Keys, Addresses, Wallets

Introduction

Ownership of bitcoin is established through *digital keys*, *bitcoin addresses* and *digital signatures*. The digital keys are not actually stored in the network, but are instead created and stored by end-users in a file, or simple database, called a *wallet*. The digital keys in a user's wallet are completely independent of the bitcoin protocol and can be generated and managed by the user's wallet software without reference to the blockchain or access to the Internet. Keys enable many of the interesting properties of bitcoin, including de-centralized trust and control, ownership attestation and the cryptographic-proof security model.

The digital keys within each user's wallet allow the user to sign transactions, thereby providing cryptographic proof of the ownership of the bitcoins sourced by the transaction. Keys come in pairs consisting of a private (secret) and public key. Think of the public key as similar to a bank account number and the private key as similar to the secret PIN number, or signature on a cheque that provides control over the account. These digital keys are very rarely seen by the users of bitcoin. For the most part, they are stored inside the wallet file and managed by the bitcoin wallet software.

In the payment portion of a bitcoin transaction, the recipient's public key is represented by its digital fingerprint, called a *bitcoin address*, which is used in the same way as the beneficiary name on a cheque (i.e. "Pay to the order of"). In most cases, a bitcoin address is generated from and corresponds to a public key. However, not all bitcoin addresses represent public keys; they can also represent other beneficiaries such as scripts, as we will see later in this chapter. This way, bitcoin addresses abstract the recipient of funds, making transaction destinations flexible, similar to paper cheques: a single payment instrument that can be used to pay into people's accounts, company accounts, pay for bills or pay to cash. The bitcoin address is the only representation of the keys that users will routinely see, as this is the part they need to share with the world.

In this chapter we will introduce wallets, which contain cryptographic keys. We will look at how keys are generated, stored and managed. We will review the various encoding formats used to represent private and public keys, addresses and script addresses. Finally we will look at special uses of keys: to sign messages, to prove ownership and to create vanity addresses and paper wallets.

Keys

Public key cryptography and crypto-currency

Public key cryptography was invented in the 1970s and is a mathematical foundation for computer and information security.

Since the invention of public key cryptography, several suitable mathematical functions, such as prime number exponentiation and elliptic curve multiplication, have been discovered. These mathematical functions are practically irreversible, meaning that they are easy to calculate in one direction and infeasible to calculate in the opposite direction. Based on these mathematical functions, cryptography enables the creation of digital secrets and unforgeable digital signatures. Bitcoin uses elliptic curve multiplication as the basis for its public key cryptography.

In bitcoin, we use public key cryptography to create a key pair that controls access to bitcoins. The key pair consists of a private key and — derived from it — a unique public key. The public key is used to receive bitcoins, and the private key is used to sign transactions to spend those bitcoins.

There is a mathematical relationship between the public and the private key that allows the private key to be used to generate signatures on messages. This signature can be validated against the public key without revealing the private key.

When spending bitcoins, the current bitcoin owner presents their public key and a signature (different each time, but created from the same private key; see [signature]) in a transaction to spend those bitcoins. Through the presentation of the public key and signature everyone in the bitcoin network can verify and accept the transaction as valid, confirming that the person transferring the bitcoins owned them at the time of the transfer.

Tip In most implementations, the private and public keys are stored together as a *key pair* for convenience. However, it is trivial to reproduce the public key if one has the private key, so storing only the private key is also possible.

Private and Public Keys

A bitcoin wallet contains a collection of key pairs, each consisting of a private key and a public key. The private key (k) is a number, usually picked at random. From the private key, we use elliptic curve multiplication, a one-way cryptographic function, to generate a public key (K). From the public key (K), we use a one-way cryptographic hash function to generate a bitcoin address (A). In this section we will start with generating the private key, look at the elliptic curve math that is used to turn that into a public key, and finally, generate a bitcoin address from the public key. The relationship between private key, public key and bitcoin address is shown below:

Figure 1. Private Key, Public Key and Bitcoin Address

Private Keys

A `private key` is simply a number, picked at random. Ownership and control over the private key is the root of user control over all funds associated with the corresponding bitcoin address. The private key is used to create signatures that are required to spend bitcoins by proving ownership of funds used in a transaction. The private key must remain secret at all times, as revealing it to a third party is equivalent to giving them control over the bitcoins secured by that key. The private key must also be backed up and protected from accidental loss, since if lost it cannot be recovered and the funds secured by it are forever lost too.

Tip The bitcoin private key is just a number. A public key can be generated from any private key. Therefore, a public key can be generated from any number, up to 256 bits long. You can pick your keys randomly using a method as simple as dice, pencil and paper.

Generating a private key from a random number

The first and most important step in generating keys is to find a secure source of entropy, or randomness. Creating a bitcoin key is essentially the same as "Pick a number between 1 and 2^{256}". The exact method you use to pick that number does not matter as long as it is not predictable or repeatable. Bitcoin software uses the underlying operating system's random number generators to produce 256 bits of entropy (randomness). Usually, the OS random number generator is initialized by a human source of randomness, which is why you may be asked to wiggle your mouse around for a few seconds. For the truly paranoid, nothing beats dice, pencil and paper.

More accurately, the private key can be any number between `1` and `n - 1`, where n is a constant ($n = 1.158 * 10^{77}$ or slightly less than 2^{256}) defined as the order of the elliptic curve used in bitcoin (see [elliptic_curve]). To create such a key, we randomly pick a 256-bit number and check that it is less than `n - 1`. In programming terms, this is usually achieved by feeding a larger string of random bits, collected from a cryptographically-secure source of randomness, into the SHA-256 hash algorithm which will conveniently produce a 256-bit number. If the result is less than `n - 1`, we have a suitable private key. Otherwise, we simply try again with another random number.

Tip Do not try and design your own pseudo random number generator (PRNG). Use a cryptographically-secure pseudo-random number generator (CSPRNG) with a seed from a source of sufficient entropy, the choice of which depends on the operating-system. Correct implementation of the CSPRNG is critical to the security of the keys. DIY is highly discouraged unless you are a professional cryptographer.

Below is a randomly generated private key shown in hexadecimal format (256 binary digits shown as 64 hexadecimal digits, each 4 bits):

Randomly generated private key (k)

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```

Tip The size of bitcoin's private key, 2^{256} is an unfathomably large number. It is approximately 10^{77} in decimal. The visible universe is estimated to contain 10^{80} atoms.

To generate a new key with the Bitcoin Core Client (see [ch03_bitcoin_client]), use the `getnewaddress` command. For security reasons it displays the public key only, not the private key. To ask bitcoind to expose the private key, use the `dumpprivkey` command. The `dumpprivkey` shows the private key in a base-58 checksum encoded format called the Wallet Import Format (WIF), which we will examine in more detail in [priv_formats]. Here's an example of generating and displaying a private key using these two commands:

```
$ bitcoind getnewaddress
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

The `dumpprivkey` command opens the wallet and extracts the private key that was generated by the `getnewaddress` command. It is not otherwise possible for bitcoind to know the private key from the public key, unless they are both stored in the wallet.

You can also use the command-line `sx tools` (see [sx_tools]) to generate and display private keys:

New key with sx tools

```
$ sx newkey
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Tip A private key is just a number. A public key can be generated from any number, up to 256 bits long. You can pick your keys randomly using just a coin, pencil and paper. Toss a coin 256 times and you have the binary digits of a random private key you can use in a bitcoin wallet. Keys really are just a pair of numbers, one calculated from the other.

Public Keys

The public key is calculated from the private key using elliptic curve multiplication, which is irreversible: \(K = k * G\)+ where `k` is the private key, `G` is a constant point called the *Generator Point* and `K` is the resulting public key. The reverse operation, division — calculating `k` if you know `K` — is as difficult as trying all possible values of `k`, i.e. a brute-force search. Before we demonstrate how to generate a public key from a private key, let's look at Elliptic Curve Cryptography in a bit more detail.

Elliptic Curve Cryptography Explained

Elliptic Curve Cryptography is a type of asymmetric or public-key cryptography based on the discrete logarithm problem as expressed by addition and multiplication on the points of an elliptic curve.

Below we see an example of an elliptic curve, similar to that used by bitcoin:

Figure 2. An Elliptic Curve

Bitcoin uses a specific elliptic curve and set of mathematical constants, as defined in a standard called `secp256k1`, established by the National Institute of Standards and Technology (NIST). The `secp256k1` curve is defined by the following function, which produces an elliptic curve:

\begin{equation} {y^2 = (x^3 \+ 7)} \text{over} \mathbb{F}_p \end{equation}

or

\begin{equation} {y^2 \mod p = (x^3 + 7) \mod p} \end{equation}

The `mod p` (modulo prime number p) indicates that this curve is over a finite field of prime order `p`, also written as \(\mathbb{F}_p\), where p = $2^{256} - 2^{32} - 2^{9} - 2^{8} - 2^{7} - 2^{6} - 2^{4} - 1$\)], a very large prime number.

Because this curve is defined over a finite field of prime order instead of over the real numbers it looks like a pattern of dots scattered in two dimensions, which makes it difficult to visualize. However, the math is identical as that of an elliptic curve over the real numbers shown above. As an example, below is the same elliptic curve over a much smaller finite field of prime order 17, showing a pattern of dots on a grid. The `secp256k1` bitcoin elliptic curve can be thought of as a much more complex pattern of dots on a unfathomably large grid.

Figure 3. Elliptic Curve Cryptography: Visualizing an elliptic curve over F(p), with p=17

Generating a public key

Starting with a private key in the form of a randomly generated number `k`, we multiply it by a predetermined point on the curve called the *generator point* `G` to produce another point somewhere else on the curve, which is the corresponding public key `K`. The generator point is specified as part of the `secp256k1` standard and is always the same for all keys in bitcoin.

\begin{equation} {K = k * G} \end{equation}

where `k` is the private key, `G` is a fixed point on the curve called the *generator point*, and `K` is the resulting public key, another point on the curve. Since the generator point is always the same, a private key k multiplied with G will always produce the same public key K.

Implementing the elliptic curve multiplication above, we take the private key generated previously and multiply it by G:

Multiply the private key k with the generator point G to find the public key K

```
K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G
```

Public Key K defined as a point `K = (x,y)`

```
K = (x, y)
where,
x = F028892BAD...DC341A
y = 07CF33DA18...505BDB
```

To visualize multiplication of a point with an integer, we will use the simpler elliptic curve over the real numbers - remember, the math is the same. Our goal is to find the multiple kG of the generator point G. That is the same as adding G to itself, k times in a row. In elliptic curves, adding a point to itself is the equivalent of drawing a tangent line on the point and finding where it intersects the curve again, then reflecting that point on the x-axis.

Starting with the generator point G, we take the tangent of the curve at G until it crosses the curve again at another point. This new point is -2G. Reflecting that point across the x-axis gives us 2G. If we take the tangent at 2G, it crosses the curve at -4G, which again we reflect on the x-axis to find G. Continuing this process, we can bounce around the curve finding the multiples of G, 2G, 4G, 8G, etc. As you can see, a randomly selected large number k, when multiplied against the generator point G is like bouncing around the curve k times, until we land on the point kG which is the public key. This process is irreversible, meaning that it is infeasible to find the factor k (the secret k) in any way other than trying all multiples of G (1G, 2G, 4G, etc) in a brute-force search for k. Since k can be an enormous number, that brute-force search would take an infeasible amount of computation and time.

Figure 4. Elliptic Curve Cryptography: Visualizing the multiplication of a point G by an integer k on an elliptic curve

Tip A private key can be converted into a public key, but a public key cannot be converted back into a private key because the math only works one

> way.

Bitcoin Addresses

An address is a string of digits and characters that can be shared with anyone who wants to send you money. In bitcoin, addresses produced from public keys begin with the digit "1". The bitcoin address is what appears most commonly in a transaction as the "recipient" of the funds. If we were to compare a bitcoin transaction to a paper cheque, the bitcoin address is the beneficiary, which is what we write on the line after "Pay to the order of". On a paper cheque, that beneficiary can sometimes be the name of a bank account holder, but can also include corporations, institutions or even cash. Because paper cheques do not need to specify an account, but rather use an abstract name as the recipient of funds, that makes paper cheques very flexible as payment instruments. Bitcoin transactions use a similar abstraction, the bitcoin address, to make them very flexible. A bitcoin address can represent the owner of a private/public key pair, or it can represent something else, such as a payment script, as we will see in [p2sh]. For now, let's examine the simple case, a bitcoin address that represents, and is derived from, a public key.

A bitcoin address derived from a public key is a string of numbers and letters that begins with the number one, such as `1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy`. The bitcoin address is derived from the public key through the use of one-way cryptographic hashing; a "hashing algorithm" or simply "hash algorithm" is a one-way function that produces a fingerprint or "hash" of an arbitrary sized input. Cryptographic hash functions are used extensively in bitcoin: in bitcoin addresses, script addresses and in the mining "Proof-of-Work" algorithm. The algorithms used to make a bitcoin address from a public key are the Secure Hash Algorithm (SHA) and the RACE Integrity Primitives Evaluation Message Digest (RIPEMD), specifically SHA256 and RIPEMD160.

Starting with the public key K, we compute the SHA256 hash and then compute the RIPEMD160 hash of the result, producing a 160 bit (20 byte) number:

\begin{equation} {A = RIPEMD160(SHA256(K))} \end{equation}

where K is the public key and A is the resulting bitcoin address.

Bitcoin addresses are almost always presented to users in an encoding called "Base58Check" (see [base58check] below), which uses 58 characters (a base-58 number system) and a checksum to help human readability, avoid ambiguity and protect against errors in address transcription and entry. Base58Check is also used in many other ways in bitcoin, whenever there is a need for a user to read and correctly transcribe a number, such as a bitcoin address, a private key, an encrypted key, or a script hash. In the next section we will examine the mechanics of Base58Check encoding and decoding, and the resulting representations.

Figure 5. Public Key to Bitcoin Address: Conversion of a public key into a bitcoin address

Base58 and Base58Check Encoding

Base-58 Encoding

In order to represent long numbers in a compact way, using fewer symbols, many computer systems use mixed-alphanumeric representations with a base (or radix) higher than 10. For example, whereas the traditional decimal system uses the ten numerals 0 through 9, the hexadecimal system uses sixteen, with the letters A through F as the six additional symbols. A number represented in hexadecimal format is shorter than the equivalent decimal representation. Even more compact, Base-64 representation uses 26 lower case letters, 26 capital letters, 10 numerals and two more characters such as "+" and "/" to transmit binary data over text-based media such as email. Base-64 is most commonly used to add binary attachments to email. Base-58 is a text-based binary-encoding format developed for use in bitcoin and used in many other crypto-currencies. It offers a balance between compact representation, readability and error detection and prevention. Base-58 is a subset of Base-64, using the upper and lower case letters and numbers but omitting some characters that are frequently mistaken for one another and can appear identical when displayed in certain fonts. Specifically, Base-58 is Base-64 without the 0 (number zero), O (capital o), l (lower L), I (capital i) and the symbols "\+" and "/". Or, more simply, it is a set of lower and capital letters and numbers without the four (0, O, l, I) mentioned above.

Bitcoin's Base-58 Alphabet

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

Base58Check Encoding

To add extra security against typos or transcription errors, Base58Check is a Base-58 encoding format, frequently used in bitcoin, which has a built-in error-checking code. The checksum is an additional four bytes added to the end of the data that is being encoded. The checksum is derived from the hash of the encoded data and can therefore be used to detect and prevent transcription and typing errors. When presented with a Base58Check code, the decoding software will calculate the checksum of the data and compare it to the checksum included in the code. If the two do not match, that indicates that an error has been introduced and the Base58Check data is invalid. For example, this prevents a mistyped bitcoin address from being accepted by the wallet software as a valid destination, an error which would otherwise result in loss of funds.

To convert data (a number) into a Base58Check format, we first add a prefix to the data, called the "version byte", which serves to easily identify the type of data that is encoded. For example, in the case of a bitcoin address the prefix is zero (0x00 in hex), whereas the prefix used when encoding a private key is 128 (0x80 in hex). A list of common version prefixes is shown below in [base58check_versions].

Next compute the "double-SHA" checksum, meaning we apply the SHA256 hash-algorithm twice on the previous result (prefix and data): `checksum = SHA256(SHA256(prefix\+data))` From the resulting 32-byte hash (hash-of-a-hash), we take only the first four bytes. These four bytes serve as the error-checking code, or checksum. The checksum is concatenated (appended) to the end.

The result of the above is now a prefix, the data and a checksum, concatenated (bytewise). This result is encoded using the base-58 alphabet described in the section above.

Figure 6. Base58Check Encoding: A base-58, versioned and checksummed format for unambiguously encoding bitcoin data

In bitcoin, most of the data presented to the user is Base58Check encoded to make it compact, easy to read and easy to detect errors. The version prefix in Base58Check encoding is used to create easily distinguishable formats, which when encoded in Base-58 contain specific characters at the beginning of the Base58Check encoded payload, making it easy for humans to identify the type of data that is encoded and how to use it. This is what differentiates, for example, a Base58Check encoded bitcoin address that starts with a "1" from a Base58Check encoded private key WIF format that starts with a "5". Some example version prefixes and the resulting Base-58 characters are shown below:

Table 1. Base58Check Version Prefix and Encoded Result Examples

Type	Version prefix (hex)	Base-58 result prefix
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K or L
BIP38 Encrypted Private Key	0x0142	6P
BIP32 Extended Public Key	0x0488B21E	xpub

Key Formats

Private Key Formats

The private key can be represented in a number of different formats, all of which correspond to the same 256-bit number. These formats include:

Table 2. Private Key Representations (Encoding Formats)

Type	Prefix	Description

Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base-58 with version prefix of 128 and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

The private key we generated earlier can be represented as:

Table 3. Example: Same Key, Different Formats

Format	Private Key
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

All of the above representations are different ways of showing the same number, the same private key. They look different, but any one format can easily be converted to any other format.

Decode from Base58Check to Hex

The sx-tools package (See [sx_tools]) makes Base58Check format decoding easy on the command line. We use the base58check-decode command:

```
$ sx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128
```

The result is the hexadecimal key, followed by the Wallet Import Format (WIF) version prefix 128

Encode from Hex to Base58Check

To encode into Base58Check, we provide the hex private key, followed by the Wallet Import Format (WIF) version prefix 128

```
$ sx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Encode from Hex (Compressed Key) to Base58Check encoding

To encode into Base58Check as a "compressed" private key (see [comp_priv]), we add the suffix `01` to the end of the hex key and then encode as above:

```
$ sx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 128
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

The resulting WIF-compressed format, starts with a "K". This denotes that the private key within has a suffix of "01" and will be used to produce compressed public keys only (See [comp_pub] below)

Public Key Formats

Public keys are also presented in different ways, most importantly as either *compressed* or *uncompressed* public keys.

As we saw previously, the public key is a point on the elliptic curve consisting of a pair of coordinates `(x,y)`. It is usually presented with the prefix `04` followed by two 256-bit numbers, one for the x-coordinate of the point, the other for the y-coordinate. The prefix `04` is used to distinguish uncompressed public keys from compressed public keys that begin with a `02` or a `03`.

Here's the public key generated by the private key we created above, shown as the coordinates `x` and `y`.

Public Key K defined as a point `K = (x,y)`

```
x = F028892BAD...DC341A
y = 07CF33DA18...505BDB
```

Here's the same public key shown as a 520-bit number (130 hex digits) with the prefix `04` followed by `x` and then `y` coordinates, as `04 x y`:

Uncompressed Public Key K shown in hex (130 hex digits) as `04xy`

```
K = 04F028892BAD...505BDB
```

Compressed Public Keys

Compressed public keys were introduced to bitcoin to reduce the size of transactions and conserve disk space on nodes that store the bitcoin blockchain database. Most transactions include the public key, required to validate the owner's credentials and spend the bitcoin. Each public key requires 520 bits (prefix \+ x \+ y), which when multiplied by several hundred transactions per block, or tens of thousands of transactions per day, adds a significant amount of data to the blockchain.

As we saw in the section [pubkey] above, a public key is a point (x,y) on an elliptic curve. Since the curve expresses a mathematical function, a point on the curve represents a solution to the equation and therefore if we know the x-coordinate we can calculate the y-coordinate by solving the equation $y^2 \bmod p = (x^3 + 7) \bmod p$. That allows us to store only the x-coordinate of the public key point, omitting the y-coordinate and reducing the size of the key and the space required to store it by 256 bits. An almost 50% reduction in size in every transaction adds up to a lot of data saved over time!

Whereas uncompressed public keys have a prefix of `04`, compressed public keys start with either a `02` or a `03` prefix. Let's look at why there are two possible prefixes: since the left side of the equation is y^2, that means the solution for y is a square root, which can have a positive or negative value. Visually, this means that the resulting y-coordinate can be above the x-axis or below the x-axis. As you can see from the graph of the elliptic curve, the curve is symmetric, meaning it is reflected like a mirror by the x-axis. So, while we can omit the y-coordinate we have to store the *sign* of y (positive or negative), or in other words we have to remember if it was above or below the x-axis, as each of those options represents a different point and a different public key. When calculating the elliptic curve in binary arithmetic on the finite field of prime order p, the y coordinate is either even or odd, which corresponds to the positive/negative sign as explained above. Therefore, to distinguish between the two possible values of y, we store a `compressed public key` with the prefix `02` if the `y` is even, and `03` if it is odd, allowing the software to correctly deduce the y-coordinate from the x-coordinate and uncompress the public key to the full coordinates of the point.

Figure 7. Public Key Compression

Here's the same public key generated previously, shown as a `compressed public key` stored in 264-bits (66 hex digits) with the prefix `03` indicating the `y` coordinate is odd:

Compressed Public Key K shown in hex (66 hex digits) as `K = {02 or 03} x`

```
K = 03F028892BAD...DC341A
```

The compressed public key, above, corresponds to the same private key, meaning that it is generated from the same private key. However it looks different from the uncompressed public key. More importantly, if we convert this compressed public key to a bitcoin address using the double-hash function (RIPEMD160(SHA256(K))) it will produce a *different* bitcoin address. This can be confusing, because it means that a single private key can produce a public key expressed in two different formats (compressed and uncompressed) which produce two different bitcoin addresses. However, the private key is identical for both bitcoin addresses.

Compressed public keys are gradually becoming the default across bitcoin clients, which is having a significant impact on reducing the size of transactions and therefore the blockchain. However, not all clients support compressed public keys yet. Newer clients that support compressed public keys have to account for transactions from older clients which do not support compressed public keys. This is especially important when a wallet application is importing private keys from another bitcoin wallet application, because the new wallet needs to scan the blockchain to find transactions corresponding to these imported keys. Which bitcoin addresses should the bitcoin wallet scan for? The bitcoin addresses produced by uncompressed public keys, or the bitcoin addresses produced by compressed public keys? Both are valid bitcoin addresses, and can be signed for by the private key, but they are different addresses!

To resolve this issue, when private keys are exported from a wallet, the Wallet Import Format that is used to represent them is implemented differently in newer bitcoin wallets, to indicate that these private keys have been used to produce *compressed* public keys and therefore *compressed* bitcoin addresses. This allows the importing wallet to distinguish between private keys originating from older or newer wallets and search the blockchain for transactions with bitcoin addresses corresponding to the uncompressed, or the compressed public keys respectively. Let’s look at how this works in more detail, in the next section.

Compressed Private Keys

Ironically, the name "compressed private key" is misleading, because when a private key is exported as WIF-compressed it is actually one byte *longer* than an "uncompressed" private key. That is because it has the added 01 suffix which signifies it comes from a newer wallet and should only be used to produce compressed public keys. Private keys are not compressed and cannot be compressed. The term "compressed private key" really means "private key from which compressed public keys should be derived", whereas "uncompressed private key" really means "private key from which uncompressed public keys should be derived". You should only refer to the export format as "WIF-compressed" or "WIF" and not refer to the private key as "compressed" to avoid further confusion.

Remember, these formats are *not* used interchangeably. In a newer wallet that implements compressed public keys, the private keys will only ever be exported as WIF-compressed (K/L prefix). If the wallet is an older implementation and does not use compressed public keys, the private keys will only ever be exported as WIF (5 prefix). The goal here is to signal to the wallet importing these private keys whether it must search the blockchain for compressed or uncompressed public keys and addresses.

If a bitcoin wallet is able to implement compressed public keys, then it will use those in all transactions. The private keys in the wallet will be used to derive the public key points on the curve, which will be compressed. The compressed public keys will be used to produce bitcoin addresses and those will be used in transactions. When exporting private keys from a new wallet that implements compressed public keys, the Wallet Import Format is modified, with the addition of a one-byte suffix +01+to the private key. The resulting base58check encoded private key is called a "Compressed WIF" and starts with the letter K or L, instead of starting with "5" as is the case with WIF encoded (non-compressed) keys from older wallets.

Here’s the same key, encoded in WIF and WIF-compressed formats

Table 4. Example: Same Key, Different Formats

Format	Private Key
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD_01_
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

Tip "Compressed private keys" is a misnomer! They are not compressed, rather the WIF-compressed format signifies that they should only be used to derive compressed public keys and their corresponding bitcoin addresses. Ironically, a "WIF-compressed" encoded private key is one byte longer because it has the added 01 suffix to distinguish it from an "uncompressed" one.

Implementing Keys and Addresses in Python

The most comprehensive bitcoin library in Python is "pybitcointools"by Vitalik Buterin (https://github.com/vbuterin/pybitcointools). In the following code example, we use the pybitcointools library (imported as "bitcoin") to generate and display keys and addresses in various formats:

Key and Address generation and formatting with the pybitcointools library

```

```

Here’s the output from running this code:

```
$ python key-to-address-ecc-example.py
Private Key (hex) is:  3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6
Private Key (decimal) is:  26563230048437957592232553826663696440606756685920117476832299673293013768870
Private Key (WIF) is:  5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K
Private Key Compressed (hex) is:  3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601
Private Key (WIF-Compressed) is:  KyBsPXxTuVD82av65KZkrGrWi5qLMah5SdNq6uftawDbgKa2wv6S
Public Key (x,y) coordinates is: (41637322786646325214887832269588396900663353932545912953362782457239403430124L,
16388935128781238405526710466724741593761085120864331449066658622400339362166L)
Public Key (hex) is:
045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec243bcefdd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176

Compressed Public Key (hex) is: 025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec
Bitcoin Address (b58check) is: 1thMirt546nngXqyPEz532S8fLwbozud8
Compressed Bitcoin Address (b58check) is: 14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3
```

Wallets

Wallets are containers for private keys, usually implemented as structured files or simple databases. Another method for making keys is *deterministic key generation*. Here you derive each new private key, using a one-way hash function from a previous private key, linking them in a sequence. As long as you can re-create that sequence, you only need the first key (known as a *seed* or *master* key) to generate them all. In this section we will examine the different methods of key generation and the wallet structures that are built around them.

Tip Wallets contain keys, not coins. The coins are stored on the blockchain in the form of transaction-outputs (often noted as *vout* or *txout*). Each user has a wallet containing keys. Wallets are really keychains containing pairs of private/public keys (See [public key]). Users sign transactions with the keys, thereby proving they own the transaction outputs (their coins).

Non-Deterministic (Random) Wallets

In the first implementations of bitcoin clients, wallets were simply collections of randomly generated private keys. For example, the Bitcoin Core Client pre-generates 100 random private keys when first started and generates more keys as needed, trying to use each key only once. This type of wallet is nicknamed "Just a Bunch Of Keys", or JBOK, and such wallets are being replaced with deterministic wallets because they are cumbersome to manage, backup and import. The disadvantage of random keys is that if you generate many of them you must keep copies of all of them, meaning that the wallet must be backed-up frequently. Each key must be backed-up, or the funds it controls are irrevocably lost if the wallet becomes inaccessible. This conflicts directly with the principle of avoiding address re-use, by using each bitcoin address for only one transaction. Address re-use reduces privacy by associating multiple transactions and addresses with each other. A Type-0 wallet is a poor choice of wallet, especially if you want to avoid address re-use as that means managing many keys, which creates the need for very frequent backups. The Bitcoin Core Client includes a wallet that is implemented as a Type-0 wallet, but the use of this wallet is actively discouraged by the Bitcoin Core developers.

Figure 8. Type-0 Non-Deterministic (Random) Wallet: A Collection of Randomly Generated Keys

Deterministic (Seeded)

Deterministic, or "seeded" wallets are wallets that contain private keys which are all derived from a common seed, through the use of a one-way hash function. The seed is a randomly generated number which is combined with other data, such as an index number or "chain code" (see [hd_wallets]) to derive the private keys. In a deterministic wallet, the seed is sufficient to recover all the derived keys and therefore a single backup at creation time is sufficient. The seed is also sufficient for a wallet export or import, allowing for easy migration of all the user's keys between different wallet implementations.

Mnemonic Code Words (BIP0039)

Mnemonic codes are English word sequences that are generated from a random sequence and used to produce a seed for use in deterministic wallets. The sequence of words is sufficient to re-create the seed and from there re-create the wallet and all the derived keys. A wallet application that implements deterministic wallets with mnemonic code will show the user a sequence of 12-24 words when first creating a wallet. That sequence of words is the wallet backup and can be used to recover and re-create all the keys in the same or any compatible wallet application.

The common standard for mnemonic codes is defined in Bitcoin Improvement Proposal 39 (see [bip0039]), currently in Draft status.

The standard defines the creation of a mnemonic code and seed as a follows:

1. Create a random sequence (entropy) of 128 to 256 bits
2. Create a checksum of the random sequence by taking the first few bits of its SHA256 hash
3. Add the checksum to the end of the random sequence
4. Divide the sequence into sections of 11 bits, using those to index a dictionary of 2048 pre-defined words
5. Produce 12-24 words representing the mnemonic code

Table 5. Mnemonic Codes: Entropy and Word Length

Entropy (bits)	Checksum (bits)	Entropy+Checksum	Word Length
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

The mnemonic code represents 128 to 256 bits which are used to derive a longer (512 bit) seed through the use of the key-stretching function PBKDF2. The resulting seed is used to create a deterministic wallet and all of its derived keys.

Here are some examples of mnemonic codes and the seeds they produce:

Table 6. 128-bit entropy mnemonic code and resulting seed

entropy input (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
mnemonic (12 words)	army van defense carry jealous true garbage claim echo media make crunch
seed (512 bits)	3338a6d2ee71c7f28eb5b882159634cd46a898463e9d2d0980f8e80dfbba5b0fa0291e5fb888a599b44b93187be6ee3ab5fd3ead7dd646341b2cdb8d08d13bf7

Table 7. 256-bit entropy mnemonic code and resulting seed

entropy input (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
mnemonic (24 words)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
seed (512 bits)	3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977 fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343

Deterministic Chains (Electrum Key Chains)

Figure 9. Type-1 Deterministic Wallet: A Chain of Keys Generated from a Seed

Deterministic Trees (BIP0032)

Figure 10. Type-2 Hierarchical Deterministic Wallet: A Tree of Keys Generated from a Seed

Advanced Keys and Addresses

Encrypted Private Keys (BIP0038)

Private keys must remain secret. The need for *confidentiality* of the private keys is a truism which is quite difficult to achieve in practice, as it conflicts with the equally important security objective of *availability*. Keeping the private key private is much harder when you need to store backups of the private key to avoid losing it. A private key stored in a wallet that is encrypted by a password may be secure, but that wallet needs to be backed up. At times, users need to move keys from one wallet to another — to upgrade or replace the wallet software, for example. Private key backups might also be stored on paper (see [paper_wallets]) or on external storage media, such as a USB flash drive. But what if the backup itself is stolen or lost? These conflicting security goals led to the introduction of a portable and convenient standard for encrypting private keys in a way that can be understood by many different wallets and bitcoin clients, standardized by Bitcoin Improvement Proposal 38 or BIP0038 (see [bip0038]).

BIP0038 proposes a common standard for encrypting private keys with a passphrase and encoding them with Base58Check so that they can be stored securely on backup media, transported securely between wallets or in any other conditions where the key might be exposed. The standard for encryption uses the Advanced Encryption Standard (AES), a standard established by the National Institute of Standards and Technology (NIST) and used broadly in data encryption implementations for commercial and military applications.

A BIP0038 encryption scheme takes a bitcoin private key, usually encoded in the Wallet Import Format (WIF), as a Base58Check string with a prefix of "5". Additionally, the BIP0038 encryption scheme takes a passphrase — a long password — usually composed of several words or a complex string of alphanumeric characters. The result of the

BIP0038 encryption scheme is a Base58Check encoded encrypted private key that begins with the prefix `6P`. If you see a key that starts with `6P` that means it is encrypted and requires a passphrase in order to convert (decrypt) it back into a WIF-formatted private key (prefix `5`) that can be used in any wallet. Many wallet applications now recognize BIP0038 encrypted private keys and will prompt the user for a passphrase to decrypt and import the key. Third party applications, such as the incredibly useful browser-based bitaddress.org (Wallet Details tab), can be used to decrypt BIP0038 keys.

The most common use case for BIP0038 encrypted keys is for paper wallets that can be used to backup private keys on a piece of paper. As long as the user selects a strong passphrase, a paper wallet with BIP0038 encrypted private keys is incredibly secure and a great way to create offline bitcoin storage (also known as "cold storage").

Test the following encrypted keys using bitaddress.org to see how you can get the decrypted key by entering the passphrase:

Table 8. Example of BIP0038 Encrypted Private Key

Private Key (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Passphrase	MyTestPassphrase
Encrypted Key (BIP0038)	6PRTHL6mWa48xSopbU1cKrVjpKbBZxcLRRCdctLJ3z5yxE87MobKoXdTsJ

Pay To Script Hash (P2SH) and Multi-Sig Addresses

As we know, traditional bitcoin addresses begin with the number “1” and are derived from the public key, which is derived from the private key. While anyone can send bitcoin to a “1” address, that bitcoin can only be spent by presenting the corresponding private key signature and public key hash.

Bitcoin addresses that begin with the number “3” are pay-to-script-hash (P2SH) addresses, sometimes erroneously called multi-signature or multi-sig addresses. They designate the beneficiary of a bitcoin transaction as the hash of a script, instead of the owner of a public key. The feature was introduced in January 2012 with Bitcoin Improvement Proposal 16 or BIP0016 (see [bip0016]) and is being widely adopted because it provides the opportunity to add functionality to the address itself. Unlike transactions that "send" funds to traditional “1” bitcoin addresses, also known as pay-to-public-key-hash (P2PKH), funds sent to “3” addresses require something more than the presentation of one public key hash and one private key signature as proof of ownership. The requirements are designated at the time the address is created, within the script, and all inputs to this address will be encumbered with the same requirements.

A pay-to-script-hash address is created from a transaction script, which defines who can spend a transaction output (for more detail, see [transactions]). Encoding a pay-to-script hash address involves using the same double-hash function as used during creation of a bitcoin address, only applied on the script instead of the public key.

```
script hash = RIPEMD160(SHA256(script))
```

The resulting "script hash" is encoded with Base58Check with a version prefix of 5, which results in an encoded address starting with a `3`. An example of a P2SH address is `32M8ednmuyZ2zVbes4puqe44NZumgG92sM`

Tip P2SH is not necessarily the same as a multi-signature standard transaction. A P2SH address *most often* represents a multi-signature script, but it might also represent a script encoding other types of transactions.

Multi-signature addresses and P2SH

Currently, the most common implementation of the P2SH function is the multi-signature address script. As the name implies, the underlying script requires more than one signature to prove ownership and therefore spend funds. The bitcoin multi-signature feature is designed to require M signatures (also known as the “threshold”) from a total of N keys, known as an M-of-N multi-sig, where M is equal to or less than N. For example, Bob the coffee shop owner from chapter 1 could use a multi-signature address requiring 1-of-2 signatures from a key belonging to him and a key belonging to his spouse, ensuring either of them could sign to spend a transaction output locked to this address. This would be similar to a “joint account” as implemented in traditional banking where either spouse can spend with a single signature. Or Gopesh, the web designer paid by Bob to create a website might have a 2-of-3 multi-signature address for his business that ensures that no funds can be spent unless at least two of the business partners sign a transaction.

We will explore how to create transactions that spend funds from P2SH (and multi-signature) addresses in [transactions].

Vanity Addresses

Vanity addresses are valid bitcoin addresses that contain human-readable messages, for example 1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33 is a valid address that contains the letters forming the word "Love" as the first four Base-58 letters. Vanity addresses require generating and testing billions of candidate private keys, until one derives a bitcoin address with the desired pattern. While there are some optimizations in the vanity generation algorithm, the process essentially involves picking a private key at random, deriving the public key, deriving the bitcoin address and checking to see if it matches the desired vanity pattern, repeating billions of times until a match is found.

Once a vanity address matching the desired pattern is found, the private key from which it was derived can be used by the owner to spend bitcoins in exactly the same way as any other address. Vanity addresses are no less or more secure than any other address. They depend on the same Elliptic Curve Cryptography (ECC) and Secure Hash Algorithm (SHA) as any other address. You can no easier find the private key of an address starting with a vanity pattern than you can any other address.

In our first chapter, we introduced Eugenia, a children’s charity director operating in the Philippines. Let’s say that Eugenia is organizing a bitcoin fundraising drive and wants to use a vanity bitcoin address to publicize the fundraising. Eugenia will create a vanity address that starts with "1Kids", to promote the children’s charity fundraiser. Let’s see how this vanity address will be created and what it means for the security of Eugenia’s charity.

Generating Vanity Addresses

It’s important to realize that a bitcoin address is simply a number represented by symbols in the Base-58 alphabet. The search for a pattern like "1Kids" can be seen as searching for an address in the range from "1Kids11111111111111111111111111111" to "1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz". There are approximately 58^{29} (approximately $1.4 * 10^{51}$) addresses in that range, all starting with "1Kids".

Table 9. The range of vanity addresses starting with "1Kids"

From	1Kids11111111111111111111111111111
To	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz

Let’s look at the pattern "1Kids" as a number and see how frequently we might find this pattern in a bitcoin address. An average desktop computer PC, without any specialized hardware, can search approximately 100,000 keys per second.

Table 10. The frequency of a vanity pattern (1KidsCharity) and average time-to-find on a desktop PC

Length	Pattern	Frequency	Average Search Time
1	1K	1 in 58 keys	< 1 milliseconds
2	1Ki	1 in 3,364	50 milliseconds
3	1Kid	1 in 195,000	< 2 seconds
4	1Kids	1 in 11 million	1 minute
5	1KidsC	1 in 656 million	1 hour
6	1KidsCh	1 in 38 billion	2 days
7	1KidsCha	1 in 2.2 trillion	3-4 months

8	1KidsChar	1 in 128 trillion	13-18 years
9	1KidsChari	1 in 7 quadrillion	800 years
10	1KidsCharit	1 in 400 quadrillion	46,000 years
11	1KidsCharity	1 in 23 quintillion	2.5 million years

As you can see, Eugenia won't be creating the vanity address "1KidsCharity" any time soon, even if she had access to several thousand computers. Each additional character increases the difficulty by a factor of 58. Patterns with more than seven characters are usually found by specialized hardware, such as custom-built desktops with multiple Graphical Processing Units (GPUs). These are often re-purposed bitcoin mining "rigs" that are no longer profitable for bitcoin mining but can be used effectively to find vanity addresses. Vanity searches on GPU systems are many orders of magnitude faster than on a general-purpose CPU.

Another way to find a vanity address is to outsource the work to a pool of vanity-miners, such as the pool at vanitypool.appspot.com. A pool is a service that allows those with GPU hardware to earn bitcoin searching for vanity addresses for others. For a small payment (0.01 bitcoin or approximately $5 when this was written), Eugenia can outsource the search for a 7-character pattern vanity address and get results in a few hours instead of having to run a CPU search for months.

Vanity Address Security

Vanity addresses can be used to enhance *and* to defeat security measures, they are truly a double-edged sword. Used to improve security, a distinctive address makes it harder for adversaries to substitute their own address and fool your customers into paying them instead of you. Unfortunately, vanity addresses also make it possible for anyone to create an address that *resembles* any random address, or even another vanity address, thereby fooling your customers.

Eugenia could advertise a randomly generated address (e.g. 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy) to which people can send their donations. Or, she could generate a vanity address that starts with 1Kids, to make it more distinctive.

In both cases, one of the risks of using a single fixed address (rather than a separate dynamic address per donor) is that a thief might be able to infiltrate your website and replace it with their own address, thereby diverting donations to themselves. If you have advertised your donation address in a number of different places, your users may visually inspect the address before making a payment to ensure it is the same one they saw on your website, on your email, and on your flyer. In the case of a random address like "1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy", the average user will inspect the first few characters "1J7mdg" perhaps and be satisfied that the address matches. Using a vanity address generator, someone with the intent to steal by substituting a similar-looking address can quickly generate addresses that match the first few characters:

Table 11. Generating vanity addresses to match a random address

Original Random Address	1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
Vanity (4 character match)	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
Vanity (5 character match)	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
Vanity (6 character match)	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

So does a vanity address increase security? If Eugenia generates the vanity address "1Kids33q44erFfpeXrmDSz7zEqG2FesZEN", users are likely to look at the vanity pattern word *and a few characters beyond*, for example noticing the "1Kids33" part of the address. That would force an attacker to generate a vanity address matching at least 6 characters (2 more), expending an effort that is 3,364 times (58 x 58) higher than the effort Eugenia expended for her 4 character vanity. Essentially, the effort Eugenia expends (or pays a vanity pool for) "pushes" the attacker into having to produce a longer pattern vanity. If Eugenia pays a pool to generate an 8 character vanity address, the attacker would be pushed into the realm of 10 characters which is infeasible on a personal computer and expensive even with a custom vanity-mining rig or vanity pool. What is affordable for Eugenia becomes unaffordable for the attacker, especially if the potential reward of fraud is not high enough to cover the cost of the vanity address generation.

Paper Wallets

Paper wallets are bitcoin private keys printed on paper. Often the paper wallet also includes the corresponding bitcoin address, for convenience, but this is not necessary since it can be derived from the private key. Paper wallets are a very effective way to create backups or offline bitcoin storage, also known as "cold storage". As a backup mechanism, a paper wallet can provide security against the loss of key due to a computer mishap such as a hard drive failure, theft, or accidental deletion. As a "cold storage" mechanism, if the paper wallet keys are generated offline and never stored on a computer system, they are much more secure against hackers, key-loggers and other online computer threats.

Paper wallets come in many shapes, sizes and designs, but at a very basic level are just a key and an address printed on paper. Here’s the simplest form of a paper wallet:

Table 12. A very simple paper wallet - a printout of the bitcoin address and private key

Public Address	1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x
Private Key (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

Paper wallets can be generated easily using a tool such as the client-side Javascript generator at bitaddress.org. This page contains all the code necessary to generate keys and paper wallets, even while completely disconnected from the Internet. To use it, save the HTML page on your local drive or on an external USB flash drive. Disconnect from the Internet and open the file in a browser. Even better, boot your computer using a pristine operating system, such as a CDROM bootable Linux OS. Any keys generated with this tool while offline can be printed on a local printer over a USB cable (not wirelessly), thereby creating paper wallets whose keys exist only on the paper and have never been stored on any online system. Put these paper wallets in a fire-proof safe and "send" bitcoin to their bitcoin address, to implement a simple yet highly effective "cold storage" solution.

Figure 11. An example of a simple paper wallet from bitaddress.org

The disadvantage of the simple paper wallet system is that the printed keys are vulnerable to theft. A thief who is able to gain access to the paper can either steal it or photograph the keys and take control of the bitcoins locked with those keys. A more sophisticated paper wallet storage system uses BIP0038 encrypted private keys. The keys printed on the paper wallet are protected by a passphrase that the owner has memorized. Without the passphrase, the encrypted keys are useless. Yet, they still are superior to a passphrase protected wallet because the keys have never been online and must be physically retrieved from a safe or other physically secured storage.

Figure 12. An example of an encrypted paper wallet from bitaddress.org. The passphrase is "test"

Warning

While you can deposit funds into a paper wallet several times, you should withdraw all funds only once, spending everything. This is because in the process of unlocking and spending funds you expose the private key and because some wallets may generate a change address if you spend less than the whole amount. One way to do this is to withdraw the entire balance stored in the paper wallet and send any remaining funds to a new paper wallet.

Paper wallets come in many designs and sizes, with many different features. Some are intended to be given as gifts and have seasonal themes, such as Christmas and New Year’s themes. Others are designed for storage in a bank vault or safe with the private key hidden in some way, either with opaque scratch-off stickers, or folded and sealed with tamper-proof adhesive foil.

Figure 13. An example of a paper wallet from bitcoinpaperwallet.com with the private key on a folding flap.

Figure 14. The bitcoinpaperwallet.com paper wallet with the private key concealed.

Other designs feature additional copies of the key and address, in the form of detachable stubs similar to ticket stubs, allowing you to store multiple copies to protect against fire, flood or other natural disasters.

Figure 15. An example of a paper wallet with additional copies of the keys on a backup "stub"

Last updated 2014-06-25 19:31:48 CDT