

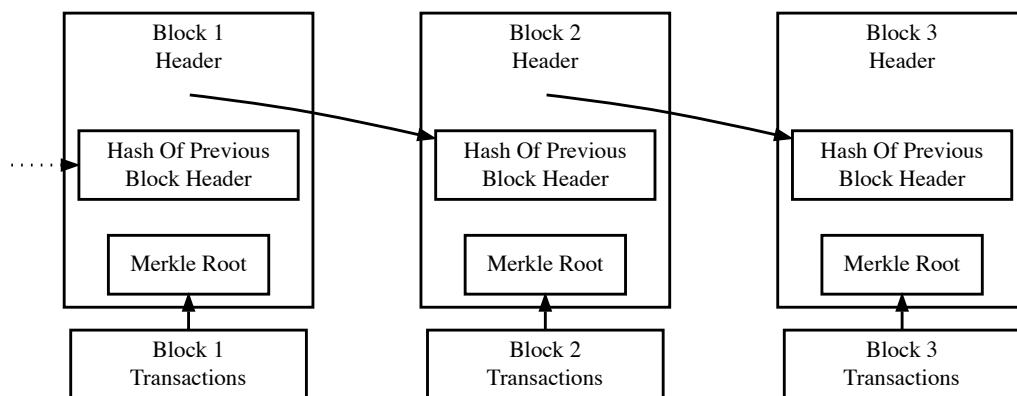
Block Chain Guide

Block Chain

The block chain provides Bitcoin's public ledger, an ordered and timestamped record of transactions. This system is used to protect against double spending and modification of previous transaction records.

Each full node in the Bitcoin network independently stores a block chain containing only blocks validated by that node. When several nodes all have the same blocks in their block chain, they are considered to be in **consensus**. The validation rules these nodes follow to maintain consensus are called **consensus rules**. This section describes many of the consensus rules used by Bitcoin Core.

Block Chain Overview

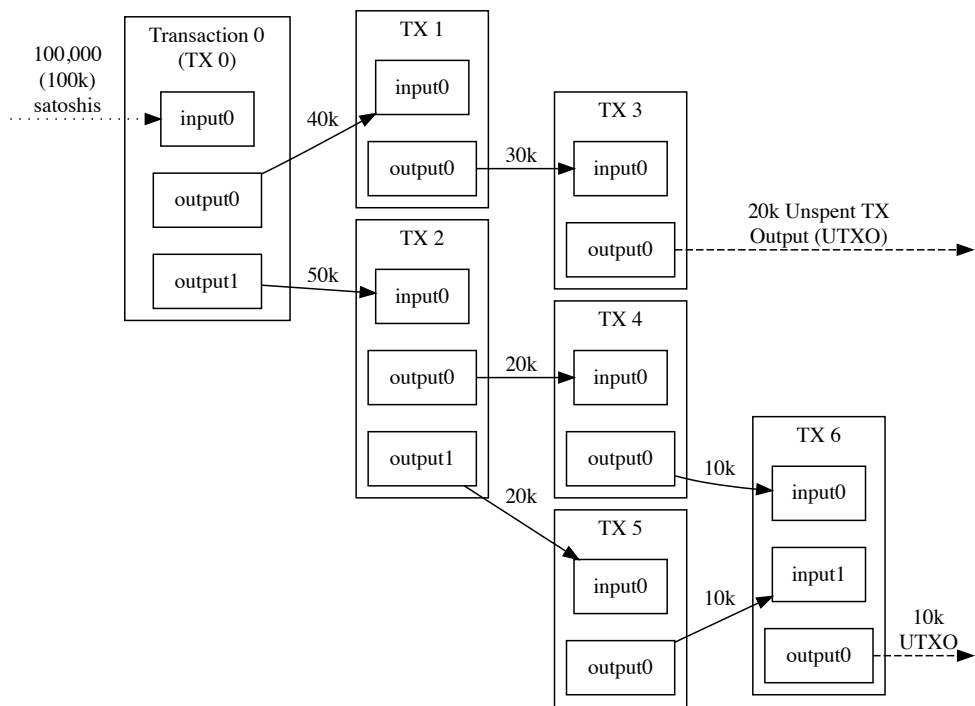


Simplified Bitcoin Block Chain

The illustration above shows a simplified version of a block chain. A **block** of one or more new transactions is collected into the transaction data part of a block. Copies of each transaction are hashed, and the hashes are then paired, hashed, paired again, and hashed again until a single hash remains, the **merkle root** of a merkle tree.

The merkle root is stored in the block header. Each block also stores the hash of the previous block's header, chaining the blocks together. This ensures a transaction cannot be modified without modifying the block that records it and all following blocks.

Transactions are also chained together. Bitcoin wallet software gives the impression that satoshis are sent from and to wallets, but bitcoins really move from transaction to transaction. Each transaction spends the satoshis previously received in one or more earlier transactions, so the input of one transaction is the output of a previous transaction.



Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

A single transaction can create multiple outputs, as would be the case when sending to multiple addresses, but each output of a particular transaction can only be used as an input once in the block chain. Any subsequent reference is a forbidden double spend—an attempt to spend the same satoshis twice.

Outputs are tied to **transaction identifiers (TXIDs)**, which are the hashes of signed transactions.

Because each output of a particular transaction can only be spent once, the outputs of all transactions included in the block chain can be categorized as either **Unspent Transaction Outputs (UTXOs)** or spent transaction outputs. For a payment to be valid, it must only use UTXOs as inputs.

Ignoring coinbase transactions (described later), if the value of a transaction's outputs exceed its inputs, the transaction will be rejected—but if the inputs exceed the value of the outputs, any difference in value may be claimed as a **transaction fee** by the Bitcoin **miner** who creates the block containing that transaction. For example, in the illustration above, each transaction spends 10,000 satoshis fewer than it receives from its combined inputs, effectively paying a 10,000 satoshi transaction fee.

Proof Of Work

The block chain is collaboratively maintained by anonymous peers on the network, so Bitcoin requires that each block prove a significant amount of work was invested in its creation to ensure that untrustworthy peers who want to modify past blocks have to work harder than honest peers who only want to add new blocks to the block chain.

Chaining blocks together makes it impossible to modify transactions included in any block without modifying all following blocks. As a result, the cost to modify a particular block increases with every new block added to the block chain, magnifying the effect of

the proof of work.

The **proof of work** used in Bitcoin takes advantage of the apparently random nature of cryptographic hashes. A good cryptographic hash algorithm converts arbitrary data into a seemingly-random number. If the data is modified in any way and the hash re-run, a new seemingly-random number is produced, so there is no way to modify the data to make the hash number predictable.

To prove you did some extra work to create a block, you must create a hash of the block header which does not exceed a certain value. For example, if the maximum possible hash value is $2^{256} - 1$, you can prove that you tried up to two combinations by producing a hash value less than 2^{255} .

In the example given above, you will produce a successful hash on average every other try. You can even estimate the probability that a given hash attempt will generate a number below the **target** threshold. Bitcoin assumes a linear probability that the lower it makes the target threshold, the more hash attempts (on average) will need to be tried.

New blocks will only be added to the block chain if their hash is at least as challenging as a **difficulty** value expected by the consensus protocol. Every 2,016 blocks, the network uses timestamps stored in each block header to calculate the number of seconds elapsed between generation of the first and last of those last 2,016 blocks. The ideal value is 1,209,600 seconds (two weeks).

- If it took fewer than two weeks to generate the 2,016 blocks, the expected difficulty value is increased proportionally (by as much as 300%) so that the next 2,016 blocks should take exactly two weeks to generate if hashes are checked at the same rate.
- If it took more than two weeks to generate the blocks, the expected difficulty value is decreased proportionally (by as much as 75%) for the same reason.

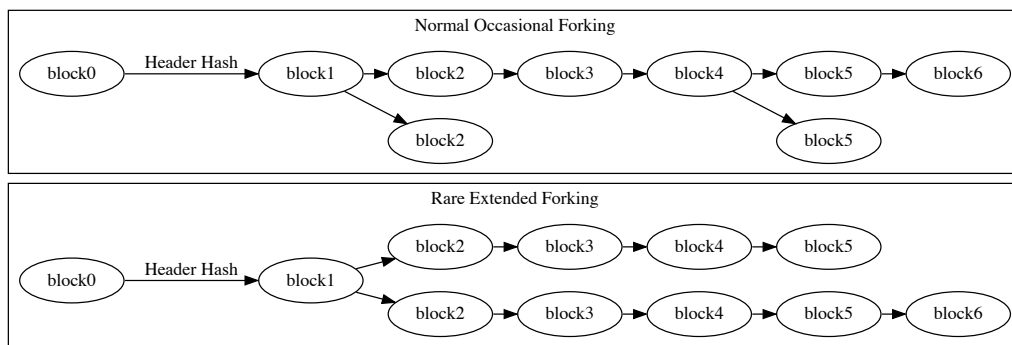
(Note: an off-by-one error in the Bitcoin Core implementation causes the difficulty to be updated every 2,016 blocks using timestamps from only 2,015 blocks, creating a slight skew.)

Because each block header must hash to a value below the target threshold, and because each block is linked to the block that preceded it, it requires (on average) as much hashing power to propagate a modified block as the entire Bitcoin network expended between the time the original block was created and the present time. Only if you acquired a majority of the network's hashing power could you reliably execute such a **51 percent attack** against transaction history (although, it should be noted, that even less than 50% of the hashing power still has a good chance of performing such attacks).

The block header provides several easy-to-modify fields, such as a dedicated nonce field, so obtaining new hashes doesn't require waiting for new transactions. Also, only the 80-byte block header is hashed for proof-of-work, so including a large volume of transaction data in a block does not slow down hashing with extra I/O, and adding additional transaction data only requires the recalculation of the ancestor hashes in the merkle tree.

Block Height And Forking

Any Bitcoin miner who successfully hashes a block header to a value below the target threshold can add the entire block to the block chain (assuming the block is otherwise valid). These blocks are commonly addressed by their **block height**—the number of blocks between them and the first Bitcoin block (block 0, most commonly known as the **genesis block**). For example, block 2016 is where difficulty could have first been adjusted.



Multiple blocks can all have the same block height, as is common when two or more miners each produce a block at roughly the same time. This creates an apparent **fork** in the block chain, as shown in the illustration above.

When miners produce simultaneous blocks at the end of the block chain, each node individually chooses which block to accept. In the absence of other considerations, discussed below, nodes usually use the first block they see.

Eventually a miner produces another block which attaches to only one of the competing simultaneously-mined blocks. This makes that side of the fork stronger than the other side. Assuming a fork only contains valid blocks, normal peers always follow the the most difficult chain to recreate and throw away **stale blocks** belonging to shorter forks. (Stale blocks are also sometimes called orphans or orphan blocks, but those terms are also used for true orphan blocks without a known parent block.)

Long-term forks are possible if different miners work at cross-purposes, such as some miners diligently working to extend the block chain at the same time other miners are attempting a 51 percent attack to revise transaction history.

Since multiple blocks can have the same height during a block chain fork, block height should not be used as a globally unique identifier. Instead, blocks are usually referenced by the hash of their header (often with the byte order reversed, and in hexadecimal).

Transaction Data

Every block must include one or more transactions. The first one of these transactions must be a coinbase transaction, also called a generation transaction, which should collect and spend the block reward (comprised of a block subsidy and any transaction fees paid by transactions included in this block).

The UTXO of a coinbase transaction has the special condition that it cannot be spent (used as an input) for at least 100 blocks. This temporarily prevents a miner from spending the transaction fees and block reward from a block that may later be determined to be stale (and therefore the coinbase transaction destroyed) after a block chain fork.

Blocks are not required to include any non-coinbase transactions, but miners almost always do include additional transactions in order to collect their transaction fees.

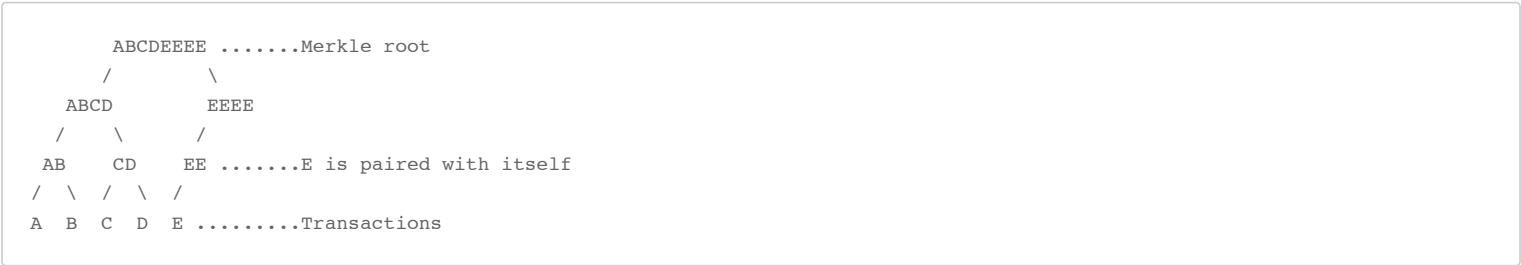
All transactions, including the coinbase transaction, are encoded into blocks in binary rawtransaction format.

The rawtransaction format is hashed to create the transaction identifier (txid). From these txids, the **merkle tree** is constructed by pairing each txid with one other txid and then hashing them together. If there are an odd number of txids, the txid without a partner is hashed with a copy of itself.

The resulting hashes themselves are each paired with one other hash and hashed together. Any hash without a partner is hashed with itself. The process repeats until only one hash remains, the merkle root.

For example, if transactions were merely joined (not hashed), a five-transaction merkle tree would look like the following text

diagram:



As discussed in the Simplified Payment Verification (SPV) subsection, the merkle tree allows clients to verify for themselves that a transaction was included in a block by obtaining the merkle root from a block header and a list of the intermediate hashes from a full peer. The full peer does not need to be trusted: it is expensive to fake block headers and the intermediate hashes cannot be faked or the verification will fail.

For example, to verify transaction D was added to the block, an SPV client only needs a copy of the C, AB, and EEEE hashes in addition to the merkle root; the client doesn’t need to know anything about any of the other transactions. If the five transactions in this block were all at the maximum size, downloading the entire block would require over 500,000 bytes—but downloading three hashes plus the block header requires only 140 bytes.

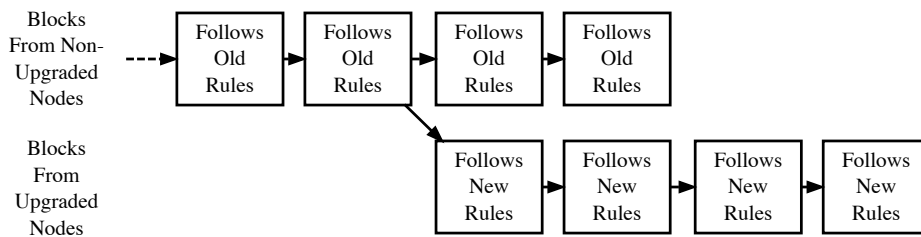
Note: If identical txids are found within the same block, there is a possibility that the merkle tree may collide with a block with some or all duplicates removed due to how unbalanced merkle trees are implemented (duplicating the lone hash). Since it is impractical to have separate transactions with identical txids, this does not impose a burden on honest software, but must be checked if the invalid status of a block is to be cached; otherwise, a valid block with the duplicates eliminated could have the same merkle root and block hash, but be rejected by the cached invalid outcome, resulting in security bugs such as CVE-2012-2459.

Consensus Rule Changes

To maintain consensus, all full nodes validate blocks using the same consensus rules. However, sometimes the consensus rules are changed to introduce new features or prevent network abuse. When the new rules are implemented, there will likely be a period of time when non-upgraded nodes follow the old rules and upgraded nodes follow the new rules, creating two possible ways consensus can break:

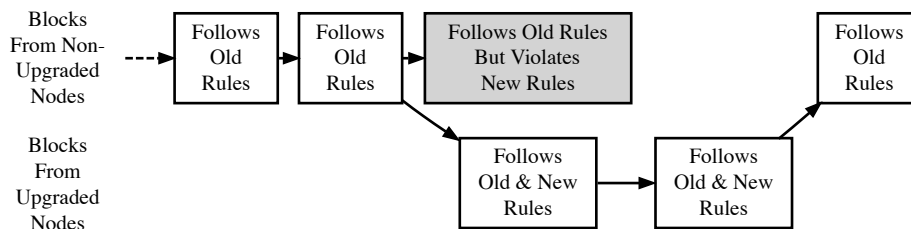
1. A block following the new consensus rules is accepted by upgraded nodes but rejected by non-upgraded nodes. For example, a new transaction feature is used within a block: upgraded nodes understand the feature and accept it, but non-upgraded nodes reject it because it violates the old rules.
2. A block violating the new consensus rules is rejected by upgraded nodes but accepted by non-upgraded nodes. For example, an abusive transaction feature is used within a block: upgraded nodes reject it because it violates the new rules, but non-upgraded nodes accept it because it follows the old rules.

In the first case, rejection by non-upgraded nodes, mining software which gets block chain data from those non-upgraded nodes refuses to build on the same chain as mining software getting data from upgraded nodes. This creates permanently divergent chains—one for non-upgraded nodes and one for upgraded nodes—called a **hard fork**.



A Hard Fork: Non-Upgraded Nodes Reject The New Rules, Diverging The Chain

In the second case, rejection by upgraded nodes, it's possible to keep the block chain from permanently diverging if upgraded nodes control a majority of the hash rate. That's because, in this case, non-upgraded nodes will accept as valid all the same blocks as upgraded nodes, so the upgraded nodes can build a stronger chain that the non-upgraded nodes will accept as the best valid block chain. This is called a **soft fork**.



A Soft Fork: Blocks Violating New Rules Are Made Stale By The Upgraded Mining Majority

Although a fork is an actual divergence in block chains, changes to the consensus rules are often described by their potential to create either a hard or soft fork. For example, “increasing the block size above 1 MB requires a hard fork.” In this example, an actual block chain fork is not required—but it is a possible outcome.

Resources: [BIP16](#), [BIP30](#), and [BIP34](#) were implemented as changes which might have lead to soft forks. [BIP50](#) describes both an accidental hard fork, resolved by temporary downgrading the capabilities of upgraded nodes, and an intentional hard fork when the temporary downgrade was removed. A document from Gavin Andresen outlines [how future rule changes may be implemented](#).

Detecting Forks

Non-upgraded nodes may use and distribute incorrect information during both types of forks, creating several situations which could lead to financial loss. In particular, non-upgraded nodes may relay and accept transactions that are considered invalid by upgraded nodes and so will never become part of the universally-recognized best block chain. Non-upgraded nodes may also refuse to relay blocks or transactions which have already been added to the best block chain, or soon will be, and so provide incomplete information.

Bitcoin Core includes code that detects a hard fork by looking at block chain proof of work. If a non-upgraded node receives block chain headers demonstrating at least six blocks more proof of work than the best chain it considers valid, the node reports an error in the `getinfo` RPC results and runs the `-alertnotify` command if set. This warns the operator that the non-upgraded node can't switch to what is likely the best block chain.

Full nodes can also check block and transaction version numbers. If the block or transaction version numbers seen in several recent blocks are higher than the version numbers the node uses, it can assume it doesn't use the current consensus rules. Bitcoin Core 0.10.0 reports this situation through the `getinfo` RPC and `-alertnotify` command if set.

In either case, block and transaction data should not be relied upon if it comes from a node that apparently isn't using the current

consensus rules.

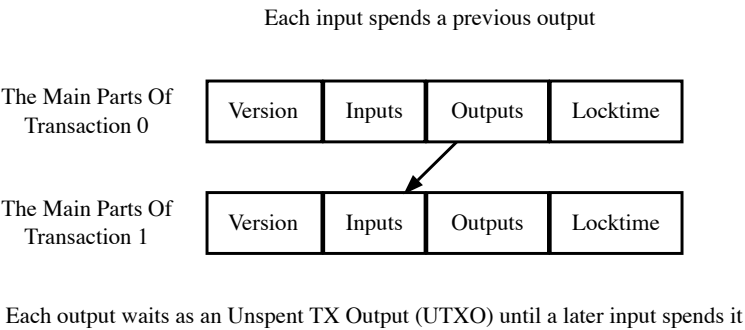
SPV clients which connect to full nodes can detect a likely hard fork by connecting to several full nodes and ensuring that they're all on the same chain with the same block height, plus or minus several blocks to account for transmission delays and stale blocks. If there's a divergence, the client can disconnect from nodes with weaker chains.

SPV clients should also monitor for block and transaction version number increases to ensure they process received transactions and create new transactions using the current consensus rules.

Transactions

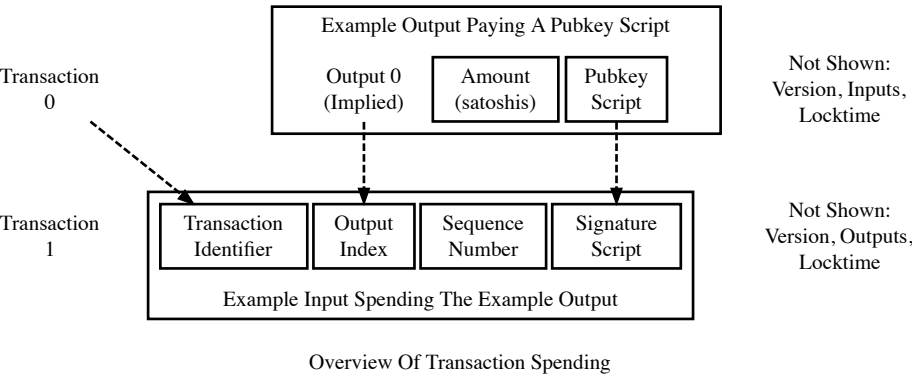
Transactions let users spend satoshis. Each transaction is constructed out of several parts which enable both simple direct payments and complex transactions. This section will describe each part and demonstrate how to use them together to build complete transactions.

To keep things simple, this section pretends coinbase transactions do not exist. Coinbase transactions can only be created by Bitcoin miners and they're an exception to many of the rules listed below. Instead of pointing out the coinbase exception to each rule, we invite you to read about coinbase transactions in the block chain section of this guide.



The figure above shows the main parts of a Bitcoin transaction. Each transaction has at least one input and one output. Each **input** spends the satoshis paid to a previous output. Each **output** then waits as an Unspent Transaction Output (UTXO) until a later input spends it. When your Bitcoin wallet tells you that you have a 10,000 satoshi balance, it really means that you have 10,000 satoshis waiting in one or more UTXOs.

Each transaction is prefixed by a four-byte **transaction version number** which tells Bitcoin peers and miners which set of rules to use to validate it. This lets developers create new rules for future transactions without invalidating previous transactions.

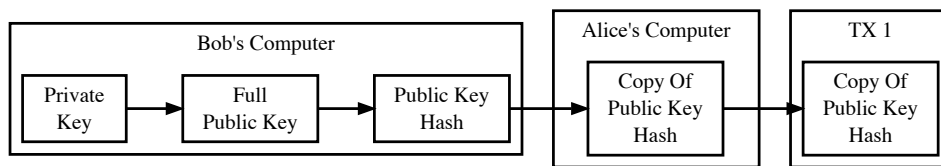


An output has an implied index number based on its location in the transaction—the first output is output zero. The output also

has an amount in satoshis which it pays to a conditional pubkey script. Anyone who can satisfy the conditions of that pubkey script can spend up to the amount of satoshis paid to it.

An input uses a transaction identifier (txid) and an output index number (often called “vout” for output vector) to identify a particular output to be spent. It also has a signature script which allows it to provide data parameters that satisfy the conditionals in the pubkey script. (The sequence number and locktime are related and will be covered together in a later subsection.)

The figures below help illustrate how these features are used by showing the workflow Alice uses to send Bob a transaction and which Bob later uses to spend that transaction. Both Alice and Bob will use the most common form of the standard Pay-To-Public-Key-Hash (P2PKH) transaction type. **P2PKH** lets Alice spend satoshis to a typical Bitcoin address, and then lets Bob further spend those satoshis using a simple cryptographic key pair.



Creating A P2PKH Public Key Hash To Receive Payment

Bob must first generate a private/public **key pair** before Alice can create the first transaction. Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) with the secp256k1 curve; secp256k1 **private keys** are 256 bits of random data. A copy of that data is deterministically transformed into an secp256k1 **public key**. Because the transformation can be reliably repeated later, the public key does not need to be stored.

The public key (pubkey) is then cryptographically hashed. This pubkey hash can also be reliably repeated later, so it also does not need to be stored. The hash shortens and obfuscates the public key, making manual transcription easier and providing security against unanticipated problems which might allow reconstruction of private keys from public key data at some later point.

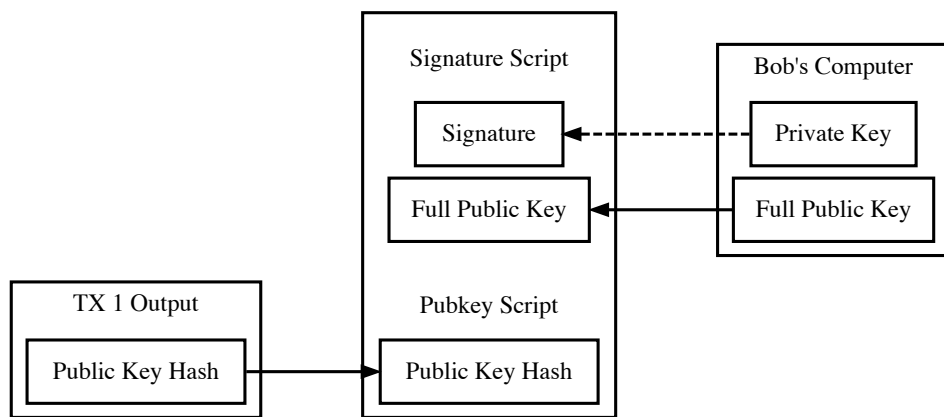
Bob provides the pubkey hash to Alice. Pubkey hashes are almost always sent encoded as Bitcoin **addresses**, which are base58-encoded strings containing an address version number, the hash, and an error-detection checksum to catch typos. The address can be transmitted through any medium, including one-way mediums which prevent the spender from communicating with the receiver, and it can be further encoded into another format, such as a QR code containing a `bitcoin:` URI.

Once Alice has the address and decodes it back into a standard hash, she can create the first transaction. She creates a standard P2PKH transaction output containing instructions which allow anyone to spend that output if they can prove they control the private key corresponding to Bob’s hashed public key. These instructions are called the **pubkey script** or `scriptPubKey`.

Alice broadcasts the transaction and it is added to the block chain. The network categorizes it as an Unspent Transaction Output (UTXO), and Bob’s wallet software displays it as a spendable balance.

When, some time later, Bob decides to spend the UTXO, he must create an input which references the transaction Alice created by its hash, called a Transaction Identifier (txid), and the specific output she used by its index number (**output index**). He must then create a **signature script**—a collection of data parameters which satisfy the conditions Alice placed in the previous output’s pubkey script. Signature scripts are also called `scriptSigs`.

Pubkey scripts and signature scripts combine secp256k1 pubkeys and signatures with conditional logic, creating a programmable authorization mechanism.

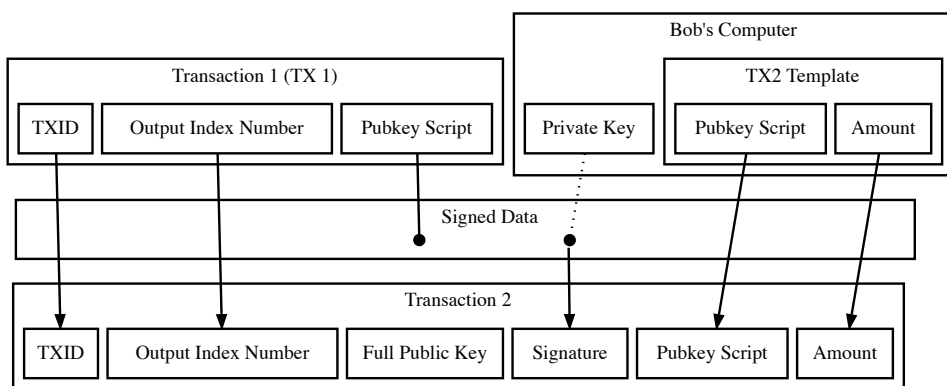


Spending A P2PKH Output

For a P2PKH-style output, Bob's signature script will contain the following two pieces of data:

1. His full (unhashed) public key, so the pubkey script can check that it hashes to the same value as the pubkey hash provided by Alice.
2. An secp256k1 **signature** made by using the ECDSA cryptographic formula to combine certain transaction data (described below) with Bob's private key. This lets the pubkey script verify that Bob owns the private key which created the public key.

Bob's secp256k1 signature doesn't just prove Bob controls his private key; it also makes the non-signature-script parts of his transaction tamper-proof so Bob can safely broadcast them over the peer-to-peer network.



Some Of The Data Signed By Default

As illustrated in the figure above, the data Bob signs includes the txid and output index of the previous transaction, the previous output's pubkey script, the pubkey script Bob creates which will let the next recipient spend this transaction's output, and the amount of satoshis to spend to the next recipient. In essence, the entire transaction is signed except for any signature scripts, which hold the full public keys and secp256k1 signatures.

After putting his signature and public key in the signature script, Bob broadcasts the transaction to Bitcoin miners through the peer-to-peer network. Each peer and miner independently validates the transaction before broadcasting it further or attempting to include it in a new block of transactions.

P2PKH Script Validation

The validation procedure requires evaluation of the signature script and pubkey script. In a P2PKH output, the pubkey script is:

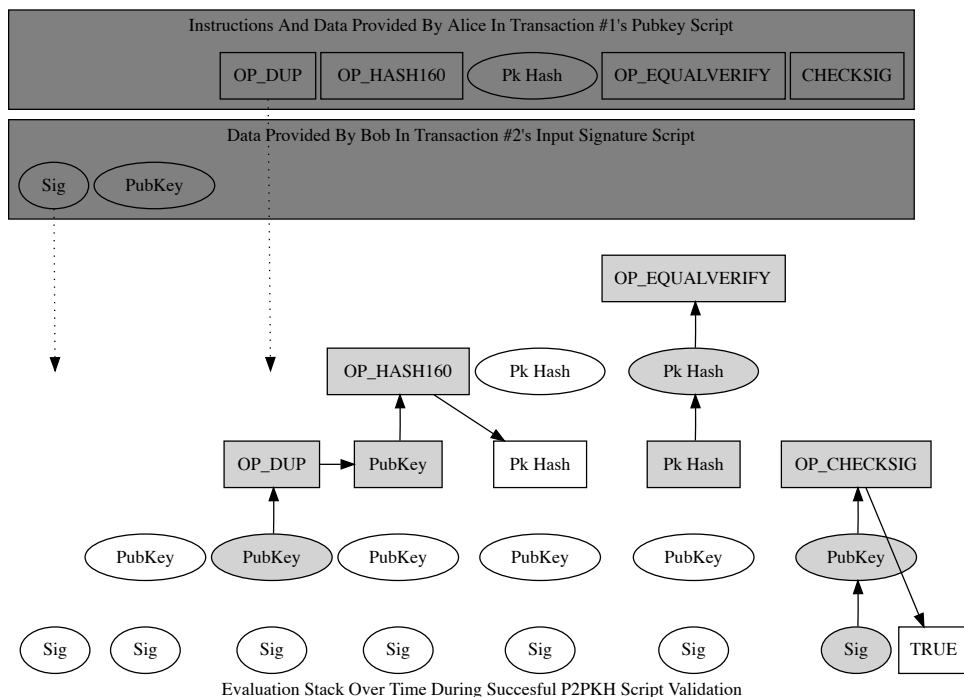
```
OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

The spender's signature script is evaluated and prefixed to the beginning of the script. In a P2PKH transaction, the signature script contains an secp256k1 signature (sig) and full public key (pubkey), creating the following concatenation:

```
<Sig> <PubKey> OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

The script language is a [Forth-like](#) stack-based language deliberately designed to be stateless and not Turing complete. Statelessness ensures that once a transaction is added to the block chain, there is no condition which renders it permanently unspendable. Turing-incompleteness (specifically, a lack of loops or gotos) makes the script language less flexible and more predictable, greatly simplifying the security model.

To test whether the transaction is valid, signature script and pubkey script operations are executed one item at a time, starting with Bob's signature script and continuing to the end of Alice's pubkey script. The figure below shows the evaluation of a standard P2PKH pubkey script; below the figure is a description of the process.



- The signature (from Bob's signature script) is added (pushed) to an empty stack. Because it's just data, nothing is done except adding it to the stack. The public key (also from the signature script) is pushed on top of the signature.
- From Alice's pubkey script, the `OP_DUP` operation is executed. `OP_DUP` pushes onto the stack a copy of the data currently at the top of it—in this case creating a copy of the public key Bob provided.
- The operation executed next, `OP_HASH160`, pushes onto the stack a hash of the data currently on top of it—in this case, Bob's public key. This creates a hash of Bob's public key.
- Alice's pubkey script then pushes the pubkey hash that Bob gave her for the first transaction. At this point, there should be two copies of Bob's pubkey hash at the top of the stack.
- Now it gets interesting: Alice's pubkey script executes `OP_EQUALVERIFY`. `OP_EQUALVERIFY` is equivalent to executing `OP_EQUAL` followed by `OP_VERIFY` (not shown).

`OP_EQUAL` (not shown) checks the two values at the top of the stack; in this case, it checks whether the pubkey hash generated from the full public key Bob provided equals the pubkey hash Alice provided when she created transaction #1.

`OP_EQUAL` pops (removes from the top of the stack) the two values it compared, and replaces them with the result of that comparison: zero (*false*) or one (*true*).

`OP_VERIFY` (not shown) checks the value at the top of the stack. If the value is *false* it immediately terminates evaluation and the transaction validation fails. Otherwise it pops the *true* value off the stack.

- Finally, Alice's pubkey script executes `OP_CHECKSIG`, which checks the signature Bob provided against the now-authenticated public key he also provided. If the signature matches the public key and was generated using all of the data required to be signed, `OP_CHECKSIG` pushes the value *true* onto the top of the stack.

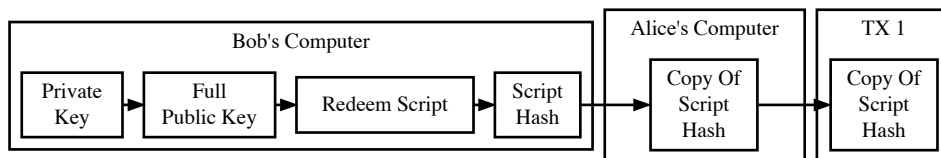
If *false* is not at the top of the stack after the pubkey script has been evaluated, the transaction is valid (provided there are no other problems with it).

P2SH Scripts

Pubkey scripts are created by spenders who have little interest what that script does. Receivers do care about the script conditions and, if they want, they can ask spenders to use a particular pubkey script. Unfortunately, custom pubkey scripts are less convenient than short Bitcoin addresses and there was no standard way to communicate them between programs prior to widespread implementation of the BIP70 Payment Protocol discussed later.

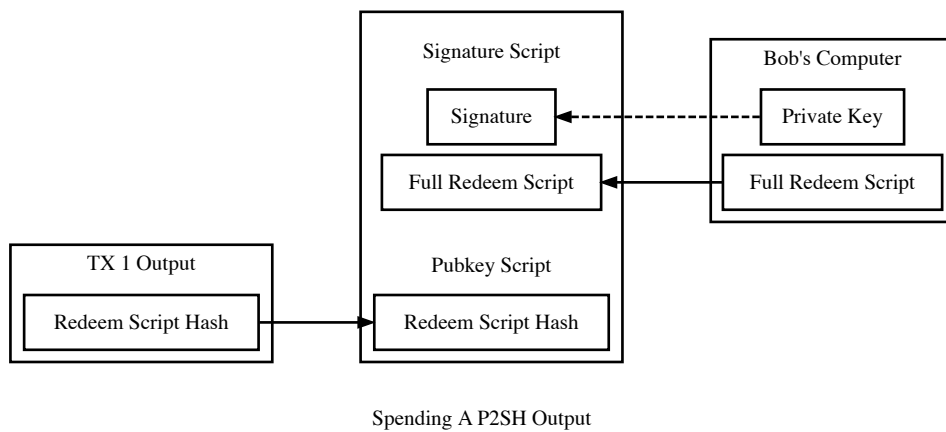
To solve these problems, pay-to-script-hash (**P2SH**) transactions were created in 2012 to let a spender create a pubkey script containing a hash of a second script, the **redeem script**.

The basic P2SH workflow, illustrated below, looks almost identical to the P2PKH workflow. Bob creates a redeem script with whatever script he wants, hashes the redeem script, and provides the redeem script hash to Alice. Alice creates a P2SH-style output containing Bob's redeem script hash.



Creating A P2SH Redeem Script Hash To Receive Payment

When Bob wants to spend the output, he provides his signature along with the full (serialized) redeem script in the signature script. The peer-to-peer network ensures the full redeem script hashes to the same value as the script hash Alice put in her output; it then processes the redeem script exactly as it would if it were the primary pubkey script, letting Bob spend the output if the redeem script does not return false.



The hash of the redeem script has the same properties as a pubkey hash—so it can be transformed into the standard Bitcoin address format with only one small change to differentiate it from a standard address. This makes collecting a P2SH-style address as simple as collecting a P2PKH-style address. The hash also obfuscates any public keys in the redeem script, so P2SH scripts are as secure as P2PKH pubkey hashes.

Standard Transactions

After the discovery of several dangerous bugs in early versions of Bitcoin, a test was added which only accepted transactions from the network if their pubkey scripts and signature scripts matched a small set of believed-to-be-safe templates, and if the rest of the transaction didn't violate another small set of rules enforcing good network behavior. This is the `IsStandard()` test, and transactions which pass it are called standard transactions.

Non-standard transactions—those that fail the test—may be accepted by nodes not using the default Bitcoin Core settings. If they are included in blocks, they will also avoid the `IsStandard` test and be processed.

Besides making it more difficult for someone to attack Bitcoin for free by broadcasting harmful transactions, the standard transaction test also helps prevent users from creating transactions today that would make adding new transaction features in the future more difficult. For example, as described above, each transaction includes a version number—if users started arbitrarily changing the version number, it would become useless as a tool for introducing backwards-incompatible features.

As of Bitcoin Core 0.9, the standard pubkey script types are:

Pay To Public Key Hash (P2PKH)

P2PKH is the most common form of pubkey script used to send a transaction to one or multiple Bitcoin addresses.

```
Pubkey script: OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
Signature script: <sig> <pubkey>
```

Pay To Script Hash (P2SH)

P2SH is used to send a transaction to a script hash. Each of the standard pubkey scripts can be used as a P2SH redeem script, but in practice only the multisig pubkey script makes sense until more transaction types are made standard.

```
Pubkey script: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
Signature script: <sig> [sig] [sig...] <redeemScript>
```

Multisig

Although P2SH multisig is now generally used for multisig transactions, this base script can be used to require multiple signatures before a UTXO can be spent.

In multisig pubkey scripts, called m-of-n, *m* is the *minimum* number of signatures which must match a public key; *n* is the *number* of public keys being provided. Both *m* and *n* should be op codes `OP_1` through `OP_16`, corresponding to the number desired.

Because of an off-by-one error in the original Bitcoin implementation which must be preserved for compatibility, `OP_CHECKMULTISIG` consumes one more value from the stack than indicated by *m*, so the list of secp256k1 signatures in the signature script must be prefaced with an extra value (`OP_0`) which will be consumed but not used.

The signature script must provide signatures in the same order as the corresponding public keys appear in the pubkey script or redeem script. See the description in `OP_CHECKMULTISIG` for details.

```
Pubkey script: <m> <A pubkey> [B pubkey] [C pubkey...] <n> OP_CHECKMULTISIG
Signature script: OP_0 <A sig> [B sig] [C sig...]
```

Although it's not a separate transaction type, this is a P2SH multisig with 2-of-3:

```
Pubkey script: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
Redeem script: <OP_2> <A pubkey> <B pubkey> <C pubkey> <OP_3> OP_CHECKMULTISIG
Signature script: OP_0 <A sig> <C sig> <redeemScript>
```

Pubkey

Pubkey outputs are a simplified form of the P2PKH pubkey script, but they aren't as secure as P2PKH, so they generally aren't used in new transactions anymore.

```
Pubkey script: <pubkey> OP_CHECKSIG
Signature script: <sig>
```

Null Data

Null data pubkey scripts let you add a small amount of arbitrary data to the block chain in exchange for paying a transaction fee, but doing so is discouraged. (Null data is a standard pubkey script type only because some people were adding data to the block chain in more harmful ways.)

```
Pubkey Script: OP_RETURN <0 to 40 bytes of data>
(Null data scripts cannot be spent, so there's no signature script.)
```

Non-Standard Transactions

If you use anything besides a standard pubkey script in an output, peers and miners using the default Bitcoin Core settings will neither accept, broadcast, nor mine your transaction. When you try to broadcast your transaction to a peer running the default settings, you will receive an error.

If you create a redeem script, hash it, and use the hash in a P2SH output, the network sees only the hash, so it will accept the output as valid no matter what the redeem script says. This allows payment to non-standard pubkey script almost as easily as payment to standard pubkey scripts. However, when you go to spend that output, peers and miners using the default settings will check the redeem script to see whether or not it's a standard pubkey script. If it isn't, they won't process it further—so it will be impossible to spend that output until you find a miner who disables the default settings.

Note: standard transactions are designed to protect and help the network, not prevent you from making mistakes. It's easy to create standard transactions which make the satoshis sent to them unspendable.

As of Bitcoin Core 0.9.3, standard transactions must also meet the following conditions:

- The transaction must be finalized: either its locktime must be in the past (or less than or equal to the current block height), or all of its sequence numbers must be 0xffffffff.
- The transaction must be smaller than 100,000 bytes. That's around 200 times larger than a typical single-input, single-output P2PKH transaction.
- Each of the transaction's signature scripts must be smaller than 1,650 bytes. That's large enough to allow 15-of-15 multisig transactions in P2SH using compressed public keys.
- Bare (non-P2SH) multisig transactions which require more than 3 public keys are currently non-standard.
- The transaction's signature script must only push data to the script evaluation stack. It cannot push new OP codes, with the exception of OP codes which solely push data to the stack.
- The transaction must not include any outputs which receive fewer than 1/3 as many satoshis as it would take to spend it in a typical input. That's [currently 546 satoshis](#) for a P2PKH or P2SH output on a Bitcoin Core node with the default relay fee. Exception: standard null data outputs must receive zero satoshis.

Signature Hash Types

`OP_CHECKSIG` extracts a non-stack argument from each signature it evaluates, allowing the signer to decide which parts of the transaction to sign. Since the signature protects those parts of the transaction from modification, this lets signers selectively choose to let other people modify their transactions.

The various options for what to sign are called **signature hash** types. There are three base SIGHASH types currently available:

- `SIGHASH_ALL`, the default, signs all the inputs and outputs, protecting everything except the signature scripts against modification.
- `SIGHASH_NONE` signs all of the inputs but none of the outputs, allowing anyone to change where the satoshis are going unless other signatures using other signature hash flags protect the outputs.
- `SIGHASH_SINGLE` the only output signed is the one corresponding to this input (the output with the same output index number as this input), ensuring nobody can change your part of the transaction but allowing other signers to change their part of the transaction. The corresponding output must exist or the value "1" will be signed, breaking the security scheme. This input, as

well as other inputs, are included in the signature. The sequence numbers of other inputs are not included in the signature, and can be updated.

The base types can be modified with the `SIGHASH_ANYONECANPAY` (anyone can pay) flag, creating three new combined types:

- `SIGHASH_ALL | SIGHASH_ANYONECANPAY` signs all of the outputs but only this one input, and it also allows anyone to add or remove other inputs, so anyone can contribute additional satoshis but they cannot change how many satoshis are sent nor where they go.
- `SIGHASH_NONE | SIGHASH_ANYONECANPAY` signs only this one input and allows anyone to add or remove other inputs or outputs, so anyone who gets a copy of this input can spend it however they'd like.
- `SIGHASH_SINGLE | SIGHASH_ANYONECANPAY` signs this one input and its corresponding output. Allows anyone to add or remove other inputs.

Because each input is signed, a transaction with multiple inputs can have multiple signature hash types signing different parts of the transaction. For example, a single-input transaction signed with `NONE` could have its output changed by the miner who adds it to the block chain. On the other hand, if a two-input transaction has one input signed with `NONE` and one input signed with `ALL`, the `ALL` signer can choose where to spend the satoshis without consulting the `NONE` signer—but nobody else can modify the transaction.

Locktime And Sequence Number

One thing all signature hash types sign is the transaction's **locktime**. (Called `nLockTime` in the Bitcoin Core source code.) The locktime indicates the earliest time a transaction can be added to the block chain.

Locktime allows signers to create time-locked transactions which will only become valid in the future, giving the signers a chance to change their minds.

If any of the signers change their mind, they can create a new non-locktime transaction. The new transaction will use, as one of its inputs, one of the same outputs which was used as an input to the locktime transaction. This makes the locktime transaction invalid if the new transaction is added to the block chain before the time lock expires.

Care must be taken near the expiry time of a time lock. The peer-to-peer network allows block time to be up to two hours ahead of real time, so a locktime transaction can be added to the block chain up to two hours before its time lock officially expires. Also, blocks are not created at guaranteed intervals, so any attempt to cancel a valuable transaction should be made a few hours before the time lock expires.

Previous versions of Bitcoin Core provided a feature which prevented transaction signers from using the method described above to cancel a time-locked transaction, but a necessary part of this feature was disabled to prevent denial of service attacks. A legacy of this system are four-byte **sequence numbers** in every input. Sequence numbers were meant to allow multiple signers to agree to update a transaction; when they finished updating the transaction, they could agree to set every input's sequence number to the four-byte unsigned maximum (`0xffffffff`), allowing the transaction to be added to a block even if its time lock had not expired.

Even today, setting all sequence numbers to `0xffffffff` (the default in Bitcoin Core) can still disable the time lock, so if you want to use locktime, at least one input must have a sequence number below the maximum. Since sequence numbers are not used by the network for any other purpose, setting any sequence number to zero is sufficient to enable locktime.

Locktime itself is an unsigned 4-byte integer which can be parsed two ways:

- If less than 500 million, locktime is parsed as a block height. The transaction can be added to any block which has this height

or higher.

- If greater than or equal to 500 million, locktime is parsed using the Unix epoch time format (the number of seconds elapsed since 1970-01-01T00:00 UTC—currently over 1.395 billion). The transaction can be added to any block whose block time is greater than the locktime.

Transaction Fees And Change

Transactions typically pay transaction fees based on the total byte size of the signed transaction. The transaction fee is given to the Bitcoin miner, as explained in the [block chain section](#), and so it is ultimately up to each miner to choose the minimum transaction fee they will accept.

By default, miners reserve 50 KB of each block for **high-priority transactions** which spend satoshis that haven't been spent for a long time. The remaining space in each block is typically allocated to transactions based on their fee per byte, with higher-paying transactions being added in sequence until all of the available space is filled.

As of Bitcoin Core 0.9, transactions which do not count as high-priority transactions need to pay a **minimum fee** (currently 1,000 satoshis) to be broadcast across the network. Any transaction paying only the minimum fee should be prepared to wait a long time before there's enough spare space in a block to include it. Please see the [verifying payment section](#) for why this could be important.

Since each transaction spends Unspent Transaction Outputs (UTXOs) and because a UTXO can only be spent once, the full value of the included UTXOs must be spent or given to a miner as a transaction fee. Few people will have UTXOs that exactly match the amount they want to pay, so most transactions include a change output.

Change outputs are regular outputs which spend the surplus satoshis from the UTXOs back to the spender. They can reuse the same P2PKH pubkey hash or P2SH script hash as was used in the UTXO, but for the reasons described in the [next subsection](#), it is highly recommended that change outputs be sent to a new P2PKH or P2SH address.

Avoiding Key Reuse

In a transaction, the spender and receiver each reveal to each other all public keys or addresses used in the transaction. This allows either person to use the public block chain to track past and future transactions involving the other person's same public keys or addresses.

If the same public key is reused often, as happens when people use Bitcoin addresses (hashed public keys) as static payment addresses, other people can easily track the receiving and spending habits of that person, including how many satoshis they control in known addresses.

It doesn't have to be that way. If each public key is used exactly twice—once to receive a payment and once to spend that payment—the user can gain a significant amount of financial privacy.

Even better, using new public keys or **unique addresses** when accepting payments or creating change outputs can be combined with other techniques discussed later, such as CoinJoin or merge avoidance, to make it extremely difficult to use the block chain by itself to reliably track how users receive and spend their satoshis.

Avoiding key reuse can also provide security against attacks which might allow reconstruction of private keys from public keys (hypothesized) or from signature comparisons (possible today under certain circumstances described below, with more general attacks hypothesized).

1. Unique (non-reused) P2PKH and P2SH addresses protect against the first type of attack by keeping ECDSA public keys hidden (hashed) until the first time satoshis sent to those addresses are spent, so attacks are effectively useless unless they can reconstruct private keys in less than the hour or two it takes for a transaction to be well protected by the block chain.
2. Unique (non-reused) private keys protect against the second type of attack by only generating one signature per private key, so attackers never get a subsequent signature to use in comparison-based attacks. Existing comparison-based attacks are only practical today when insufficient entropy is used in signing or when the entropy used is exposed by some means, such as a [side-channel attack](#).

So, for both privacy and security, we encourage you to build your applications to avoid public key reuse and, when possible, to discourage users from reusing addresses. If your application needs to provide a fixed URI to which payments should be sent, please see the [bitcoin: URI section](#) below.

Transaction Malleability

None of Bitcoin's signature hash types protect the signature script, leaving the door open for a limited denial of service attack called **transaction malleability**. The signature script contains the secp256k1 signature, which can't sign itself, allowing attackers to make non-functional modifications to a transaction without rendering it invalid. For example, an attacker can add some data to the signature script which will be dropped before the previous pubkey script is processed.

Although the modifications are non-functional—so they do not change what inputs the transaction uses nor what outputs it pays—they do change the computed hash of the transaction. Since each transaction links to previous transactions using hashes as a transaction identifier (txid), a modified transaction will not have the txid its creator expected.

This isn't a problem for most Bitcoin transactions which are designed to be added to the block chain immediately. But it does become a problem when the output from a transaction is spent before that transaction is added to the block chain.

Bitcoin developers have been working to reduce transaction malleability among standard transaction types, but a complete fix is still only in the planning stages. At present, new transactions should not depend on previous transactions which have not been added to the block chain yet, especially if large amounts of satoshis are at stake.

Transaction malleability also affects payment tracking. Bitcoin Core's RPC interface lets you track transactions by their txid—but if that txid changes because the transaction was modified, it may appear that the transaction has disappeared from the network.

Current best practices for transaction tracking dictate that a transaction should be tracked by the transaction outputs (UTXOs) it spends as inputs, as they cannot be changed without invalidating the transaction.

Best practices further dictate that if a transaction does seem to disappear from the network and needs to be reissued, that it be reissued in a way that invalidates the lost transaction. One method which will always work is to ensure the reissued payment spends all of the same outputs that the lost transaction used as inputs.

Contracts

Contracts are transactions which use the decentralized Bitcoin system to enforce financial agreements. Bitcoin contracts can often be crafted to minimize dependency on outside agents, such as the court system, which significantly decreases the risk of dealing with unknown entities in financial transactions.

The following subsections will describe a variety of Bitcoin contracts already in use. Because contracts deal with real people, not just transactions, they are framed below in story format.

Besides the contract types described below, many other contract types have been proposed. Several of them are collected on the [Contracts page](#) of the Bitcoin Wiki.

Escrow And Arbitration

Charlie-the-customer wants to buy a product from Bob-the-businessman, but neither of them trusts the other person, so they use a contract to help ensure Charlie gets his merchandise and Bob gets his payment.

A simple contract could say that Charlie will spend satoshis to an output which can only be spent if Charlie and Bob both sign the input spending it. That means Bob won't get paid unless Charlie gets his merchandise, but Charlie can't get the merchandise and keep his payment.

This simple contract isn't much help if there's a dispute, so Bob and Charlie enlist the help of Alice-the-arbitrator to create an **escrow contract**. Charlie spends his satoshis to an output which can only be spent if two of the three people sign the input. Now Charlie can pay Bob if everything is ok, Bob can refund Charlie's money if there's a problem, or Alice can arbitrate and decide who should get the satoshis if there's a dispute.

To create a multiple-signature (**multisig**) output, they each give the others a public key. Then Bob creates the following **P2SH multisig** redeem script:

```
OP_2 [A's pubkey] [B's pubkey] [C's pubkey] OP_3 OP_CHECKMULTISIG
```

(Op codes to push the public keys onto the stack are not shown.)

`OP_2` and `OP_3` push the actual numbers 2 and 3 onto the stack. `OP_2` specifies that 2 signatures are required to sign; `OP_3` specifies that 3 public keys (unhashed) are being provided. This is a 2-of-3 multisig pubkey script, more generically called a *m-of-n* pubkey script (where *m* is the *minimum* matching signatures required and *n* in the *number* of public keys provided).

Bob gives the redeem script to Charlie, who checks to make sure his public key and Alice's public key are included. Then he hashes the redeem script to create a P2SH redeem script and pays the satoshis to it. Bob sees the payment get added to the block chain and ships the merchandise.

Unfortunately, the merchandise gets slightly damaged in transit. Charlie wants a full refund, but Bob thinks a 10% refund is sufficient. They turn to Alice to resolve the issue. Alice asks for photo evidence from Charlie along with a copy of the redeem script Bob created and Charlie checked.

After looking at the evidence, Alice thinks a 40% refund is sufficient, so she creates and signs a transaction with two outputs, one that spends 60% of the satoshis to Bob's public key and one that spends the remaining 40% to Charlie's public key.

In the signature script Alice puts her signature and a copy of the unhashed serialized redeem script that Bob created. She gives a copy of the incomplete transaction to both Bob and Charlie. Either one of them can complete it by adding his signature to create the following signature script:

```
OP_0 [A's signature] [B's or C's signature] [serialized redeem script]
```

(Op codes to push the signatures and redeem script onto the stack are not shown. `OP_0` is a workaround for an off-by-one error in the original implementation which must be preserved for compatibility. Note that the signature script must provide signatures in the same order as the corresponding public keys appear in the redeem script. See the description in [OP_CHECKMULTISIG](#) for details.)

When the transaction is broadcast to the network, each peer checks the signature script against the P2SH output Charlie previously paid, ensuring that the redeem script matches the redeem script hash previously provided. Then the redeem script is evaluated, with the two signatures being used as input data. Assuming the redeem script validates, the two transaction outputs show up in Bob's and Charlie's wallets as spendable balances.

However, if Alice created and signed a transaction neither of them would agree to, such as spending all the satoshis to herself, Bob and Charlie can find a new arbitrator and sign a transaction spending the satoshis to another 2-of-3 multisig redeem script hash, this one including a public key from that second arbitrator. This means that Bob and Charlie never need to worry about their arbitrator stealing their money.

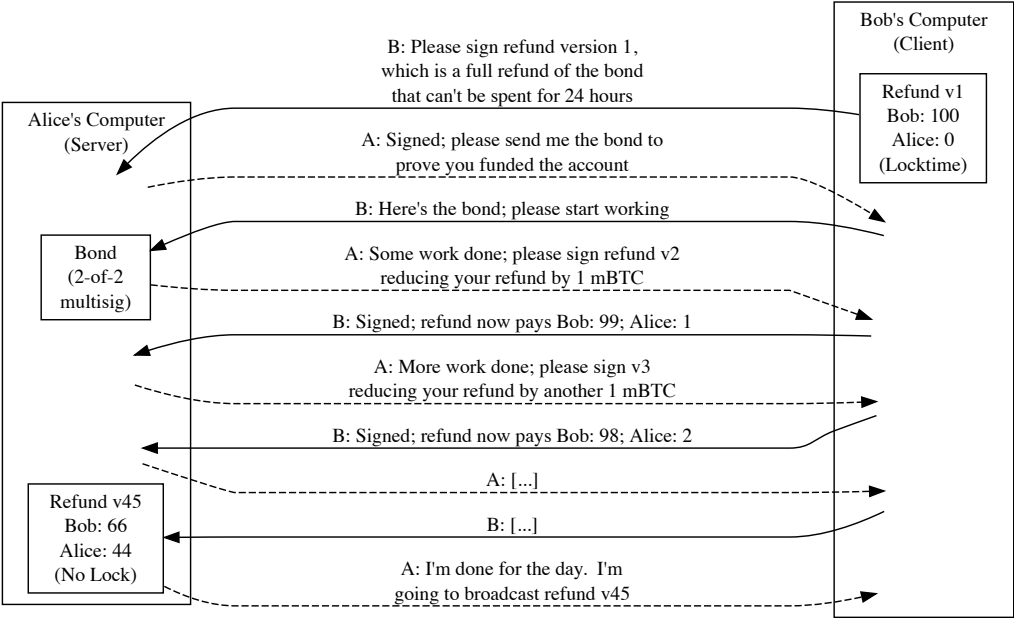
Resource: [BitRated](#) provides a multisig arbitration service interface using HTML/JavaScript on a GNU AGPL-licensed website.

Micropayment Channel

Alice also works part-time moderating forum posts for Bob. Every time someone posts to Bob's busy forum, Alice skims the post to make sure it isn't offensive or spam. Alas, Bob often forgets to pay her, so Alice demands to be paid immediately after each post she approves or rejects. Bob says he can't do that because hundreds of small payments will cost him thousands of satoshis in transaction fees, so Alice suggests they use a **micropayment channel**.

Bob asks Alice for her public key and then creates two transactions. The first transaction pays 100 millibitcoins to a P2SH output whose 2-of-2 multisig redeem script requires signatures from both Alice and Bob. This is the bond transaction. Broadcasting this transaction would let Alice hold the millibitcoins hostage, so Bob keeps this transaction private for now and creates a second transaction.

The second transaction spends all of the first transaction's millibitcoins (minus a transaction fee) back to Bob after a 24 hour delay enforced by locktime. This is the refund transaction. Bob can't sign the refund transaction by himself, so he gives it to Alice to sign, as shown in the illustration below.



Alice broadcasts the bond to the Bitcoin network immediately. She broadcasts the final version of the refund when she finishes work or before the locktime. If she fails to broadcast before refund v1's time lock expires, Bob can broadcast refund v1 to get a full refund.

Alice checks that the refund transaction's locktime is 24 hours in the future, signs it, and gives a copy of it back to Bob. She then asks Bob for the bond transaction and checks that the refund transaction spends the output of the bond transaction. She can now broadcast the bond transaction to the network to ensure Bob has to wait for the time lock to expire before further spending his millibitcoins. Bob hasn't actually spent anything so far, except possibly a small transaction fee, and he'll be able to broadcast the refund transaction in 24 hours for a full refund.

Now, when Alice does some work worth 1 millibitcoin, she asks Bob to create and sign a new version of the refund transaction. Version two of the transaction spends 1 millibitcoin to Alice and the other 99 back to Bob; it does not have a locktime, so Alice can sign it and spend it whenever she wants. (But she doesn't do that immediately.)

Alice and Bob repeat these work-and-pay steps until Alice finishes for the day, or until the time lock is about to expire. Alice signs the final version of the refund transaction and broadcasts it, paying herself and refunding any remaining balance to Bob. The next day, when Alice starts work, they create a new micropayment channel.

If Alice fails to broadcast a version of the refund transaction before its time lock expires, Bob can broadcast the first version and receive a full refund. This is one reason micropayment channels are best suited to small payments—if Alice's Internet service goes out for a few hours near the time lock expiry, she could be cheated out of her payment.

Transaction malleability, discussed above in the Transactions section, is another reason to limit the value of micropayment channels. If someone uses transaction malleability to break the link between the two transactions, Alice could hold Bob's 100 millibitcoins hostage even if she hadn't done any work.

For larger payments, Bitcoin transaction fees are very low as a percentage of the total transaction value, so it makes more sense to protect payments with immediately-broadcast separate transactions.

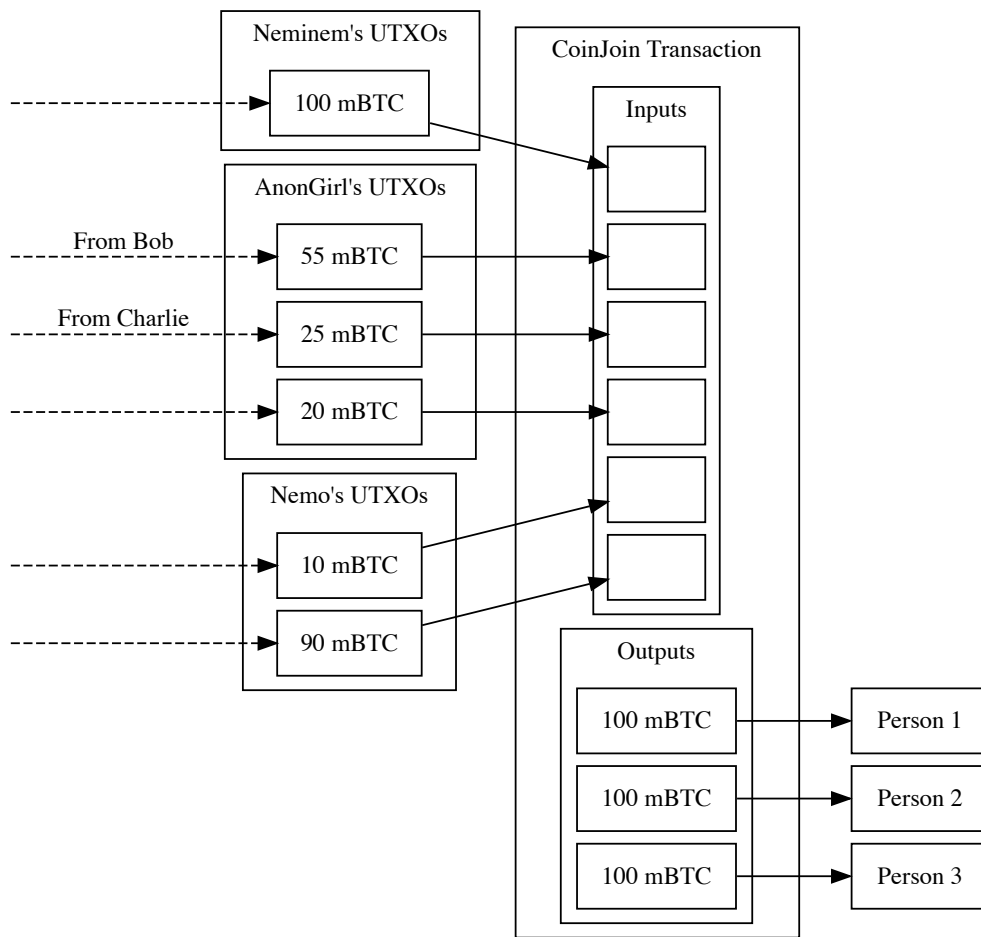
Resource: The [bitcoinj](#) Java library provides a complete set of micropayment functions, an example implementation, and [a tutorial](#) all under an Apache license.

CoinJoin

Alice is concerned about her privacy. She knows every transaction gets added to the public block chain, so when Bob and Charlie pay her, they can each easily track those satoshis to learn what Bitcoin addresses she pays, how much she pays them, and possibly how many satoshis she has left.

Alice isn't a criminal, she just wants plausible deniability about where she has spent her satoshis and how many she has left, so she starts up the Tor anonymity service on her computer and logs into an IRC chatroom as "AnonGirl."

Also in the chatroom are "Nemo" and "Neminem." They collectively agree to transfer satoshis between each other so no one besides them can reliably determine who controls which satoshis. But they're faced with a dilemma: who transfers their satoshis to one of the other two pseudonymous persons first? The CoinJoin-style contract, shown in the illustration below, makes this decision easy: they create a single transaction which does all of the spending simultaneously, ensuring none of them can steal the others' satoshis.



Example CoinJoin Transaction
Only the participants know who gets which output.

Each contributor looks through their collection of Unspent Transaction Outputs (UTXOs) for 100 millibitcoins they can spend. They then each generate a brand new public key and give UTXO details and pubkey hashes to the facilitator. In this case, the facilitator is AnonGirl; she creates a transaction spending each of the UTXOs to three equally-sized outputs. One output goes to each of the contributors' pubkey hashes.

AnonGirl then signs her inputs using `SIGHASH_ALL` to ensure nobody can change the input or output details. She gives the partially-signed transaction to Nemo who signs his inputs the same way and passes it to Neminem, who also signs it the same way. Neminem then broadcasts the transaction to the peer-to-peer network, mixing all of the millibitcoins in a single transaction.

As you can see in the illustration, there's no way for anyone besides AnonGirl, Nemo, and Neminem to confidently determine who received which output, so they can each spend their output with plausible deniability.

Now when Bob or Charlie try to track Alice's transactions through the block chain, they'll also see transactions made by Nemo and Neminem. If Alice does a few more CoinJoins, Bob and Charlie might have to guess which transactions made by dozens or hundreds of people were actually made by Alice.

The complete history of Alice's satoshis is still in the block chain, so a determined investigator could talk to the people AnonGirl CoinJoined with to find out the ultimate origin of her satoshis and possibly reveal AnonGirl as Alice. But against anyone casually browsing block chain history, Alice gains plausible deniability.

The CoinJoin technique described above costs the participants a small amount of satoshis to pay the transaction fee. An alternative technique, purchaser CoinJoin, can actually save them satoshis and improve their privacy at the same time.

AnonGirl waits in the IRC chatroom until she wants to make a purchase. She announces her intention to spend satoshis and waits

until someone else wants to make a purchase, likely from a different merchant. Then they combine their inputs the same way as before but set the outputs to the separate merchant addresses so nobody will be able to figure out solely from block chain history which one of them bought what from the merchants.

Since they would've had to pay a transaction fee to make their purchases anyway, AnonGirl and her co-spenders don't pay anything extra—but because they reduced overhead by combining multiple transactions, saving bytes, they may be able to pay a smaller aggregate transaction fee, saving each one of them a tiny amount of satoshis.

Resource: An alpha-quality (as of this writing) implementation of decentralized CoinJoin is [CoinMux](#), available under the Apache license.

Wallets

A Bitcoin wallet can refer to either a wallet program or a wallet file. Wallet programs create public keys to receive satoshis and use the corresponding private keys to spend those satoshis. Wallet files store private keys and (optionally) other information related to transactions for the wallet program.

Wallet programs and wallet files are addressed below in separate subsections, and this document attempts to always make it clear whether we're talking about wallet programs or wallet files.

Wallet Programs

Permitting receiving and spending of satoshis is the only essential feature of wallet software—but a particular wallet program doesn't need to do both things. Two wallet programs can work together, one program distributing public keys in order to receive satoshis and another program signing transactions spending those satoshis.

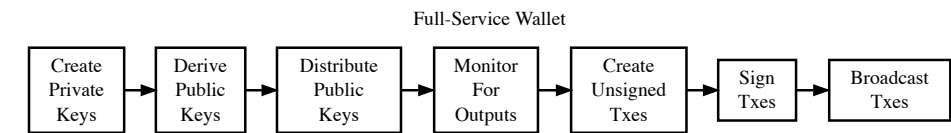
Wallet programs also need to interact with the peer-to-peer network to get information from the block chain and to broadcast new transactions. However, the programs which distribute public keys or sign transactions don't need to interact with the peer-to-peer network themselves.

This leaves us with three necessary, but separable, parts of a wallet system: a public key distribution program, a signing program, and a networked program. In the subsections below, we will describe common combinations of these parts.

Note: we speak about distributing public keys generically. In many cases, P2PKH or P2SH hashes will be distributed instead of public keys, with the actual public keys only being distributed when the outputs they control are spent.

Full-Service Wallets

The simplest wallet is a program which performs all three functions: it generates private keys, derives the corresponding public keys, helps distribute those public keys as necessary, monitors for outputs spent to those public keys, creates and signs transactions spending those outputs, and broadcasts the signed transactions.



As of this writing, almost all popular wallets can be used as full-service wallets.

The main advantage of full-service wallets is that they are easy to use. A single program does everything the user needs to receive and spend satoshis.

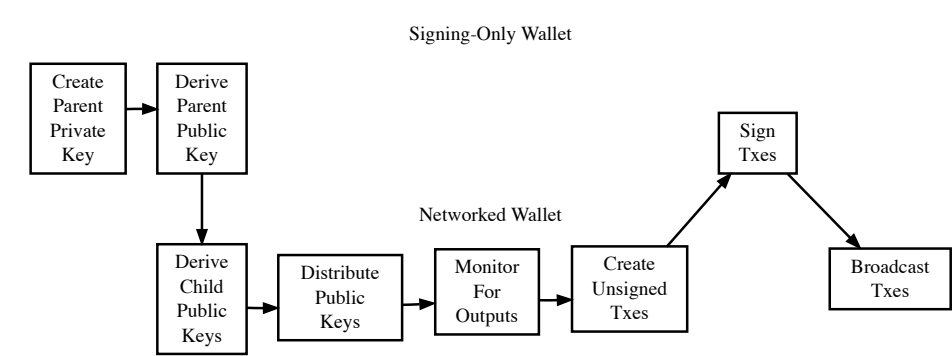
The main disadvantage of full-service wallets is that they store the private keys on a device connected to the Internet. The compromise of such devices is a common occurrence, and an Internet connection makes it easy to transmit private keys from a compromised device to an attacker.

To help protect against theft, many wallet programs offer users the option of encrypting the wallet files which contain the private keys. This protects the private keys when they aren't being used, but it cannot protect against an attack designed to capture the encryption key or to read the decrypted keys from memory.

Signing-Only Wallets

To increase security, private keys can be generated and stored by a separate wallet program operating in a more secure environment. These signing-only wallets work in conjunction with a networked wallet which interacts with the peer-to-peer network.

Signing-only wallets programs typically use deterministic key creation (described in a later subsection) to create parent private and public keys which can create child private and public keys.



When first run, the signing-only wallet creates a parent private key and transfers the corresponding parent public key to the networked wallet.

The networked wallet uses the parent public key to derive child public keys, optionally helps distribute them, monitors for outputs spent to those public keys, creates unsigned transactions spending those outputs, and transfers the unsigned transactions to the signing-only wallet.

Often, users are given a chance to review the unsigned transactions' details (particularly the output details) using the signing-only wallet.

After the optional review step, the signing-only wallet uses the parent private key to derive the appropriate child private keys and signs the transactions, giving the signed transactions back to the networked wallet.

The networked wallet then broadcasts the signed transactions to the peer-to-peer network.

The following subsections describe the two most common variants of signing-only wallets: offline wallets and hardware wallets.

Offline Wallets

Several full-service wallets programs will also operate as two separate wallets: one program instance acting as a signing-only wallet (often called an “offline wallet”) and the other program instance acting as the networked wallet (often called an “online wallet” or “watching-only wallet”).

The offline wallet is so named because it is intended to be run on a device which does not connect to any network, greatly reducing the number of attack vectors. If this is the case, it is usually up to the user to handle all data transfer using removable media such as USB drives. The user’s workflow is something like:

1. (Offline) Disable all network connections on a device and install the wallet software. Start the wallet software in offline mode to create the parent private and public keys. Copy the parent public key to removable media.
2. (Online) Install the wallet software on another device, this one connected to the Internet, and import the parent public key from the removable media. As you would with a full-service wallet, distribute public keys to receive payment. When ready to spend satoshis, fill in the output details and save the unsigned transaction generated by the wallet to removable media.
3. (Offline) Open the unsigned transaction in the offline instance, review the output details to make sure they spend the correct amount to the correct address. This prevents malware on the online wallet from tricking the user into signing a transaction which pays an attacker. After review, sign the transaction and save it to removable media.
4. (Online) Open the signed transaction in the online instance so it can broadcast it to the peer-to-peer network.

The primary advantage of offline wallets is their possibility for greatly improved security over full-service wallets. As long as the offline wallet is not compromised (or flawed) and the user reviews all outgoing transactions before signing, the user’s satoshis are safe even if the online wallet is compromised.

The primary disadvantage of offline wallets is hassle. For maximum security, they require the user dedicate a device to only offline tasks. The offline device must be booted up whenever funds are to be spent, and the user must physically copy data from the online device to the offline device and back.

Hardware Wallets

Hardware wallets are devices dedicated to running a signing-only wallet. Their dedication lets them eliminate many of the vulnerabilities present in operating systems designed for general use, allowing them to safely communicate directly with other devices so users don’t need to transfer data manually. The user’s workflow is something like:

1. (Hardware) Create parent private and public keys. Connect hardware wallet to a networked device so it can get the parent public key.
2. (Networked) As you would with a full-service wallet, distribute public keys to receive payment. When ready to spend satoshis, fill in the transaction details, connect the hardware wallet, and click Spend. The networked wallet will automatically send the transaction details to the hardware wallet.
3. (Hardware) Review the transaction details on the hardware wallet’s screen. Some hardware wallets may prompt for a passphrase or PIN number. The hardware wallet signs the transaction and uploads it to the networked wallet.
4. (Networked) The networked wallet receives the signed transaction from the hardware wallet and broadcasts it to the network.

The primary advantage of hardware wallets is their possibility for greatly improved security over full-service wallets with much less hassle than offline wallets.

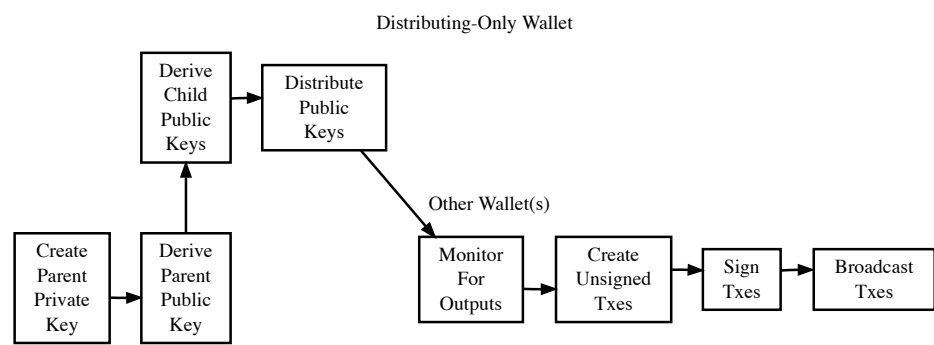
The primary disadvantage of hardware wallets is their hassle. Even though the hassle is less than that of offline wallets, the user

must still purchase a hardware wallet device and carry it with them whenever they need to make a transaction using the signing-only wallet.

An additional (hopefully temporary) disadvantage is that, as of this writing, very few popular wallet programs support hardware wallets—although almost all popular wallet programs have announced their intention to support at least one model of hardware wallet.

Distributing-Only Wallets

Wallet programs which run in difficult-to-secure environments, such as webserver, can be designed to distribute public keys (including P2PKH or P2SH addresses) and nothing more. There are two common ways to design these minimalist wallets:



- Pre-populate a database with a number of public keys or addresses, and then distribute on request a pubkey script or address using one of the database entries. To [avoid key reuse](#), webserver should keep track of used keys and never run out of public keys. This can be made easier by using parent public keys as suggested in the next method.
- Use a parent public key to create child public keys. To avoid key reuse, a method must be used to ensure the same public key isn't distributed twice. This can be a database entry for each key distributed or an incrementing pointer to the key index number.

Neither method adds a significant amount of overhead, especially if a database is used anyway to associate each incoming payment with a separate public key for payment tracking. See the [Payment Processing](#) section for details.

Wallet Files

Bitcoin wallets at their core are a collection of private keys. These collections are stored digitally in a file, or can even be physically stored on pieces of paper.

Private Key Formats

Private keys are what are used to unlock satoshis from a particular address. In Bitcoin, a private key in standard format is simply a 256-bit number, between the values:

0x01 and 0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140, representing nearly the entire range of $2^{256}-1$ values. The range is governed by the secp256k1 ECDSA encryption standard used by Bitcoin.

Wallet Import Format (WIF)

In order to make copying of private keys less prone to error, **Wallet Import Format** may be utilized. WIF uses base58Check encoding on an private key, greatly decreasing the chance of copying error, much like standard Bitcoin addresses.

1. Take a private key.
2. Add a 0x80 byte in front of it for mainnet addresses or 0xef for testnet addresses.
3. Append a 0x01 byte after it if it should be used with compressed public keys (described in a later subsection). Nothing is appended if it is used with uncompressed public keys.
4. Perform a SHA-256 hash on the extended key.
5. Perform a SHA-256 hash on result of SHA-256 hash.
6. Take the first four bytes of the second SHA-256 hash; this is the checksum.
7. Add the four checksum bytes from point 5 at the end of the extended key from point 2.
8. Convert the result from a byte string into a Base58 string using Base58Check encoding.

The process is easily reversible, using the Base58 decoding function, and removing the padding.

Mini Private Key Format

Mini private key format is a method for encoding a private key in under 30 characters, enabling keys to be embedded in a small physical space, such as physical bitcoin tokens, and more damage-resistant QR codes.

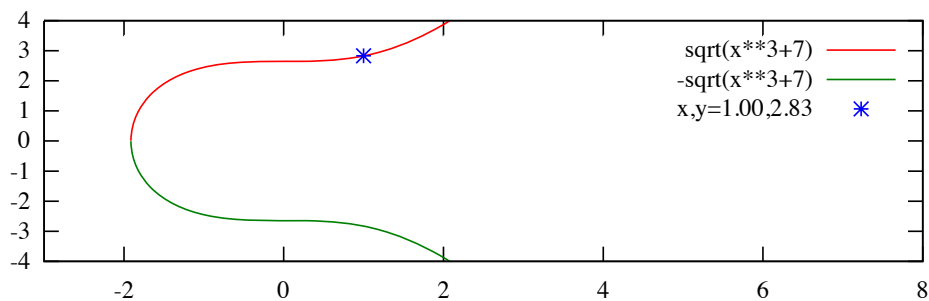
1. The first character of mini keys is 'S'.
2. In order to determine if a mini private key is well-formatted, a question mark is added to the private key.
3. The SHA256 hash is calculated. If the first byte produced is a '00', it is well-formatted. This key restriction acts as a typo-checking mechanism. A user brute forces the process using random numbers until a well-formatted mini private key is produced.
4. In order to derive the full private key, the user simply takes a single SHA256 hash of the original mini private key. This process is one-way: it is intractable to compute the mini private key format from the derived key.

Many implementations disallow the character '1' in the mini private key due to its visual similarity to 'l'.

Resource: A common tool to create and redeem these keys is the [Casascius Bitcoin Address Utility](#).

Public Key Formats

Bitcoin ECDSA public keys represent a point on a particular Elliptic Curve (EC) defined in secp256k1. In their traditional uncompressed form, public keys contain an identification byte, a 32-byte X coordinate, and a 32-byte Y coordinate. The extremely simplified illustration below shows such a point on the elliptic curve used by Bitcoin, $x^2 = y^3 + 7$, over a field of contiguous numbers.



(Secp256k1 actually modulus coordinates by a large prime, which produces a field of non-contiguous integers and a significantly less clear plot, although the principles are the same.)

An almost 50% reduction in public key size can be realized without changing any fundamentals by dropping the Y coordinate. This is possible because only two points along the curve share any particular X coordinate, so the 32-byte Y coordinate can be replaced with a single bit indicating whether the point is on what appears in the illustration as the “top” side or the “bottom” side.

No data is lost by creating these compressed public keys—only a small amount of CPU is necessary to reconstruct the Y coordinate and access the uncompressed public key. Both uncompressed and compressed public keys are described in official secp256k1 documentation and supported by default in the widely-used OpenSSL library.

Because they’re easy to use, and because they reduce almost by half the block chain space used to store public keys for every spent output, compressed public keys are the default in Bitcoin Core and are the recommended default for all Bitcoin software.

However, Bitcoin Core prior to 0.6 used uncompressed keys. This creates a few complications, as the hashed form of an uncompressed key is different than the hashed form of a compressed key, so the same key works with two different P2PKH addresses. This also means that the key must be submitted in the correct format in the signature script so it matches the hash in the previous output’s pubkey script.

For this reason, Bitcoin Core uses several different identifier bytes to help programs identify how keys should be used:

- Private keys meant to be used with compressed public keys have 0x01 appended to them before being Base-58 encoded. (See the private key encoding section above.)
- Uncompressed public keys start with 0x04; compressed public keys begin with 0x03 or 0x02 depending on whether they’re greater or less than the midpoint of the curve. These prefix bytes are all used in official secp256k1 documentation.

Hierarchical Deterministic Key Creation

The hierarchical deterministic key creation and transfer protocol (**HD protocol**) greatly simplifies wallet backups, eliminates the need for repeated communication between multiple programs using the same wallet, permits creation of child accounts which can operate independently, gives each parent account the ability to monitor or control its children even if the child account is compromised, and divides each account into full-access and restricted-access parts so untrusted users or programs can be allowed to receive or monitor payments without being able to spend them.

The HD protocol takes advantage of the ECDSA public key creation function, `point()`, which takes a large integer (the private key) and turns it into a graph point (the public key):

```
point(private_key) == public_key
```

Because of the way `point()` functions, it’s possible to create a **child public key** by combining an existing (**parent**) public key with

another public key created from any integer (i) value. This child public key is the same public key which would be created by the `point()` function if you added the i value to the original (parent) private key and then found the remainder of that sum divided by a global constant used by all Bitcoin software (G):

```
point( (parent_private_key + i) % G ) == parent_public_key + point(i)
```

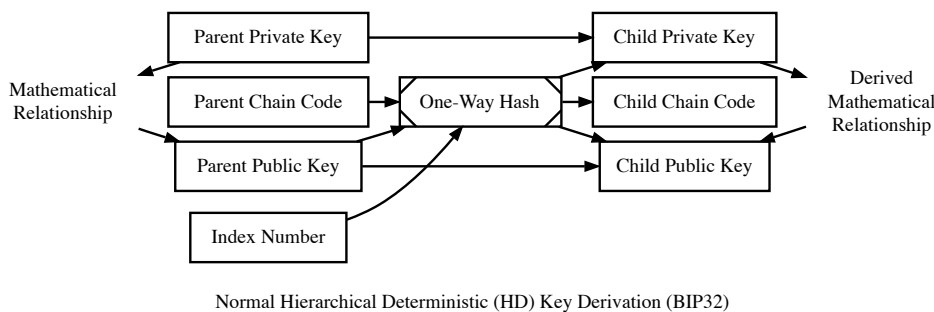
This means that two or more independent programs which agree on a sequence of integers can create a series of unique **child key** pairs from a single **parent key** pair without any further communication. Moreover, the program which distributes new public keys for receiving payment can do so without any access to the private keys, allowing the public key distribution program to run on a possibly-insecure platform such as a public web server.

Child public keys can also create their own child public keys (grandchild public keys) by repeating the child key derivation operations:

```
point( (child_private_key + i) % G ) == child_public_key + point(i)
```

Whether creating child public keys or further-descended public keys, a predictable sequence of integer values would be no better than using a single public key for all transactions, as anyone who knew one child public key could find all of the other child public keys created from the same parent public key. Instead, a random seed can be used to deterministically generate the sequence of integer values so that the relationship between the child public keys is invisible to anyone without that seed.

The HD protocol uses a single root seed to create a hierarchy of child, grandchild, and other descended keys with unlinkable deterministically-generated integer values. Each child key also gets a deterministically-generated seed from its parent, called a **chain code**, so the compromising of one chain code doesn't necessary compromise the integer sequence for the whole hierarchy, allowing the **master chain code** to continue being useful even if, for example, a web-based public key distribution program gets hacked.



As illustrated above, HD key derivation takes four inputs:

- The *parent private key* and *parent public key* are regular uncompressed 256-bit ECDSA keys.
- The *parent chain code* is 256 bits of seemingly-random data.
- The *index* number is a 32-bit integer specified by the program.

In the normal form shown in the above illustration, the parent chain code, the parent public key, and the index number are fed into a one-way cryptographic hash ([HMAC-SHA512](#)) to produce 512 bits of deterministically-generated-but-seemingly-random data. The seemingly-random 256 bits on the righthand side of the hash output are used as a new child chain code. The seemingly-random 256 bits on the lefthand side of the hash output are used as the integer value to be combined with either the parent private key or parent public key to, respectively, create either a child private key or child public key:

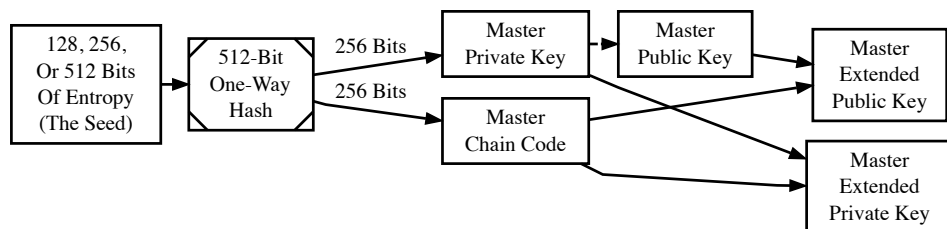
```

point( (parent_private_key + lefthand_hash_output) % G ) == child_public_key
point(child_private_key) == parent_public_key + point(lefthand_hash_output)

```

Specifying different index numbers will create different unlinkable child keys from the same parent keys. Repeating the procedure for the child keys using the child chain code will create unlinkable grandchild keys.

Because creating child keys requires both a key and a chain code, the key and chain code together are called the **extended key**. An **extended private key** and its corresponding **extended public key** have the same chain code. The (top-level parent) **master private key** and master chain code are derived from random data, as illustrated below.



Creation Of The Master Keys

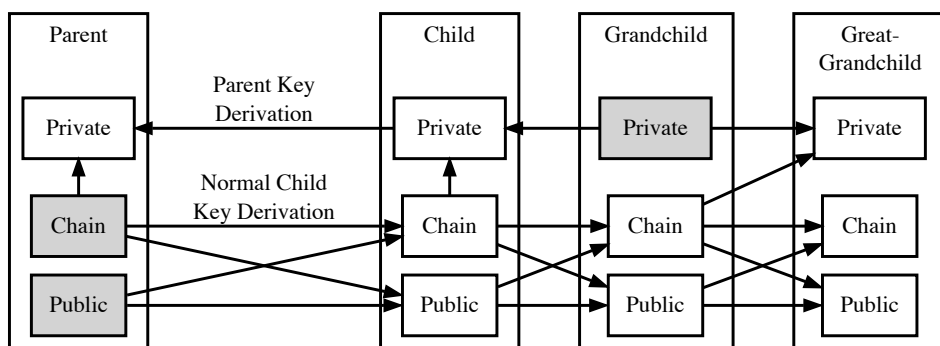
A **root seed** is created from either 128 bits, 256 bits, or 512 bits of random data. This root seed of as little as 128 bits is the the only data the user needs to backup in order to derive every key created by a particular wallet program using particular settings.

⚠ Warning: As of this writing, HD wallet programs are not expected to be fully compatible, so users must only use the same HD wallet program with the same HD-related settings for a particular root seed.

The root seed is hashed to create 512 bits of seemingly-random data, from which the master private key and master chain code are created (together, the master extended private key). The master public key is derived from the master private key using `point()`, which, together with the master chain code, is the master extended public key. The master extended keys are functionally equivalent to other extended keys; it is only their location at the top of the hierarchy which makes them special.

Hardened Keys

Hardened extended keys fix a potential problem with normal extended keys. If an attacker gets a normal parent chain code and parent public key, he can brute-force all chain codes deriving from it. If the attacker also obtains a child, grandchild, or further-descended private key, he can use the chain code to generate all of the extended private keys descending from that private key, as shown in the grandchild and great-grandchild generations of the illustration below.



Cross-Generational Key Compromise

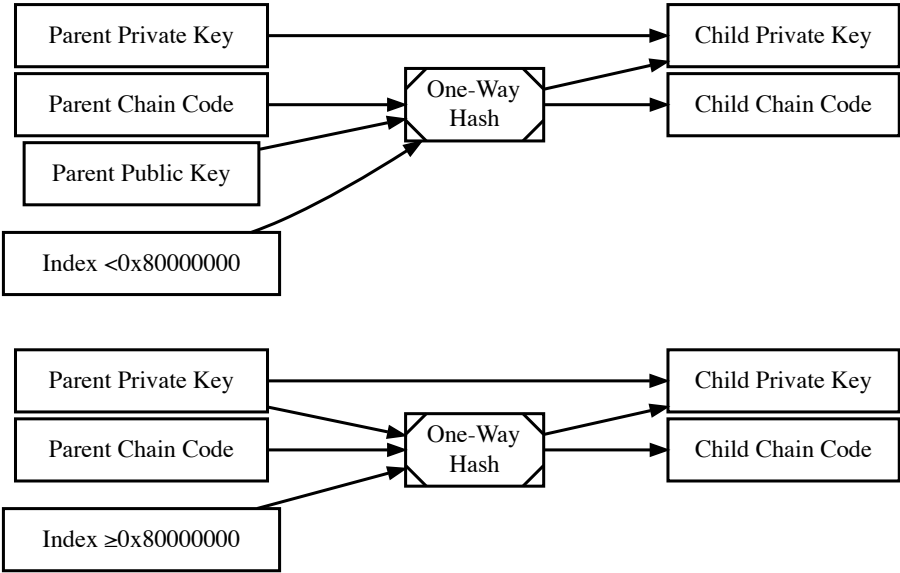
Perhaps worse, the attacker can reverse the normal child private key derivation formula and subtract a parent chain code from a

child private key to recover the parent private key, as shown in the child and parent generations of the illustration above. This means an attacker who acquires an extended public key and any private key descended from it can recover that public key's private key and all keys descended from it.

For this reason, the chain code part of an extended public key should be better secured than standard public keys and users should be advised against exporting even non-extended private keys to possibly-untrustworthy environments.

This can be fixed, with some tradeoffs, by replacing the the normal key derivation formula with a hardened key derivation formula.

The normal key derivation formula, described in the section above, combines together the index number, the parent chain code, and the parent public key to create the child chain code and the integer value which is combined with the parent private key to create the child private key.



Normal (Top) And Hardened (Bottom) Child Private Key Derivation

The hardened formula, illustrated above, combines together the index number, the parent chain code, and the parent private key to create the data used to generate the child chain code and child private key. This formula makes it impossible to create child public keys without knowing the parent private key. In other words, parent extended public keys can't create hardened child public keys.

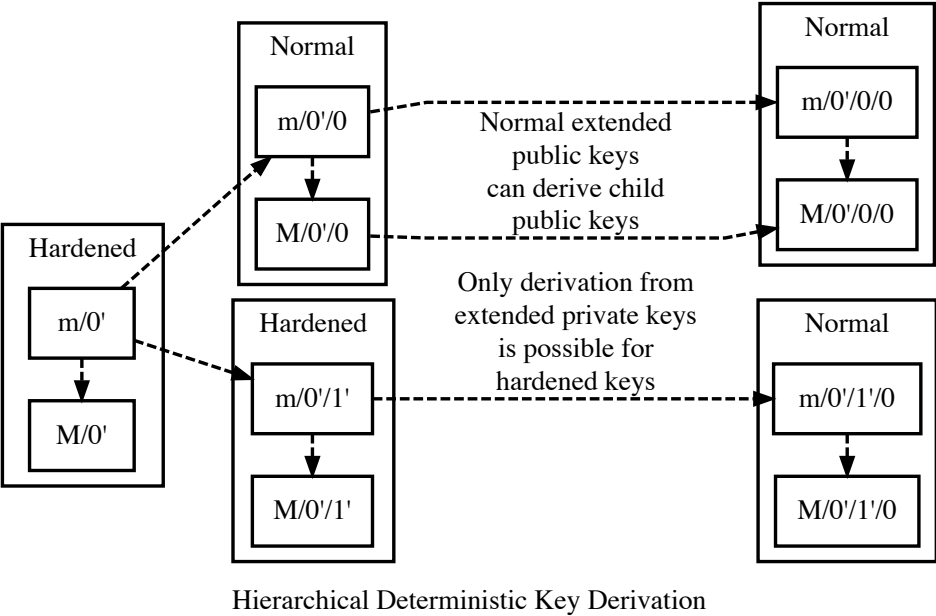
Because of that, a **hardened extended private key** is much less useful than a normal extended private key—however, hardened extended private keys create a firewall through which multi-level key derivation compromises cannot happen. Because hardened child extended public keys cannot generate grandchild chain codes on their own, the compromise of a parent extended public key cannot be combined with the compromise of a grandchild private key to create great-grandchild extended private keys.

The HD protocol uses different index numbers to indicate whether a normal or hardened key should be generated. Index numbers from 0x00 to 0x7ffffff (0 to $2^{31}-1$) will generate a normal key; index numbers from 0x80000000 to 0xffffffff will generate a hardened key. To make descriptions easy, many developers use the [prime symbol](#) to indicate hardened keys, so the first normal key (0x00) is 0 and the first hardened key (0x80000000) is 0'.

(Bitcoin developers typically use the ASCII apostrophe rather than the unicode prime symbol, a convention we will henceforth follow.)

This compact description is further combined with slashes prefixed by *m* or *M* to indicate hierarchy and key type, with *m* being a private key and *M* being a public key. For example, m/0'/0/122' refers to the 123rd hardened private child (by index number) of the

first normal child (by index) of the first hardened child (by index) of the master private key. The following hierarchy illustrates prime notation and hardened key firewalls.



Wallets following the BIP32 HD protocol only create hardened children of the master private key (m) to prevent a compromised child key from compromising the master key. As there are no normal children for the master keys, the master public key is not used in HD wallets. All other keys can have normal children, so the corresponding extended public keys may be used instead.

The HD protocol also describes a serialization format for extended public keys and extended private keys. For details, please see the [wallet section in the developer reference](#) or BIP32 for the full HD protocol specification.

Storing Root Seeds

Root seeds in the HD protocol are 128, 256, or 512 bits of random data which must be backed up precisely. To make it more convenient to use non-digital backup methods, such as memorization or hand-copying, BIP39 defines a method for creating a 512-bit root seed from a pseudo-sentence (mnemonic) of common natural-language words which was itself created from 128 to 256 bits of entropy and optionally protected by a password.

The number of words generated correlates to the amount of entropy used:

Entropy Bits	Words
128	12
160	15
192	18
224	21
256	24

The passphrase can be of any length. It is simply appended to the mnemonic pseudo-sentence, and then both the mnemonic and

password are hashed 2,048 times using HMAC-SHA512, resulting in a seemingly-random 512-bit seed. Because any input to the hash function creates a seemingly-random 512-bit seed, there is no fundamental way to prove the user entered the correct password, possibly allowing the user to protect a seed even when under duress.

For implementation details, please see BIP39.

Loose-Key Wallets

Loose-Key wallets, also called “Just a Bunch Of Keys (JBOK)”, are a deprecated form of wallet that originated from the Bitcoin Core client wallet. The Bitcoin Core client wallet would create 100 private key/public key pairs automatically via a Pseudo-Random-Number Generator (PRNG) for later use.

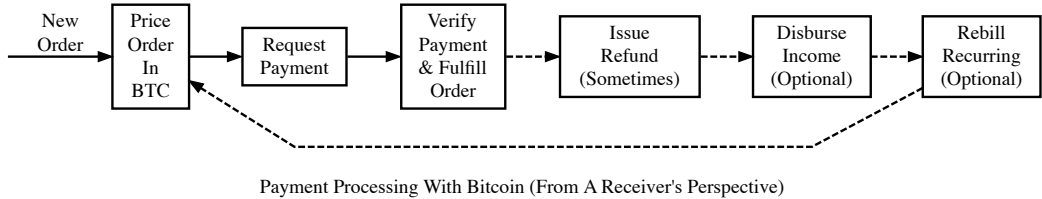
These unused private keys are stored in a virtual “key pool”, with new keys being generated whenever a previously-generated key was used, ensuring the pool maintained 100 unused keys. (If the wallet is encrypted, new keys are only generated while the wallet is unlocked.)

This created considerable difficulty in backing up one’s keys, considering backups have to be run manually to save the newly-generated private keys. If a new key pair set is generated, used, and then lost prior to a backup, the stored satoshis are likely lost forever. Many older-style mobile wallets followed a similar format, but only generated a new private key upon user demand.

This wallet type is being actively phased out and discouraged from being used due to the backup hassle.

Payment Processing

Payment processing encompasses the steps spenders and receivers perform to make and accept payments in exchange for products or services. The basic steps have not changed since the dawn of commerce, but the technology has. This section will explain how receivers and spenders can, respectively, request and make payments using Bitcoin—and how they can deal with complications such as refunds and recurrent rebilling.



The figure above illustrates payment processing using Bitcoin from a receiver’s perspective, starting with a new order. The following subsections will each address the three common steps and the three occasional or optional steps.

It is worth mentioning that each of these steps can be outsourced by using third party APIs and services.

Pricing Orders

Because of exchange rate variability between satoshis and national currencies (fiat), many Bitcoin orders are priced in fiat but paid in satoshis, necessitating a price conversion.

Exchange rate data is widely available through HTTP-based APIs provided by currency exchanges. Several organizations also aggregate data from multiple exchanges to create index prices, which are also available using HTTP-based APIs.

Any applications which automatically calculate order totals using exchange rate data must take steps to ensure the price quoted reflects the current general market value of satoshis, or the applications could accept too few satoshis for the product or service being sold. Alternatively, they could ask for too many satoshis, driving away potential spenders.

To minimize problems, your applications may want to collect data from at least two separate sources and compare them to see how much they differ. If the difference is substantial, your applications can enter a safe mode until a human is able to evaluate the situation.

You may also want to program your applications to enter a safe mode if exchange rates are rapidly increasing or decreasing, indicating a possible problem in the Bitcoin market which could make it difficult to spend any satoshis received today.

Exchange rates lie outside the control of Bitcoin and related technologies, so there are no new or planned technologies which will make it significantly easier for your program to correctly convert order totals from fiat into satoshis.

Because the exchange rate fluctuates over time, order totals pegged to fiat must expire to prevent spenders from delaying payment in the hope that satoshis will drop in price. Most widely-used payment processing systems currently expire their invoices after 10 to 20 minutes.

Shorter expiration periods increase the chance the invoice will expire before payment is received, possibly necessitating manual intervention to request an additional payment or to issue a refund. Longer expiration periods increase the chance that the exchange rate will fluctuate a significant amount before payment is received.


Requesting Payments

Before requesting payment, your application must create a Bitcoin address, or acquire an address from another program such as Bitcoin Core. Bitcoin addresses are described in detail in the [Transactions](#) section. Also described in that section are two important reasons to [avoid using an address more than once](#)—but a third reason applies especially to payment requests:

Using a separate address for each incoming payment makes it trivial to determine which customers have paid their payment requests. Your applications need only track the association between a particular payment request and the address used in it, and then scan the block chain for transactions matching that address.

The next subsections will describe in detail the following four compatible ways to give the spender the address and amount to be paid. For increased convenience and compatibility, providing all of these options in your payment requests is recommended.

1. All wallet software lets its users paste in or manually enter an address and amount into a payment screen. This is, of course, inconvenient—but it makes an effective fallback option.
2. Almost all desktop wallets can associate with `bitcoin:` URIs, so spenders can click a link to pre-fill the payment screen. This also works with many mobile wallets, but it generally does not work with web-based wallets unless the spender installs a browser extension or manually configures a URI handler.
3. Most mobile wallets support scanning `bitcoin:` URIs encoded in a QR code, and almost all wallets can display them for accepting payment. While also handy for online orders, QR Codes are especially useful for in-person purchases.
4. Recent wallet updates add support for the new payment protocol providing increased security, authentication of a receiver's identity using X.509 certificates, and other important features such as refunds.

 **Warning:** Special care must be taken to avoid the theft of incoming payments. In particular, private keys should not be stored on web servers, and payment requests should be sent over HTTPS or other secure methods to prevent man-in-the-middle attacks from replacing your Bitcoin address with the attacker's address.

Plain Text

To specify an amount directly for copying and pasting, you must provide the address, the amount, and the denomination. An expiration time for the offer may also be specified. For example:

(Note: all examples in this section use testnet addresses.)

```
Pay: mJsk1Ny9spzU2fouzYgLqGUD8U41iR35QN
Amount: 100 BTC
You must pay by: 2014-04-01 at 23:00 UTC
```

Indicating the denomination is critical. As of this writing, popular Bitcoin wallet software defaults to denominating amounts in either bitcoins (BTC) , millibitcoins (mBTC) or microbitcoins (uBTC, “bits”). Choosing between each unit is widely supported, but other software also lets its users select denomination amounts from some or all of the following options:

Bitcoins	Unit (Abbreviation)
1.0	bitcoin (BTC)
0.01	bitcent (cBTC)
0.001	millibitcoin (mBTC)
0.000001	microbitcoin (uBTC, “bits”)
0.00000001	satoshi

bitcoin: URI

The `bitcoin:` URI scheme defined in BIP21 eliminates denomination confusion and saves the spender from copying and pasting two separate values. It also lets the payment request provide some additional information to the spender. An example:

```
bitcoin:mJsk1Ny9spzU2fouzYgLqGUD8U41iR35QN?amount=100
```

Only the address is required, and if it is the only thing specified, wallets will pre-fill a payment request with it and let the spender enter an amount. The amount specified is always in decimal bitcoins (BTC).

Two other parameters are widely supported. The `label` parameter is generally used to provide wallet software with the recipient’s name. The `message` parameter is generally used to describe the payment request to the spender. Both the label and the message are commonly stored by the spender’s wallet software—but they are never added to the actual transaction, so other Bitcoin users cannot see them. Both the label and the message must be [URI encoded](#).

All four parameters used together, with appropriate URI encoding, can be seen in the line-wrapped example below.

```
bitcoin:mJsk1Ny9spzU2fouzYgLqGUD8U41iR35QN\
?amount=0.10\
&label=Example+Merchant\
&message=Order+of+flowers+%26+chocolates
```

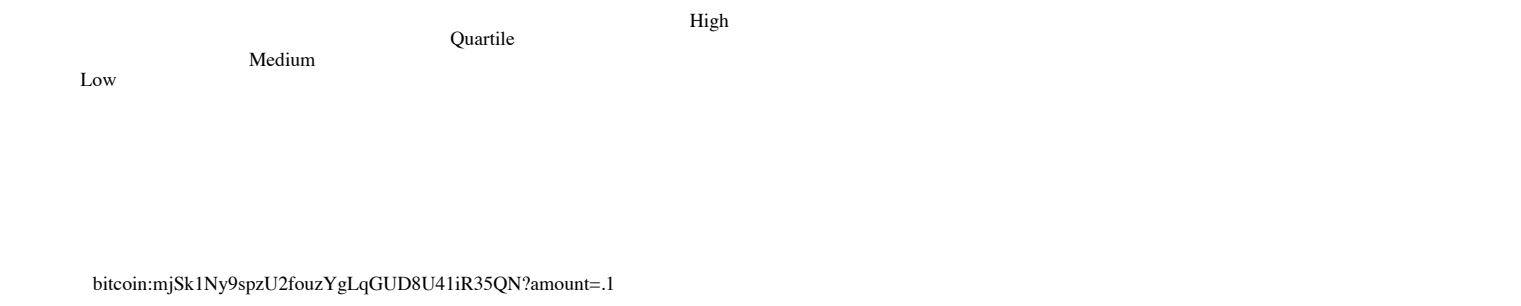
The URI scheme can be extended, as will be seen in the payment protocol section below, with both new optional and required parameters. As of this writing, the only widely-used parameter besides the four described above is the payment protocol’s `r` parameter.

Programs accepting URIs in any form must ask the user for permission before paying unless the user has explicitly disabled prompting (as might be the case for micropayments).

QR Codes

QR codes are a popular way to exchange `bitcoin:` URIs in person, in images, or in videos. Most mobile Bitcoin wallet apps, and some desktop wallets, support scanning QR codes to pre-fill their payment screens.

The figure below shows the same `bitcoin:` URI code encoded as four different **Bitcoin QR codes** at four different error correction levels. The QR code can include the `label` and `message` parameters—and any other optional parameters—but they were omitted here to keep the QR code small and easy to scan with unsteady or low-resolution mobile cameras.



The error correction is combined with a checksum to ensure the Bitcoin QR code cannot be successfully decoded with data missing or accidentally altered, so your applications should choose the appropriate level of error correction based on the space you have available to display the code. Low-level damage correction works well when space is limited, and quartile-level damage correction helps ensure fast scanning when displayed on high-resolution screens.

Payment Protocol

Bitcoin Core 0.9 supports the new **payment protocol**. The payment protocol adds many important features to payment requests:

- Supports X.509 certificates and SSL encryption to verify receivers’ identity and help prevent man-in-the-middle attacks.
- Provides more detail about the requested payment to spenders.
- Allows spenders to submit transactions directly to receivers without going through the peer-to-peer network. This can speed up payment processing and work with planned features such as child-pays-for-parent transaction fees and offline NFC or Bluetooth-based payments.

Instead of being asked to pay a meaningless address, such as “mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN”, spenders are asked to pay the Common Name (CN) description from the receiver’s X.509 certificate, such as “www.bitcoin.org”.

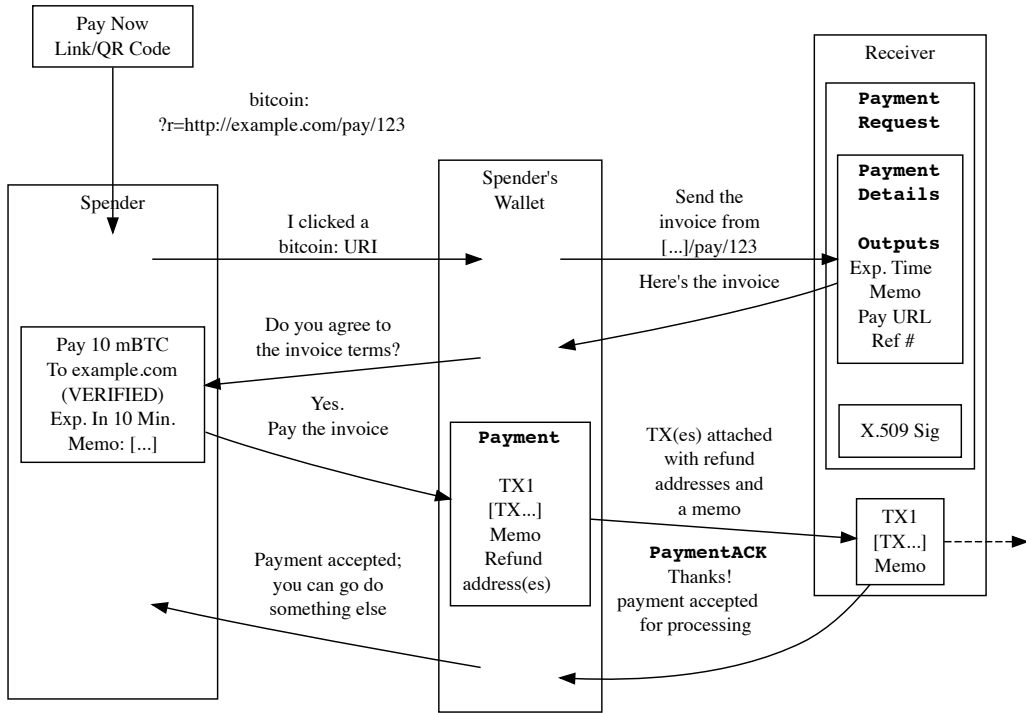
To request payment using the payment protocol, you use an extended (but backwards-compatible) `bitcoin:` URI. For example:

```
bitcoin:mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN\
?amount=0.10\
&label=Example+Merchant\
&message=Order+of+flowers+%26+chocolates\
```

&r=https://example.com/pay/mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN

None of the parameters provided above, except `r`, are required for the payment protocol—but your applications may include them for backwards compatibility with wallet programs which don't yet handle the payment protocol.

The `r` parameter tells payment-protocol-aware wallet programs to ignore the other parameters and fetch a `PaymentRequest` from the URL provided. The browser, QR code reader, or other program processing the URI opens the spender's Bitcoin wallet program on the URI.



The Bitcoin Payment Protocol As Described In BIP70

The Payment Protocol is described in depth in BIP70, BIP71, and BIP72. An example CGI program and description of all the parameters which can be used in the Payment Protocol is provided in the Developer Examples [Payment Protocol](#) subsection. In this subsection, we will briefly describe in story format how the Payment Protocol is typically used.

Charlie, the client, is shopping on a website run by Bob, the businessman. Charlie adds a few items to his shopping cart and clicks the “Checkout With Bitcoin” button.

Bob’s server automatically adds the following information to its invoice database:

- The details of Charlie’s order, including items ordered and shipping address.
- An order total in satoshis, perhaps created by converting prices in fiat to prices in satoshis.
- An expiration time when that total will no longer be acceptable.
- A pubkey script to which Charlie should send payment. Typically this will be a P2PKH or P2SH pubkey script containing a unique (never before used) secp256k1 public key.

After adding all that information to the database, Bob’s server displays a `bitcoin:` URI for Charlie to click to pay.

Charlie clicks on the `bitcoin:` URI in his browser. His browser’s URI handler sends the URI to his wallet program. The wallet is aware of the Payment Protocol, so it parses the `r` parameter and sends an HTTP GET to that URL looking for a `PaymentRequest`

message.

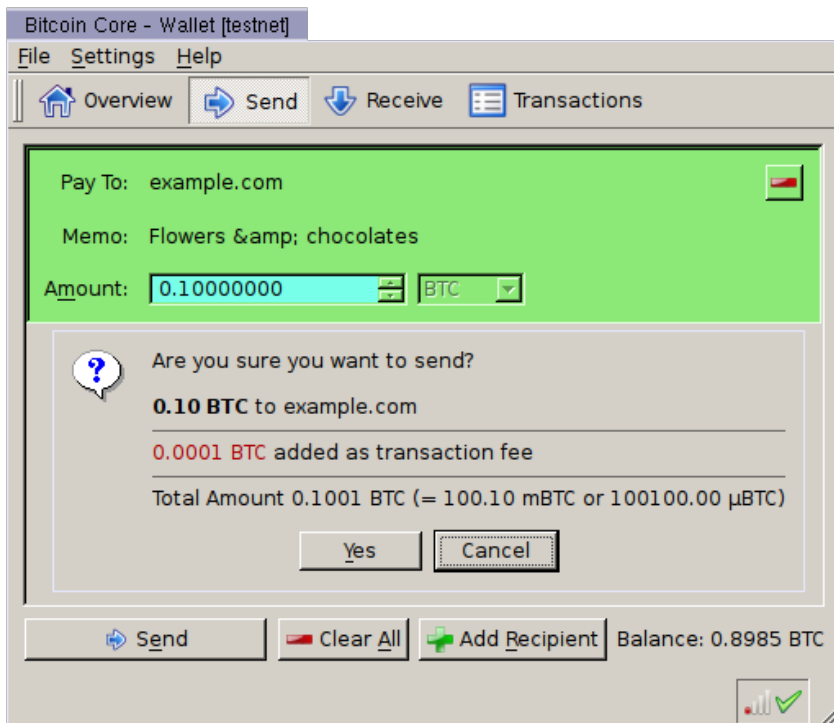
The PaymentRequest message returned may include private information, such as Charlie's mailing address, but the wallet must be able to access it without using prior authentication, such as HTTP cookies, so a publicly-accessible HTTPS URL with a guess-resistant part is typically used. The unique public key created for the payment request can be used to create a unique identifier. This is why, in the example URI above, the PaymentRequest URL contains the P2PKH address:

```
https://example.com/pay/mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN
```

After receiving the HTTP GET to the URL above, the PaymentRequest-generating CGI program on Bob's webserver takes the unique identifier from the URL and looks up the corresponding details in the database. It then creates a PaymentDetails message with the following information:

- The amount of the order in satoshis and the pubkey script to be paid.
- A memo containing the list of items ordered, so Charlie knows what he's paying for. It may also include Charlie's mailing address so he can double-check it.
- The time the PaymentDetails message was created plus the time it expires.
- A URL to which Charlie's wallet should send its completed transaction.

That PaymentDetails message is put inside a PaymentRequest message. The payment request lets Bob's server sign the entire Request with the server's X.509 SSL certificate. (The Payment Protocol has been designed to allow other signing methods in the future.) Bob's server sends the payment request to Charlie's wallet in the reply to the HTTP GET.



Charlie's wallet receives the PaymentRequest message, checks its signature, and then displays the details from the PaymentDetails message to Charlie. Charlie agrees to pay, so the wallet constructs a payment to the pubkey script Bob's server provided. Unlike a traditional Bitcoin payment, Charlie's wallet doesn't necessarily automatically broadcast this payment to the network. Instead, the wallet constructs a Payment message and sends it to the URL provided in the PaymentDetails message as an HTTP POST. Among other things, the Payment message contains:

- The signed transaction in which Charlie pays Bob.

- An optional memo Charlie can send to Bob. (There's no guarantee that Bob will read it.)
- A refund address (pubkey script) which Bob can pay if he needs to return some or all of Charlie's satoshis.

Bob's server receives the Payment message, verifies the transaction pays the requested amount to the address provided, and then broadcasts the transaction to the network. It also replies to the HTTP POSTed Payment message with a PaymentACK message, which includes an optional memo from Bob's server thanking Charlie for his patronage and providing other information about the order, such as the expected arrival date.

Charlie's wallet sees the PaymentACK and tells Charlie that the payment has been sent. The PaymentACK doesn't mean that Bob has verified Charlie's payment—see the Verifying Payment subsection below—but it does mean that Charlie can go do something else while the transaction gets confirmed. After Bob's server verifies from the block chain that Charlie's transaction has been suitably confirmed, it authorizes shipping Charlie's order.

In the case of a dispute, Charlie can generate a cryptographically-proven **receipt** out of the various signed or otherwise-proven information.

- The PaymentDetails message signed by Bob's webserver proves Charlie received an invoice to pay a specified pubkey script for a specified number of satoshis for goods specified in the memo field.
- The Bitcoin block chain can prove that the pubkey script specified by Bob was paid the specified number of satoshis.

If a refund needs to be issued, Bob's server can safely pay the refund-to pubkey script provided by Charlie. See the Refunds section below for more details.

Verifying Payment

As explained in the [Transactions](#) and [Block Chain](#) sections, broadcasting a transaction to the network doesn't ensure that the receiver gets paid. A malicious spender can create one transaction that pays the receiver and a second one that pays the same input back to himself. Only one of these transactions will be added to the block chain, and nobody can say for sure which one it will be.

Two or more transactions spending the same input are commonly referred to as a **double spend**.

Once the transaction is included in a block, double spends are impossible without modifying block chain history to replace the transaction, which is quite difficult. Using this system, the Bitcoin protocol can give each of your transactions an updating confidence score based on the number of blocks which would need to be modified to replace a transaction. For each block, the transaction gains one **confirmation**. Since modifying blocks is quite difficult, higher confirmation scores indicate greater protection.

0 confirmations: The transaction has been broadcast but is still not included in any block. Zero confirmation transactions (unconfirmed transactions) should generally not be trusted without risk analysis. Although miners usually confirm the first transaction they receive, fraudsters may be able to manipulate the network into including their version of a transaction.

1 confirmation: The transaction is included in the latest block and double-spend risk decreases dramatically. Transactions which pay sufficient transaction fees need 10 minutes on average to receive one confirmation. However, the most recent block gets replaced fairly often by accident, so a double spend is still a real possibility.

2 confirmations: The most recent block was chained to the block which includes the transaction. As of March 2014, two block replacements were exceedingly rare, and a two block replacement attack was impractical without expensive mining equipment.

6 confirmations: The network has spent about an hour working to protect the transaction against double spends and the

transaction is buried under six blocks. Even a reasonably lucky attacker would require a large percentage of the total network hashing power to replace six blocks. Although this number is somewhat arbitrary, software handling high-value transactions, or otherwise at risk for fraud, should wait for at least six confirmations before treating a payment as accepted.

Bitcoin Core provides several RPCs which can provide your program with the confirmation score for transactions in your wallet or arbitrary transactions. For example, the `listunspent` RPC provides an array of every satoshi you can spend along with its confirmation score.

Although confirmations provide excellent double-spend protection most of the time, there are at least three cases where double-spend risk analysis can be required:

1. In the case when the program or its user cannot wait for a confirmation and wants to accept unconfirmed payments.
2. In the case when the program or its user is accepting high value transactions and cannot wait for at least six confirmations or more.
3. In the case of an implementation bug or prolonged attack against Bitcoin which makes the system less reliable than expected.

An interesting source of double-spend risk analysis can be acquired by connecting to large numbers of Bitcoin peers to track how transactions and blocks differ from each other. Some third-party APIs can provide you with this type of service.

For example, unconfirmed transactions can be compared among all connected peers to see if any UTXO is used in multiple unconfirmed transactions, indicating a double-spend attempt, in which case the payment can be refused until it is confirmed. Transactions can also be ranked by their transaction fee to estimate the amount of time until they're added to a block.

Another example could be to detect a fork when multiple peers report differing block header hashes at the same block height. Your program can go into a safe mode if the fork extends for more than two blocks, indicating a possible problem with the block chain. For more details, see the [Detecting Forks subsection](#).

Another good source of double-spend protection can be human intelligence. For example, fraudsters may act differently from legitimate customers, letting savvy merchants manually flag them as high risk. Your program can provide a safe mode which stops automatic payment acceptance on a global or per-customer basis.

Issuing Refunds

Occasionally receivers using your applications will need to issue refunds. The obvious way to do that, which is very unsafe, is simply to return the satoshis to the pubkey script from which they came. For example:

- Alice wants to buy a widget from Bob, so Bob gives Alice a price and Bitcoin address.
- Alice opens her wallet program and sends some satoshis to that address. Her wallet program automatically chooses to spend those satoshis from one of its unspent outputs, an output corresponding to the Bitcoin address `mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN`.
- Bob discovers Alice paid too many satoshis. Being an honest fellow, Bob refunds the extra satoshis to the `mjSk...` address.

This seems like it should work, but Alice is using a centralized multi-user web wallet which doesn't give unique addresses to each user, so it has no way to know that Bob's refund is meant for Alice. Now the refund is a unintentional donation to the company behind the centralized wallet, unless Alice opens a support ticket and proves those satoshis were meant for her.

This leaves receivers only two correct ways to issue refunds:

- If an address was copy-and-pasted or a basic `bitcoin:` URI was used, contact the spender directly and ask them to provide

a refund address.

- If the payment protocol was used, send the refund to the output listed in the `refund_to` field of the Payment message.

Note: it would be wise to contact the spender directly if the refund is being issued a long time after the original payment was made. This allows you to ensure the user still has access to the key or keys for the `refund_to` address.

Disbursing Income (Limiting Forex Risk)

Many receivers worry that their satoshis will be less valuable in the future than they are now, called foreign exchange (forex) risk. To limit forex risk, many receivers choose to disburse newly-acquired payments soon after they're received.

If your application provides this business logic, it will need to choose which outputs to spend first. There are a few different algorithms which can lead to different results.

- A merge avoidance algorithm makes it harder for outsiders looking at block chain data to figure out how many satoshis the receiver has earned, spent, and saved.
- A last-in-first-out (LIFO) algorithm spends newly acquired satoshis while there's still double spend risk, possibly pushing that risk on to others. This can be good for the receiver's balance sheet but possibly bad for their reputation.
- A first-in-first-out (FIFO) algorithm spends the oldest satoshis first, which can help ensure that the receiver's payments always confirm, although this has utility only in a few edge cases.

Merge Avoidance

When a receiver receives satoshis in an output, the spender can track (in a crude way) how the receiver spends those satoshis. But the spender can't automatically see other satoshis paid to the receiver by other spenders as long as the receiver uses unique addresses for each transaction.

However, if the receiver spends satoshis from two different spenders in the same transaction, each of those spenders can see the other spender's payment. This is called a **merge**, and the more a receiver merges outputs, the easier it is for an outsider to track how many satoshis the receiver has earned, spent, and saved.

Merge avoidance means trying to avoid spending unrelated outputs in the same transaction. For persons and businesses which want to keep their transaction data secret from other people, it can be an important strategy.

A crude merge avoidance strategy is to try to always pay with the smallest output you have which is larger than the amount being requested. For example, if you have four outputs holding, respectively, 100, 200, 500, and 900 satoshis, you would pay a bill for 300 satoshis with the 500-satoshi output. This way, as long as you have outputs larger than your bills, you avoid merging.

More advanced merge avoidance strategies largely depend on enhancements to the payment protocol which will allow payers to avoid merging by intelligently distributing their payments among multiple outputs provided by the receiver.

Last In, First Out (LIFO)

Outputs can be spent as soon as they're received—even before they're confirmed. Since recent outputs are at the greatest risk of being double-spent, spending them before older outputs allows the spender to hold on to older confirmed outputs which are much less likely to be double-spent.

There are two closely-related downsides to LIFO:

- If you spend an output from one unconfirmed transaction in a second transaction, the second transaction becomes invalid if transaction malleability changes the first transaction.
- If you spend an output from one unconfirmed transaction in a second transaction and the first transaction’s output is successfully double spent to another output, the second transaction becomes invalid.

In either of the above cases, the receiver of the second transaction will see the incoming transaction notification disappear or turn into an error message.

Because LIFO puts the recipient of secondary transactions in as much double-spend risk as the recipient of the primary transaction, they’re best used when the secondary recipient doesn’t care about the risk—such as an exchange or other service which is going to wait for six confirmations whether you spend old outputs or new outputs.

LIFO should not be used when the primary transaction recipient’s reputation might be at stake, such as when paying employees. In these cases, it’s better to wait for transactions to be fully verified (see the [Verification subsection](#) above) before using them to make payments.

First In, First Out (FIFO)

The oldest outputs are the most reliable, as the longer it’s been since they were received, the more blocks would need to be modified to double spend them. However, after just a few blocks, a point of rapidly diminishing returns is reached. The [original Bitcoin paper](#) predicts the chance of an attacker being able to modify old blocks, assuming the attacker has 30% of the total network hashing power:

Blocks	Chance of successful modification
5	17.73523%
10	4.16605%
15	1.01008%
20	0.24804%
25	0.06132%
30	0.01522%
35	0.00379%
40	0.00095%
45	0.00024%
50	0.00006%

FIFO does have a small advantage when it comes to transaction fees, as older outputs may be eligible for inclusion in the 50,000 bytes set aside for no-fee-required high-priority transactions by miners running the default Bitcoin Core codebase. However, with transaction fees being so low, this is not a significant advantage.

The only practical use of FIFO is by receivers who spend all or most of their income within a few blocks, and who want to reduce the chance of their payments becoming accidentally invalid. For example, a receiver who holds each payment for six confirmations, and then spends 100% of verified payments to vendors and a savings account on a bi-hourly schedule.

Rebiling Recurring Payments

Automated recurring payments are not possible with decentralized Bitcoin wallets. Even if a wallet supported automatically sending non-reversible payments on a regular schedule, the user would still need to start the program at the appointed time, or leave it running all the time unprotected by encryption.

This means automated recurring Bitcoin payments can only be made from a centralized server which handles satoshis on behalf of its spenders. In practice, receivers who want to set prices in fiat terms must also let the same centralized server choose the appropriate exchange rate.

Non-automated rebilling can be managed by the same mechanism used before credit-card recurring payments became common: contact the spender and ask them to pay again—for example, by sending them a PaymentRequest `bitcoin:` URI in an HTML email.

In the future, extensions to the payment protocol and new wallet features may allow some wallet programs to manage a list of recurring transactions. The spender will still need to start the program on a regular basis and authorize payment—but it should be easier and more secure for the spender than clicking an emailed invoice, increasing the chance receivers get paid on time.

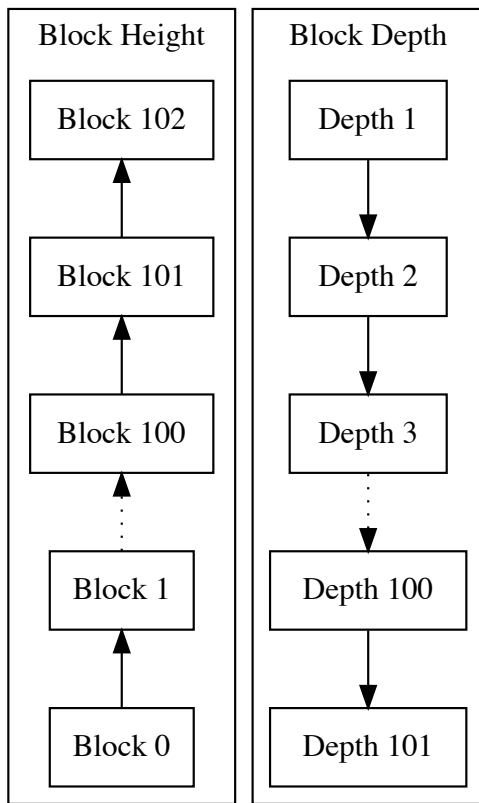
Operating Modes

Currently there are two primary methods of validating the block chain as a client: Full nodes and SPV clients. Other methods, such as server-trusting methods, are not discussed as they are not recommended.

Full Node

The first and most secure model is the one followed by Bitcoin Core, also known as a “thick” or “full chain” client. This security model assures the validity of the block chain by downloading and validating blocks from the genesis block all the way to the most recently discovered block. This is known as using the *height* of a particular block to verify the client’s view of the network.

For a client to be fooled, an adversary would need to give a complete alternative block chain history that is of greater difficulty than the current “true” chain, which is impossible due to the fact that the longest chain is by definition the true chain. After the suggested six confirmations, the ability to fool the client become intractable, as only a single honest network node is needed to have the complete state of the block chain.



Block Height Compared
To Block Depth

Simplified Payment Verification (SPV)

An alternative approach detailed in the [original Bitcoin paper](#) is a client that only downloads the headers of blocks during the initial syncing process and then requests transactions from full nodes as needed. This scales linearly with the height of the block chain at only 80 bytes per block header, or up to 4.2MB per year, regardless of total block size.

As described in the white paper, the merkle root in the block header along with a merkle branch can prove to the SPV client that the transaction in question is embedded in a block in the block chain. This does not guarantee validity of the transactions that are embedded. Instead it demonstrates the amount of work required to perform a double-spend attack.

The block's depth in the block chain corresponds to the cumulative difficulty that has been performed to build on top of that particular block. The SPV client knows the merkle root and associated transaction information, and requests the respective merkle branch from a full node. Once the merkle branch has been retrieved, proving the existence of the transaction in the block, the SPV client can then look to block *depth* as a proxy for transaction validity and security. The cost of an attack on a user by a malicious node who inserts an invalid transaction grows with the cumulative difficulty built on top of that block, since the malicious node alone will be mining this forged chain.

Potential SPV Weaknesses

If implemented naively, an SPV client has a few important weaknesses.

First, while the SPV client can not be easily fooled into thinking a transaction is in a block when it is not, the reverse is not true. A full node can simply lie by omission, leading an SPV client to believe a transaction has not occurred. This can be considered a

form of Denial of Service. One mitigation strategy is to connect to a number of full nodes, and send the requests to each node. However this can be defeated by network partitioning or Sybil attacks, since identities are essentially free, and can be bandwidth intensive. Care must be taken to ensure the client is not cut off from honest nodes.

Second, the SPV client only requests transactions from full nodes corresponding to keys it owns. If the SPV client downloads all blocks and then discards unneeded ones, this can be extremely bandwidth intensive. If they simply ask full nodes for blocks with specific transactions, this allows full nodes a complete view of the public addresses that correspond to the user. This is a large privacy leak, and allows for tactics such as denial of service for clients, users, or addresses that are disfavored by those running full nodes, as well as trivial linking of funds. A client could simply spam many fake transaction requests, but this creates a large strain on the SPV client, and can end up defeating the purpose of thin clients altogether.

To mitigate the latter issue, Bloom filters have been implemented as a method of obfuscation and compression of block data requests.

Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure that is used to test membership of an element. The data structure achieves great data compression at the expense of a prescribed false positive rate.

A Bloom filter starts out as an array of n bits all set to 0. A set of k random hash functions are chosen, each of which output a single integer between the range of 1 and n .

When adding an element to the Bloom filter, the element is hashed k times separately, and for each of the k outputs, the corresponding Bloom filter bit at that index is set to 1.

Querying of the Bloom filter is done by using the same hash functions as before. If all k bits accessed in the bloom filter are set to 1, this demonstrates with high probability that the element lies in the set. Clearly, the k indices could have been set to 1 by the addition of a combination of other elements in the domain, but the parameters allow the user to choose the acceptable false positive rate.

Removal of elements can only be done by scrapping the bloom filter and re-creating it from scratch.

Application Of Bloom Filters

Rather than viewing the false positive rates as a liability, it is used to create a tunable parameter that represents the desired privacy level and bandwidth trade-off. A SPV client creates their Bloom filter and sends it to a full node using the message `filterload`, which sets the filter for which transactions are desired. The command `filteradd` allows addition of desired data to the filter without needing to send a totally new Bloom filter, and `filterclear` allows the connection to revert to standard block discovery mechanisms. If the filter has been loaded, then full nodes will send a modified form of blocks, called a merkle block. The merkle block is simply the block header with the merkle branch associated with the set Bloom filter.

An SPV client can not only add transactions as elements to the filter, but also public keys, data from signature scripts and pubkey scripts, and more. This enables P2SH transaction finding.

If a user is more privacy-conscious, he can set the Bloom filter to include more false positives, at the expense of extra bandwidth used for transaction discovery. If a user is on a tight bandwidth budget, he can set the false-positive rate to low, knowing that this will allow full nodes a clear view of what transactions are associated with his client.

Resources: [BitcoinJ](#), a Java implementation of Bitcoin that is based on the SPV security model and Bloom filters. Used in most Android wallets.

Bloom filters were standardized for use via [BIP37](#). Review the BIP for implementation details.

Future Proposals

There are future proposals such as Unspent Transaction Output (UTXO) commitments in the block chain to find a more satisfactory middle-ground for clients between needing a complete copy of the block chain, or trusting that a majority of your connected peers are not lying. UTXO commitments would enable a very secure client using a finite amount of storage using a data structure that is authenticated in the block chain. These type of proposals are, however, in very early stages, and will require soft forks in the network.

Until these types of operating modes are implemented, modes should be chosen based on the likely threat model, computing and bandwidth constraints, and liability in bitcoin value.

Resources: [Original Thread on UTXO Commitments](#), [Authenticated Prefix Trees BIP Proposal](#)

P2P Network

The Bitcoin network protocol allows full nodes (**peers**) to collaboratively maintain a **peer-to-peer network** for block and transaction exchange. Many SPV clients also use this protocol to connect to full nodes.

Consensus rules do not cover networking, so Bitcoin programs may use alternative networks and protocols, such as the [high-speed block relay network](#) used by some miners and the [dedicated transaction information servers](#) used by some wallets that provide SPV-level security.

To provide practical examples of the Bitcoin peer-to-peer network, this section uses Bitcoin Core as a representative full node and [BitcoinJ](#) as a representative SPV client. Both programs are flexible, so only default behavior is described. Also, for privacy, actual IP addresses in the example output below have been replaced with [RFC5737](#) reserved IP addresses.

Peer Discovery

When started for the first time, programs don't know the IP addresses of any active full nodes. In order to discover some IP addresses, they query one or more DNS names (called **DNS seeds**) hardcoded into Bitcoin Core and BitcoinJ. The response to the lookup should include one or more [DNS A records](#) with the IP addresses of full nodes that may accept new incoming connections. For example, using the Unix `dig` command:

```
;; QUESTION SECTION:
;seed.bitcoin.sipa.be.      IN  A

;; ANSWER SECTION:
seed.bitcoin.sipa.be.      60  IN  A   192.0.2.113
seed.bitcoin.sipa.be.      60  IN  A   198.51.100.231
seed.bitcoin.sipa.be.      60  IN  A   203.0.113.183
[...]
```

The DNS seeds are maintained by Bitcoin community members: some of them provide dynamic DNS seed servers which automatically get IP addresses of active nodes by scanning the network; others provide static DNS seeds that are updated manually and are more likely to provide IP addresses for inactive nodes. In either case, nodes are added to the DNS seed if they

run on the default Bitcoin ports of 8333 for mainnet or 18333 for testnet.

DNS seed results are not authenticated and a malicious seed operator or network man-in-the-middle attacker can return only IP addresses of nodes controlled by the attacker, isolating a program on the attacker's own network and allowing the attacker to feed it bogus transactions and blocks. For this reason, programs should not rely on DNS seeds exclusively.

Once a program has connected to the network, its peers can begin to send it `addr` (address) messages with the IP addresses and port numbers of other peers on the network, providing a fully decentralized method of peer discovery. Bitcoin Core keeps a record of known peers in a persistent on-disk database which usually allows it to connect directly to those peers on subsequent startups without having to use DNS seeds.

However, peers often leave the network or change IP addresses, so programs may need to make several different connection attempts at startup before a successful connection is made. This can add a significant delay to the amount of time it takes to connect to the network, forcing a user to wait before sending a transaction or checking the status of payment.

To avoid this possible delay, BitcoinJ always uses dynamic DNS seeds to get IP addresses for nodes believed to be currently active. Bitcoin Core also tries to strike a balance between minimizing delays and avoiding unnecessary DNS seed use: if Bitcoin Core has entries in its peer database, it spends up to 11 seconds attempting to connect to at least one of them before falling back to seeds; if a connection is made within that time, it does not query any seeds.

Both Bitcoin Core and BitcoinJ also include a hardcoded list of IP addresses and port numbers to several dozen nodes which were active around the time that particular version of the software was first released. Bitcoin Core will start attempting to connect to these nodes if none of the DNS seed servers have responded to a query within 60 seconds, providing an automatic fallback option.

As a manual fallback option, Bitcoin Core also provides several command-line connection options, including the ability to get a list of peers from a specific node by IP address, or to make a persistent connection to a specific node by IP address. See the `-help` text for details. BitcoinJ can be programmed to do the same thing.

Resources: [Bitcoin Seeder](#), the program run by several of the seeds used by Bitcoin Core and BitcoinJ. The Bitcoin Core [DNS Seed Policy](#). The hardcoded list of IP addresses used by Bitcoin Core and BitcoinJ is generated using the [makeseeds script](#).

Connecting To Peers

Connecting to a peer is done by sending a `version` message, which contains your version number, block, and current time to the remote node. The remote node responds with its own `version` message. Then both nodes send a `verack` message to the other node to indicate the connection has been established.

Once connected, the client can send to the remote node `getaddr` and `addr` messages to gather additional peers.

In order to maintain a connection with a peer, nodes by default will send a message to peers before 30 minutes of inactivity. If 90 minutes pass without a message being received by a peer, the client will assume that connection has closed.

Initial Block Download

Before a full node can validate unconfirmed transactions and recently-mined blocks, it must download and validate all blocks from block 1 (the block after the hardcoded genesis block) to the current tip of the best block chain. This is the Initial Block Download (IBD) or initial sync.

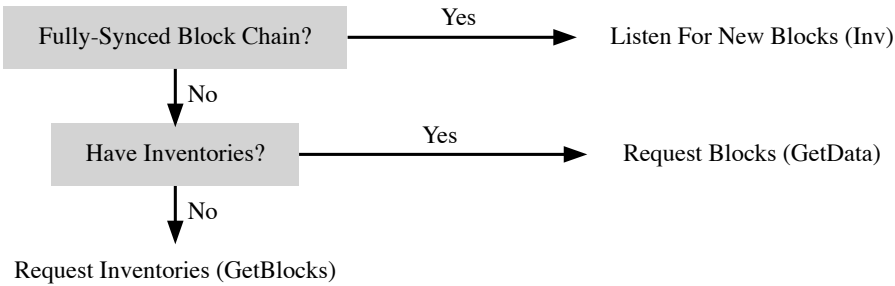
Although the word “initial” implies this method is only used once, it can also be used any time a large number of blocks need to

be downloaded, such as when a previously-caught-up node has been offline for a long time. In this case, a node can use the IBD method to download all the blocks which were produced since the last time it was online.

Bitcoin Core uses the IBD method any time the last block on its local best block chain has a block header time more than 24 hours in the past. Bitcoin Core 0.10.0 will also perform IBD if its local best block chain is more than 144 blocks lower than its local best header chain (that is, the local block chain is more than about 24 hours in the past).

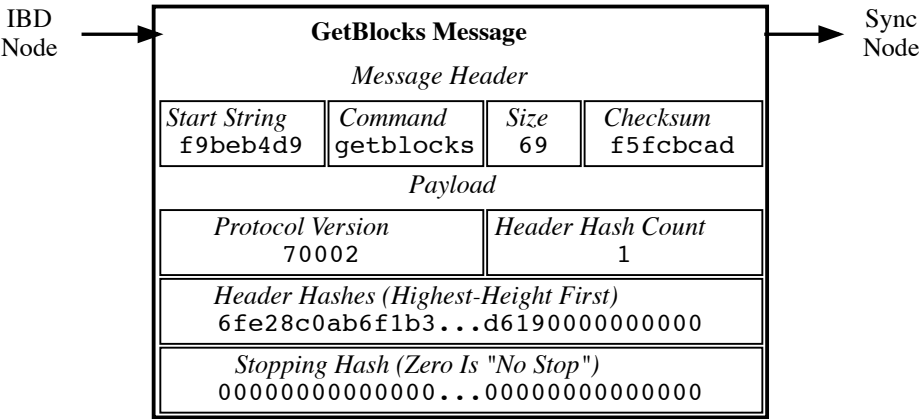
Blocks-First

Bitcoin Core (up until version 0.9.3) uses a simple initial block download (IBD) method we'll call *blocks-first*. The goal is to download the blocks from the best block chain in sequence.



Overview Of Blocks-First Initial Blocks Download (IBD)

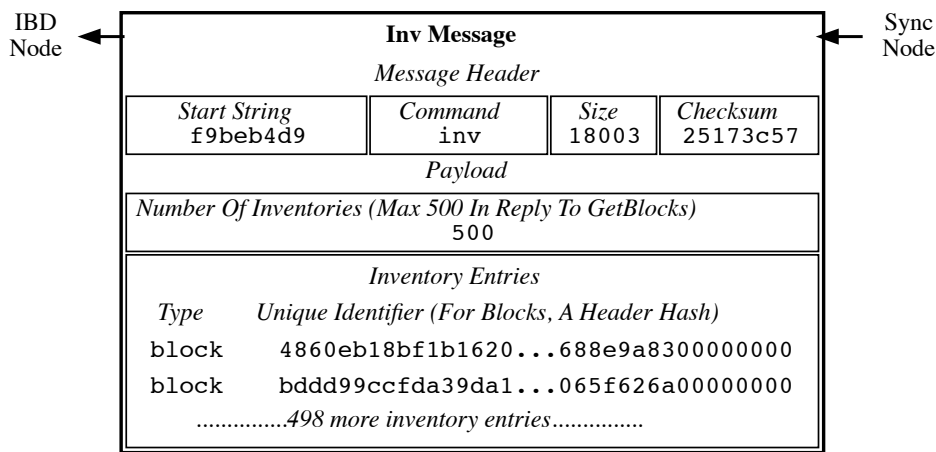
The first time a node is started, it only has a single block in its local best block chain—the hardcoded genesis block (block 0). This node chooses a remote peer, called the sync node, and sends it the `getblocks` message illustrated below.



First getblocks message sent from Initial Blocks Download (IBD) node

In the header hashes field of the `getblocks` message, this new node sends the header hash of the only block it has, the genesis block (6fe2...0000 in internal byte order). It also sets the stop hash field to all zeroes to request a maximum-size response.

Upon receipt of the `getblocks` message, the sync node takes the first (and only) header hash and searches its local best block chain for a block with that header hash. It finds that block 0 matches, so it replies with 500 block inventories (the maximum response to a `getblocks` message) starting from block 1. It sends these inventories in the `inv` message illustrated below.

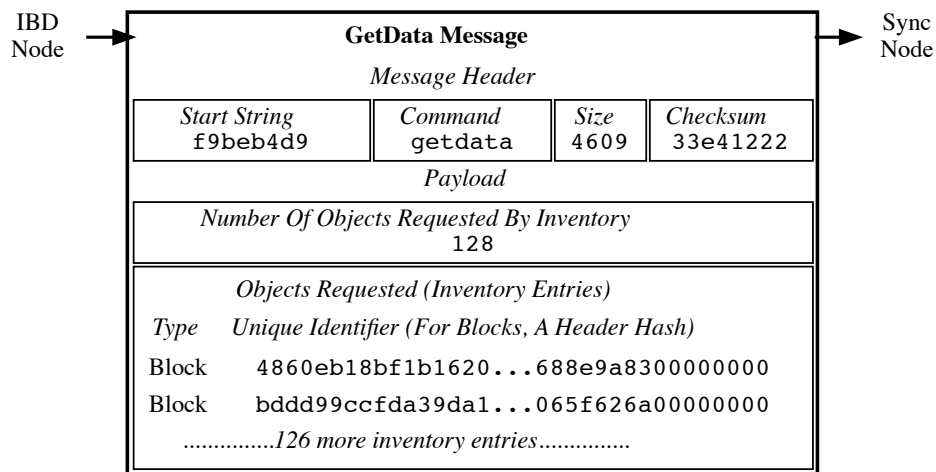


First inv message reply sent to Initial Blocks Download (IBD) node

Inventories are unique identifiers for information on the network. Each inventory contains a type field and the unique identifier for an instance of the object. For blocks, the unique identifier is a hash of the block's header.

The block inventories appear in the `inv` message in the same order they appear in the block chain, so this first `inv` message contains inventories for blocks 1 through 501. (For example, the hash of block 1 is 4860...0000 as seen in the illustration above.)

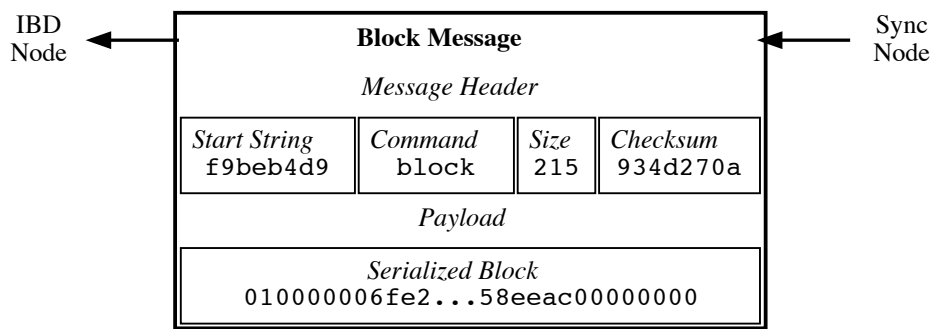
The IBD node uses the received inventories to request 128 blocks from the sync node in the `getdata` message illustrated below.



First getdata message sent from Initial Blocks Download (IBD) node

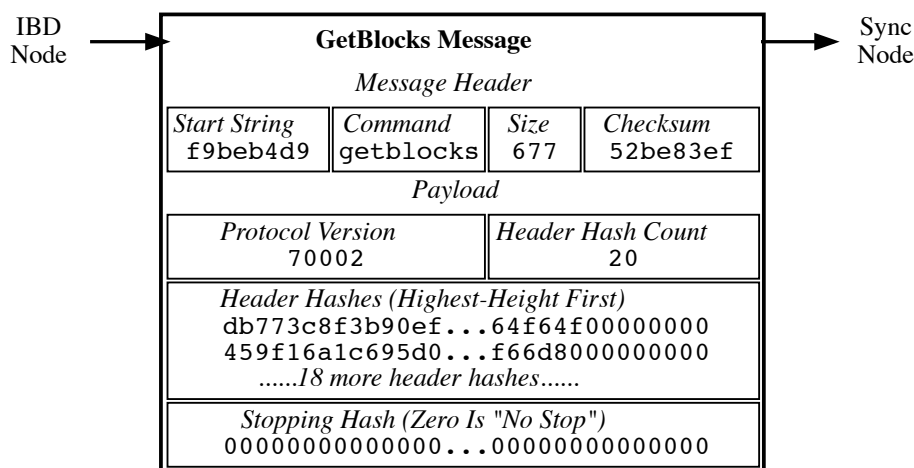
It's important to blocks-first nodes that the blocks be requested and sent in order because each block header references the header hash of the preceding block. That means the IBD node can't fully validate a block until its parent block has been received. Blocks that can't be validated because their parents haven't been received are called orphan blocks; a subsection below describes them in more detail.

Upon receipt of the `getdata` message, the sync node replies with each of the blocks requested. Each block is put into serialized block format and sent in a separate `block` message. The first `block` message sent (for block 1) is illustrated below.



First block message sent to Initial Blocks Download (IBD) node

The IBD node downloads each block, validates it, and then requests the next block it hasn't requested yet, maintaining a queue of up to 128 blocks to download. When it has requested every block for which it has an inventory, it sends another `getblocks` message to the sync node requesting the inventories of up to 500 more blocks. This second `getblocks` message contains multiple header hashes as illustrated below:



Second `getblocks` message sent from Initial Blocks Download (IBD) node

Upon receipt of the second `getblocks` message, the sync node searches its local best block chain for a block that matches one of the header hashes in the message, trying each hash in the order they were received. If it finds a matching hash, it replies with 500 block inventories starting with the next block from that point. But if there is no matching hash (besides the stopping hash), it assumes the only block the two nodes have in common is block 0 and so it sends an `inv` starting with block 1 (the same `inv` message seen several illustrations above).

This repeated search allows the sync node to send useful inventories even if the IBD node's local block chain forked from the sync node's local block chain. This fork detection becomes increasingly useful the closer the IBD node gets to the tip of the block chain.

When the IBD node receives the second `inv` message, it will request those blocks using `getdata` messages. The sync node will respond with `block` messages. Then the IBD node will request more inventories with another `getblocks` message—and the cycle will repeat until the IBD node is synced to the tip of the block chain. At that point, the node will accept blocks sent through the regular block broadcasting described in a later subsection.

Blocks-First Advantages & Disadvantages

The primary advantage of blocks-first IBD is its simplicity. The primary disadvantage is that the IBD node relies on a single sync node for all of its downloading. This has several implications:

- **Speed Limits:** All requests are made to the sync node, so if the sync node has limited upload bandwidth, the IBD node will have slow download speeds. Note: if the sync node goes offline, Bitcoin Core will continue downloading from another node—but it will still only download from a single sync node at a time.
- **Download Restarts:** The sync node can send a non-best (but otherwise valid) block chain to the IBD node. The IBD node won't be able to identify it as non-best until the initial block download nears completion, forcing the IBD node to restart its block chain download over again from a different node. Bitcoin Core ships with several block chain checkpoints at various block heights selected by developers to help an IBD node detect that it is being fed an alternative block chain history—allowing the IBD node to restart its download earlier in the process.
- **Disk Fill Attacks:** Closely related to the download restarts, if the sync node sends a non-best (but otherwise valid) block chain, the chain will be stored on disk, wasting space and possibly filling up the disk drive with useless data.
- **High Memory Use:** Whether maliciously or by accident, the sync node can send blocks out of order, creating orphan blocks which can't be validated until their parents have been received and validated. Orphan blocks are stored in memory while they await validation, which may lead to high memory use.

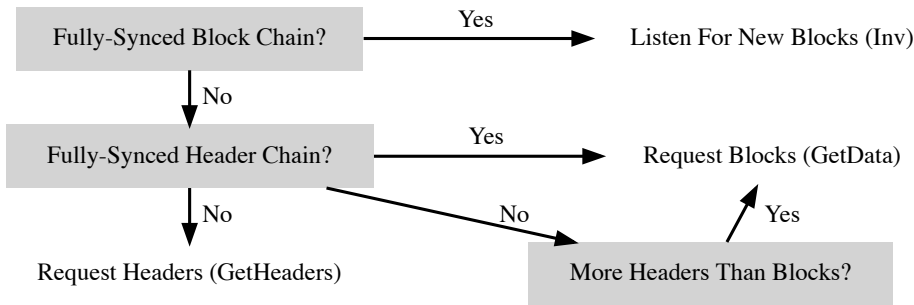
All of these problems are addressed in part or in full by the headers-first IBD method used in Bitcoin Core 0.10.0.

Resources: The table below summarizes the messages mentioned throughout this subsection. The links in the message field will take you to the reference page for that message.

Message	getblocks	inv	getdata	block
From→To	IBD→Sync	Sync→IBD	IBD→Sync	Sync→IBD
Payload	One or more header hashes	Up to 500 block inventories (unique identifiers)	One or more block inventories	One serialized block

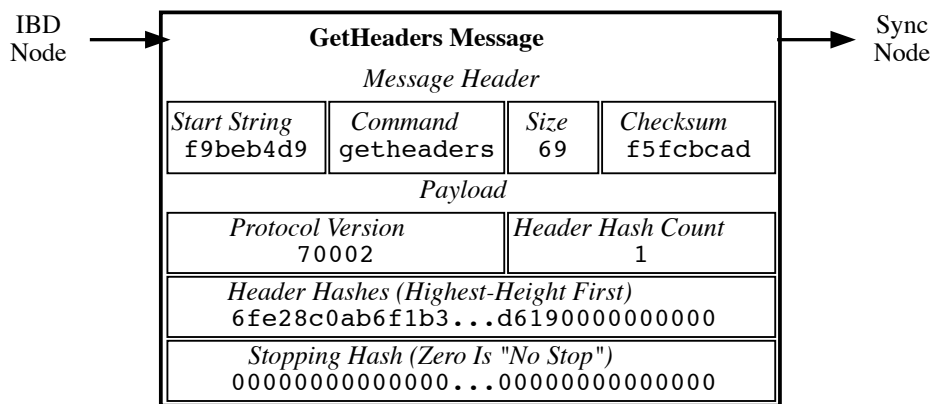
Headers-First

Bitcoin Core 0.10.0 uses an initial block download (IBD) method called *headers-first*. The goal is to download the headers for the best **header chain**, partially validate them as best as possible, and then download the corresponding blocks in parallel. This solves several problems with the older blocks-first IBD method.



Overview Of Headers-First Initial Blocks Download (IBD)

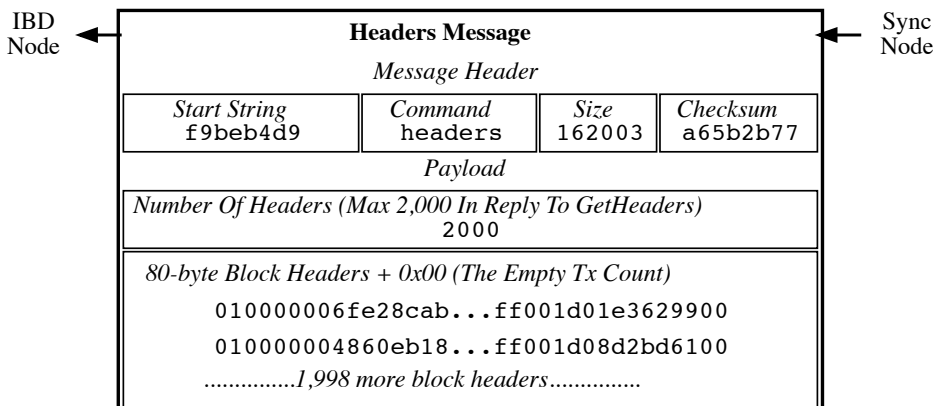
The first time a node is started, it only has a single block in its local best block chain—the hardcoded genesis block (block 0). The node chooses a remote peer, which we'll call the sync node, and sends it the `getheaders` message illustrated below.



First getheaders message sent from Initial Blocks Download (IBD) node

In the header hashes field of the `getheaders` message, the new node sends the header hash of the only block it has, the genesis block (6fe2...0000 in internal byte order). It also sets the stop hash field to all zeroes to request a maximum-size response.

Upon receipt of the `getheaders` message, the sync node takes the first (and only) header hash and searches its local best block chain for a block with that header hash. It finds that block 0 matches, so it replies with 2,000 header (the maximum response) starting from block 1. It sends these header hashes in the `headers` message illustrated below.



First headers message reply sent to Initial Blocks Download (IBD) node

The IBD node can partially validate these block headers by ensuring that all fields follow consensus rules and that the hash of the header is below the target threshold according to the nBits field. (Full validation still requires all transactions from the corresponding block.)

After the IBD node has partially validated the block headers, it can do two things in parallel:

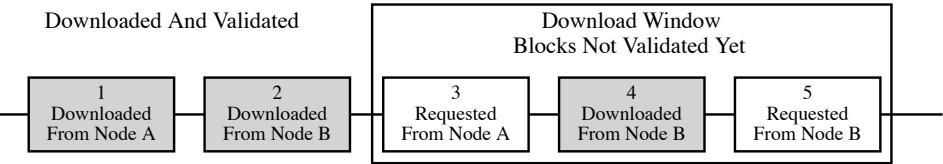
1. **Download More Headers:** the IBD node can send another `getheaders` message to the sync node to request the next 2,000 headers on the best header chain. Those headers can be immediately validated and another batch requested repeatedly until a `headers` message is received from the sync node with fewer than 2,000 headers, indicating that it has no more headers to offer. As of this writing, headers sync can be completed in fewer than 200 round trips, or about 32 MB of downloaded data.

Once the IBD node receives a `headers` message with fewer than 2,000 headers from the sync node, it sends a `getheaders` message to each of its outbound peers to get their view of best header chain. By comparing the responses, it can easily determine if the headers it has downloaded belong to the best header chain reported by any of its outbound peers. This means a dishonest sync node will quickly be discovered even if checkpoints aren't used (as long as the IBD node connects to at least one honest peer; Bitcoin Core will continue to provide checkpoints in case honest peers can't be found).

2. **Download Blocks:** While the IBD node continues downloading headers, and after the headers finish downloading, the IBD

node will request and download each block. The IBD node can use the block header hashes it computed from the header chain to create `getdata` messages that request the blocks it needs by their inventory. It doesn't need to request these from the sync node—it can request them from any of its full node peers. (Although not all full nodes may store all blocks.) This allows it to fetch blocks in parallel and avoid having its download speed constrained to the upload speed of a single sync node.

To spread the load between multiple peers, Bitcoin Core will only request up to 16 blocks at a time from a single peer. Combined with its maximum of 8 outbound connections, this means headers-first Bitcoin Core will request a maximum of 128 blocks simultaneously during IBD (the same maximum number that blocks-first Bitcoin Core requested from its sync node).



Simulated Headers-First Download Window (Real Window Is Much Larger)

Bitcoin Core's headers-first mode uses a 1,024-block moving download window to maximize download speed. The lowest-height block in the window is the next block to be validated; if the block hasn't arrived by the time Bitcoin Core is ready to validate it, Bitcoin Core will wait a minimum of two more seconds for the stalling node to send the block. If the block still hasn't arrived, Bitcoin Core will disconnect from the stalling node and attempt to connect to another node. For example, in the illustration above, Node A will be disconnected if it doesn't send block 3 within at least two seconds.

Once the IBD node is synced to the tip of the block chain, it will accept blocks sent through the regular block broadcasting described in a later subsection.

Resources: The table below summarizes the messages mentioned throughout this subsection. The links in the message field will take you to the reference page for that message.

Message	getheaders	headers	getdata	block
From→To	IBD→Sync	Sync→IBD	IBD→Many	Many→IBD
Payload	One or more header hashes	Up to 2,000 block headers	One or more block inventories derived from header hashes	One serialized block

Block Broadcasting

When a miner discovers a new block, it broadcasts the new block to its peers using one of the following methods:

- Unsolicited Block Push:** the miner sends a `block` message to each of its full node peers with the new block. The miner can reasonably bypass the standard relay method in this way because it knows none of its peers already have the just-discovered block.
- Standard Block Relay:** the miner, acting as a standard relay node, sends an `inv` message to each of its peers (both full node and SPV) with an inventory referring to the new block. The most common responses are:
 - Each blocks-first (BF) peer that wants the block replies with a `getdata` message requesting the full block.
 - Each headers-first (HF) peer that wants the block replies with a `getheaders` message containing the header hash of the highest-height header on its best header chain, and likely also some headers further back on the best header chain to allow

fork detection. That message is immediately followed by a `getdata` message requesting the full block. By requesting headers first, a headers-first peer can refuse orphan blocks as described in the subsection below.

- Each Simplified Payment Verification (SPV) client that wants the block replies with a `getdata` message typically requesting a merkle block.

The miner replies to each request accordingly by sending the block in a `block` message, one or more headers in a `headers` message, or the merkle block and transactions relative to the SPV client’s bloom filter in a `merkleblock` message followed by zero or more `tx` messages.

By default, Bitcoin Core broadcasts blocks using standard block relay, but it will accept blocks sent using either of the methods described above.

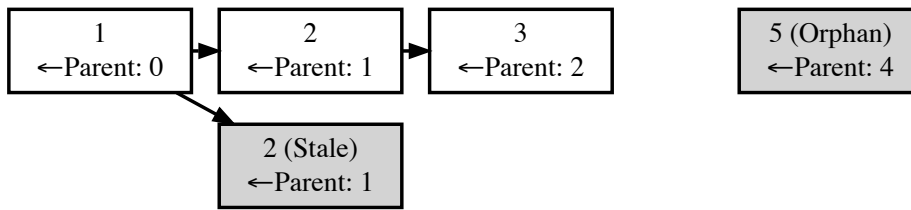
Full nodes validate the received block and then advertise it to their peers using the standard block relay method described above. The condensed table below highlights the operation of the messages described above (Relay, BF, HF, and SPV refer to the relay node, a blocks-first node, a headers-first node, and an SPV client; *any* refers to a node using any block retrieval method.)

Message	<code>inv</code>	<code>getdata</code>	<code>getheaders</code>	<code>headers</code>
From→To	Relay→Any	BF→Relay	HF→Relay	Relay→HF
Payload	The inventory of the new block	The inventory of the new block	One or more header hashes on the HF node’s best header chain (BHC)	Up to 2,000 headers connecting HF node’s BHC to relay node’s BHC
Message	<code>block</code>	<code>merkleblock</code>	<code>tx</code>	
From→To	Relay→BF/HF	Relay→SPV	Relay→SPV	
Payload	The new block in <code>serialized format</code>	The new block filtered into a merkle block	Serialized transactions from the new block that match the bloom filter	

Orphan Blocks

Blocks-first nodes may download orphan blocks—blocks whose previous block header hash field refers to a block header this node hasn’t seen yet. In other words, orphan blocks have no known parent (unlike stale blocks, which have known parents but which aren’t part of the best block chain).

Orphan blocks have no known parent, so they can't be validated



Stale blocks are valid but not part of the best block chain

When a blocks-first node downloads an orphan block, it will not validate it. Instead, it will send a `getblocks` message to the node which sent the orphan block; the broadcasting node will respond with an `inv` message containing inventories of any blocks the downloading node is missing (up to 500); the downloading node will request those blocks with a `getdata` message; and the broadcasting node will send those blocks with a `block` message. The downloading node will validate those blocks, and once the parent of the former orphan block has been validated, it will validate the former orphan block.

Headers-first nodes avoid some of this complexity by always requesting block headers with the `getheaders` message before requesting a block with the `getdata` message. The broadcasting node will send a `headers` message containing all the block headers (up to 2,000) it thinks the downloading node needs to reach the tip of the best header chain; each of those headers will point to its parent, so when the downloading node receives the `block` message, the block shouldn't be an orphan block—all of its parents should be known (even if they haven't been validated yet). If, despite this, the block received in the `block` message is an orphan block, a headers-first node will discard it immediately.

However, orphan discarding does mean that headers-first nodes will ignore orphan blocks sent by miners in an unsolicited block push.

Transaction Broadcasting

In order to send a transaction to a peer, an `inv` message is sent. If a `getdata` response message is received, the transaction is sent using `tx`. The peer receiving this transaction also forwards the transaction in the same manner, given that it is a valid transaction.

Memory Pool

Full peers may keep track of unconfirmed transactions which are eligible to be included in the next block. This is essential for miners who will actually mine some or all of those transactions, but it's also useful for any peer who wants to keep track of unconfirmed transactions, such as peers serving unconfirmed transaction information to SPV clients.

Because unconfirmed transactions have no permanent status in Bitcoin, Bitcoin Core stores them in non-persistent memory, calling them a memory pool or mempool. When a peer shuts down, its memory pool is lost except for any transactions stored by its wallet. This means that never-mined unconfirmed transactions tend to slowly disappear from the network as peers restart or as they purge some transactions to make room in memory for others.

Transactions which are mined into blocks that later become stale blocks may be added back into the memory pool. These re-added transactions may be re-removed from the pool almost immediately if the replacement blocks include them. This is the case in Bitcoin Core, which removes stale blocks from the chain one by one, starting with the tip (highest block). As each block is

removed, its transactions are added back to the memory pool. After all of the stale blocks are removed, the replacement blocks are added to the chain one by one, ending with the new tip. As each block is added, any transactions it confirms are removed from the memory pool.

SPV clients don't have a memory pool for the same reason they don't relay transactions. They can't independently verify that a transaction hasn't yet been included in a block and that it only spends UTXOs, so they can't know which transactions are eligible to be included in the next block.

Misbehaving Nodes

Take note that for both types of broadcasting, mechanisms are in place to punish misbehaving peers who take up bandwidth and computing resources by sending false information. If a peer gets a banscore above the `-banscore=<n>` threshold, he will be banned for the number of seconds defined by `-bantime=<n>`, which is 86,400 by default (24 hours).

Alerts

In case of a bug or attack, the Bitcoin Core developers provide a [Bitcoin alert service](#) with an RSS feed and users of Bitcoin Core can check the error field of the `getinfo` RPC results to get currently active alerts for their specific version of Bitcoin Core.

These messages are aggressively broadcast using the `alert` message, being sent to each peer upon connect for the duration of the alert.

These messages are signed by a specific ECDSA private key that only a small number of developers control.

Resource: More details about the structure of messages and a complete list of message types can be found in the [P2P reference section](#).

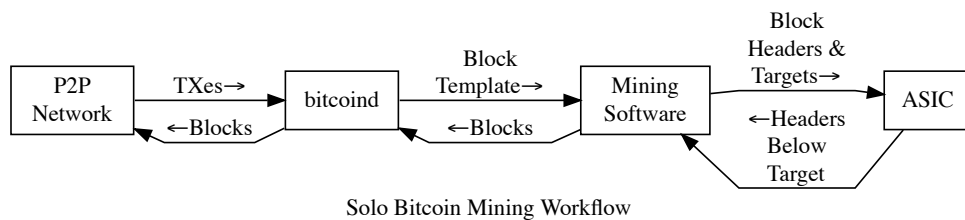
Mining

Mining adds new blocks to the block chain, making transaction history hard to modify. Mining today takes on two forms:

- Solo mining, where the miner attempts to generate new blocks on his own, with the proceeds from the block reward and transaction fees going entirely to himself, allowing him to receive large payments with a higher variance (longer time between payments)
- Pooled mining, where the miner pools resources with other miners to find blocks more often, with the proceeds being shared among the pool miners in rough correlation to the amount of hashing power they each contributed, allowing the miner to receive small payments with a lower variance (shorter time between payments).

Solo Mining

As illustrated below, solo miners typically use `bitcoind` to get new transactions from the network. Their mining software periodically polls `bitcoind` for new transactions using the `getblocktemplate` RPC, which provides the list of new transactions plus the public key to which the coinbase transaction should be sent.



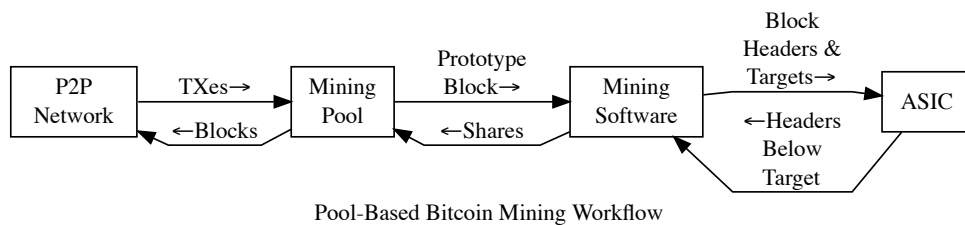
The mining software constructs a block using the template (described below) and creates a block header. It then sends the 80-byte block header to its mining hardware (an ASIC) along with a target threshold (difficulty setting). The mining hardware iterates through every possible value for the block header nonce and generates the corresponding hash.

If none of the hashes are below the threshold, the mining hardware gets an updated block header with a new merkle root from the mining software; this new block header is created by adding extra nonce data to the coinbase field of the coinbase transaction.

On the other hand, if a hash is found below the target threshold, the mining hardware returns the block header with the successful nonce to the mining software. The mining software combines the header with the block and sends the completed block to `bitcoind` to be broadcast to the network for addition to the block chain.

Pool Mining

Pool miners follow a similar workflow, illustrated below, which allows mining pool operators to pay miners based on their share of the work done. The mining pool gets new transactions from the network using `bitcoind`. Using one of the methods discussed later, each miner's mining software connects to the pool and requests the information it needs to construct block headers.



In pooled mining, the mining pool sets the target threshold a few orders of magnitude higher (less difficult) than the network difficulty. This causes the mining hardware to return many block headers which don't hash to a value eligible for inclusion on the block chain but which do hash below the pool's target, proving (on average) that the miner checked a percentage of the possible hash values.

The miner then sends to the pool a copy of the information the pool needs to validate that the header will hash below the target and that the block of transactions referred to by the header merkle root field is valid for the pool's purposes. (This usually means that the coinbase transaction must pay the pool.)

The information the miner sends to the pool is called a share because it proves the miner did a share of the work. By chance, some shares the pool receives will also be below the network target—the mining pool sends these to the network to be added to the block chain.

The block reward and transaction fees that come from mining that block are paid to the mining pool. The mining pool pays out a portion of these proceeds to individual miners based on how many shares they generated. For example, if the mining pool's target threshold is 100 times lower than the network target threshold, 100 shares will need to be generated on average to create a successful block, so the mining pool can pay 1/100th of its payout for each share received. Different mining pools use different reward distribution systems based on this basic share system.

Block Prototypes

In both solo and pool mining, the mining software needs to get the information necessary to construct block headers. This subsection describes, in a linear way, how that information is transmitted and used. However, in actual implementations, parallel threads and queuing are used to keep ASIC hashers working at maximum capacity,

getwork RPC

The simplest and earliest method was the now-deprecated Bitcoin Core `getwork` RPC, which constructs a header for the miner directly. Since a header only contains a single 4-byte nonce good for about 4 gigahashes, many modern miners need to make dozens or hundreds of `getwork` requests a second. Solo miners may still use `getwork`, but most pools today discourage or disallow its use.

getblocktemplate RPC

An improved method is the Bitcoin Core `getblocktemplate` RPC. This provides the mining software with much more information:

1. The information necessary to construct a coinbase transaction paying the pool or the solo miner's `bitcoin` wallet.
2. A complete dump of the transactions `bitcoin` or the mining pool suggests including in the block, allowing the mining software to inspect the transactions, optionally add additional transactions, and optionally remove non-required transactions.
3. Other information necessary to construct a block header for the next block: the block version, previous block hash, and bits (target).
4. The mining pool's current target threshold for accepting shares. (For solo miners, this is the network target.)

Using the transactions received, the mining software adds a nonce to the coinbase extra nonce field and then converts all the transactions into a merkle tree to derive a merkle root it can use in a block header. Whenever the extra nonce field needs to be changed, the mining software rebuilds the necessary parts of the merkle tree and updates the time and merkle root fields in the block header.

Like all `bitcoin` RPCs, `getblocktemplate` is sent over HTTP. To ensure they get the most recent work, most miners use [HTTP longpoll](#) to leave a `getblocktemplate` request open at all times. This allows the mining pool to push a new `getblocktemplate` to the miner as soon as any miner on the peer-to-peer network publishes a new block or the pool wants to send more transactions to the mining software.

Stratum

A widely used alternative to `getblocktemplate` is the [Stratum mining protocol](#). Stratum focuses on giving miners the minimal information they need to construct block headers on their own:

1. The information necessary to construct a coinbase transaction paying the pool.
2. The parts of the merkle tree which need to be re-hashed to create a new merkle root when the coinbase transaction is updated with a new extra nonce. The other parts of the merkle tree, if any, are not sent, effectively limiting the amount of data which

needs to be sent to (at most) about a kilobyte at current transaction volume.

3. All of the other non-merkle root information necessary to construct a block header for the next block.
4. The mining pool's current target threshold for accepting shares.

Using the coinbase transaction received, the mining software adds a nonce to the coinbase extra nonce field, hashes the coinbase transaction, and adds the hash to the received parts of the merkle tree. The tree is hashed as necessary to create a merkle root, which is added to the block header information received. Whenever the extra nonce field needs to be changed, the mining software updates and re-hashes the coinbase transaction, rebuilds the merkle root, and updates the header merkle root field.

Unlike `getblocktemplate`, miners using Stratum cannot inspect or add transactions to the block they're currently mining. Also unlike `getblocktemplate`, the Stratum protocol uses a two-way TCP socket directly, so miners don't need to use HTTP longpoll to ensure they receive immediate updates from mining pools when a new block is broadcast to the peer-to-peer network.

Resources: The GPLv3 [BFGMiner](#) mining software and AGPLv3 [Eloipool](#) mining pool software are widely-used among miners and pools. The [libblkmaker](#) C library and [python-blkmaker](#) library, both MIT licensed, can interpret GetBlockTemplate for your programs.