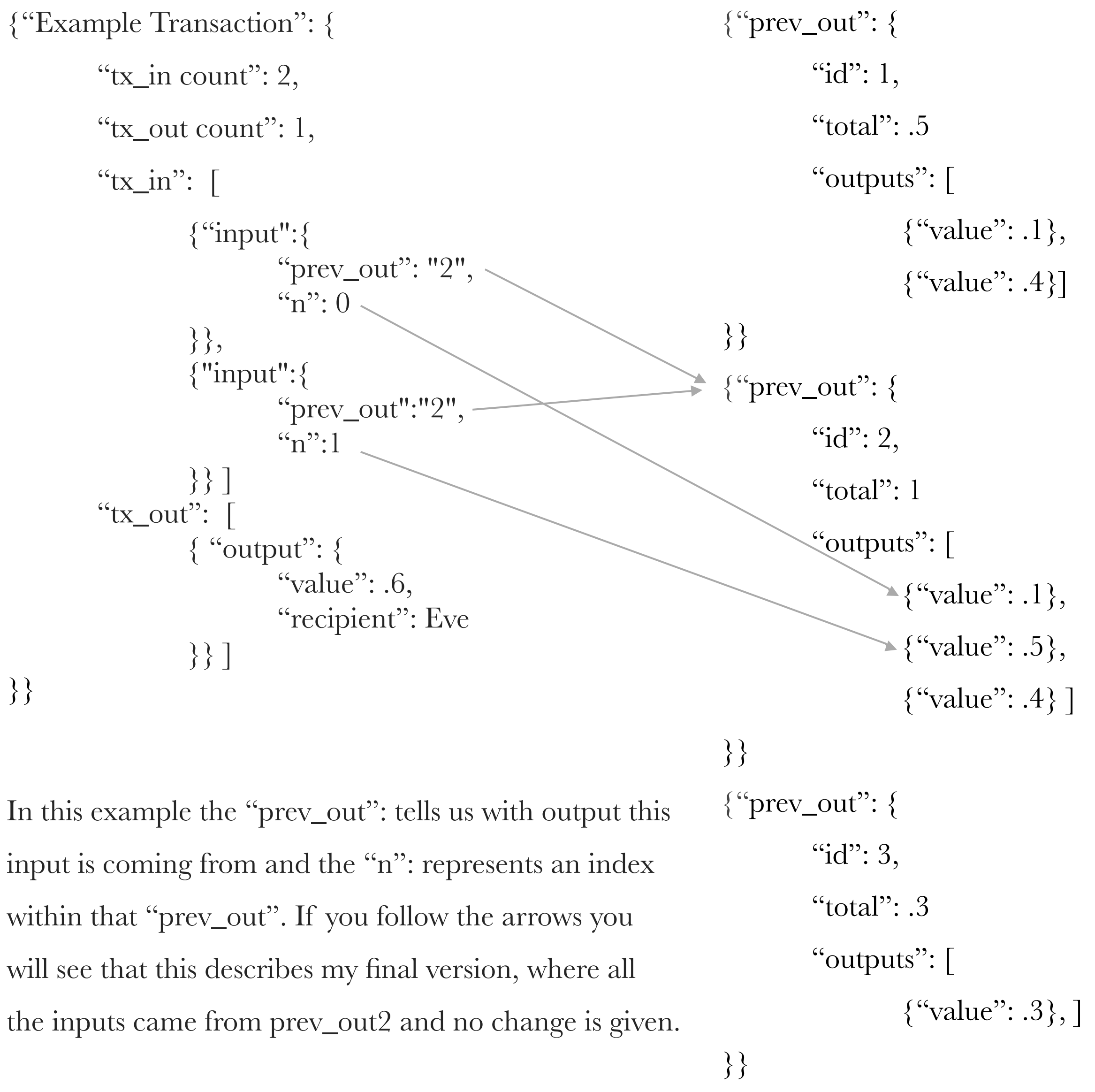
To go one step further, Alice could have even taken .6 out from previous_out2 (1 bitcoin) and sent back .4 to herself.

Alice —— .6 bitcoin—> Eve

Alice —— .4 bitcoin —> Alice

And you will hate me for doing this, but we could even do this another way! If previous_out2 was built from more previous_outs :) , i.e (.1 bitcoin + .5 bitcoin + .4 bitcoin) = previous_out2, then Alice would only need to take the first two outputs from previous_out2 (.1 + .5) and send that to Eve. Cool right!

To get a better understanding of this process, the following JSON is a sudo representation of that last version (Dumbed down version of a transaction)



```
{“Example Transaction”: {
        “tx_in count”: 2,
        “tx_out count”: 1,
        “tx_in”:  [
                {“input”:{
                        “prev_out”: "2",
                        “n”: 0
                }},
                {"input":{
                        “prev_out”:"2",
                        “n”:1
                }} ]
        “tx_out”:  [
                { “output”: {
                        “value”: .6,
                        “recipient”: Eve
                }} ]
}}
```

```
{“prev_out”: {
        “id”: 1,
        “total”: .5
        “outputs”: [
                {“value”: .1},
                {“value”: .4}]
}}
{“prev_out”: {
        “id”: 2,
        “total”: 1
        “outputs”: [
                {“value”: .1},
                {“value”: .5},
                {“value”: .4} ]
}}
{“prev_out”: {
        “id”: 3,
        “total”: .3
        “outputs”: [
                {“value”: .3}, ]
}}
```

In this example the “prev_out”: tells us with output this input is coming from and the “n”: represents an index within that “prev_out”. If you follow the arrows you will see that this describes my final version, where all the inputs came from prev_out2 and no change is given.

Now that a basic understanding of transactions has been laid out, I will dive into how it's actually done. In the real Bitcoin protocol, a transaction is formatted as follows:

tx

Field Size	Description	Data type	Comments
4	version	uint32_t	Transaction data format version
1+	tx_in count	var_int	Number of Transaction inputs
41+	tx_in	tx_in[]	A list of 1 or more transaction inputs or sources for coins
1+	tx_out count	var_int	Number of Transaction outputs
9+	tx_out	tx_out[]	A list of 1 or more transaction outputs or destinations for coins
4	lock_time	uint32_t	The block number or timestamp at which this transaction is locked: If all TxIn inputs have final (0xffffffff) sequence numbers then lock_time is irrelevant. Otherwise, the transaction may not be added to a block until after lock_time (see NLockTime).

Where tx_in and tx_out objects are defined by the following tables:

tx_in

Field Size	Description	Data type	Comments
36	previous_output	outpoint	The previous output transaction reference, as an OutPoint structure (Defined on the next page)
1+	script length	var_int	The length of the signature script
?	signature script	uchar[]	Computational Script for confirming transaction authorization
4	sequence	uint32_t	Transaction version as defined by the sender. Intended for "replacement" of transactions when information is updated before inclusion into a block.

tx_out

Field Size	Description	Data type	Comments
8	value	int64_t	Transaction Value
1+	pk_script length	var_int	Length of the pk_script
?	pk_script	uchar[]	Usually contains the public key as a Bitcoin script setting up conditions to claim this output.

There are a few new things in these tables that I have not yet mentioned. First, a "lock_time" is something that can be used to delay the verification of a transaction. In other words, you can actually give your payment a longer process time if you wish. Typically though, this is set such that the transaction is immediate.

Another new object is the outpoint object located in the datatype field of previous_output in the tx_in table. I have already described something similar in my JSON example above except I called it an "input". An outpoint object is defined by the following table:

Field Size	Description	Data type	Comments
32	hash	char[32]	The hash of the referenced transaction.
4	index	uint32_t	The index of the specific output in the transaction. The first output is 0, etc.

This is nearly identical to the "input" object I used. Although, rather than a simple decimal number, previous outputs are referenced by their unique hash. The index field is exactly the same as the previous example, describing which output from the previous transaction this input is coming from.

Finally, these tables include fields that describe "script". In addition to the protocol itself, Bitcoin includes its own scripting language. This leaves Bitcoin open for extension. The idea is that transactions can include specific scripts within the script field, describing how this transaction is to be processed. Bitcoin script is very simple, and is best described by the following: "A script is essentially a list of instructions recorded with each transaction that describe how the next person wanting to spend the Bitcoins being transferred can gain access to them. The script for a typical Bitcoin transfer to destination Bitcoin address D simply encumbers future spending of the bitcoins with two things: the spender must provide

1. a public key that, when hashed, yields destination address D embedded in the script, and
2. a signature to show evidence of the private key corresponding to the public key just provided (Protocol Documentation)." This allows the parameters to spend bitcoin to vary, i.e you could modify the script to require more than just a signature. The following depicts this process.

Alice: creates function **f** (**x, y, z**) = true or false

Alice: includes function f in her transaction **tx**

Eve: receives payment from alice

Eve: goes to spend the bitcoins received from alice

Eve: must provide the correct parameters **x,y,z** to **f** and get a result of true in order to spend the bitcoin.

When you take all of these attributes and put them together you get a bitcoin transaction. And, now that all of the components have been described, we can assemble a complete transaction. Below is the JSON representation of a real transaction.

```
{"Example Transaction": {
        "version":1,
        "tx_in count": 2,
        "tx_in":  [
                { "script length": 0x00FA,
                  {"outpoint":{
                        "hash": "2007ae...",
                        "index": 0
                  }},
                  "scriptSig": "304502... 042b2d…"  },
                { "script length": 0x00FA,
                  {"outpoint":{
                        "hash": "3beabc...",
                        "index": 0
                  }},
                  "scriptSig": "304402... 038a52..."  },
                ],
        "tx_out count": 1,
        "tx_out":  [
                {  "value": .6,
                   "pk_script length": 0x0E0F
                   "pk_script": "OP_DUP OP_HASH160 e8c306... OP_EQUALVERIFY
                                OP_CHECKSIG"
                } ],
        "lock_time": 0,
}}
```

As before, this transaction includes where the money came from, and where it is going. Although, this time, it also includes the signatures and a signature script.

Conclusion:

This completes my description of bitcoin and the Bitcoin protocol. I hope this document has improved your understanding of this new and powerful technology. There is much more to Bitcoin than I have covered. It's a revolutionary new technology, and we have only scratched the surface. The future holds many new uses for this technology, and I cant wait to see what we make of it.

Works Cited:

"Bitcoin-labs/bitcoin-details." GitHub. Web. 14 May 2015. <https://github.com/bitcoin-labs/bitcoin-details/blob/master/bitcoin-capture1.pcap>.

bobalot. ”Re: What Is the Difference between an Address and a Public Key? • /r/Bitcoin." Reddit. Web. 6 May 2015.

"Hashcash." - Bitcoin Wiki. Web. 14 May 2015. <https://en.bitcoin.it/wiki/Hashcash>.

"Hashcash." Wikipedia. Wikimedia Foundation. Web. 14 May 2015.

"Merkle Tree." Wikipedia. Wikimedia Foundation. Web. 14 May 2015.

Nielsen, Michael. "How the Bitcoin Protocol Actually Works." DDI. 6 Dec. 2013. Web. 3 Apr. 2015.

"Protocol Documentation." - Bitcoin. Web. 6 May 2015. <https://en.bitcoin.it/wiki/Protocol_specification#Message_structure>.

"Script." - Bitcoin Wiki. Web. 14 May 2015. <https://en.bitcoin.it/wiki/Script>.