Chapter 4. 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 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.

Every bitcoin transaction requires a valid signature to be included in the blockchain, which can only be generated with valid digital keys; therefore, anyone with a copy of those keys has control of the bitcoin in that account. Keys come in pairs consisting of a private (secret) key and a public key. Think of the public key as similar to a bank account number and the private key as similar to the secret PIN, or signature on a check 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.

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 checks: a single payment instrument that can be used to pay into people's accounts, pay into company accounts, pay for bills, or pay to cash. The bitcoin address is the only representation of the keys that users will routinely see, because 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.

Public Key Cryptography and Cryptocurrency

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 her public key and a signature (different each time, but created from the same private key) in a transaction to spend those bitcoins. Through

> **TIP**
>
> In most wallet implementations, the private and public keys are stored together as a *key pair* for convenience. However, the public key can be calculated from 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 in Figure 4-1.

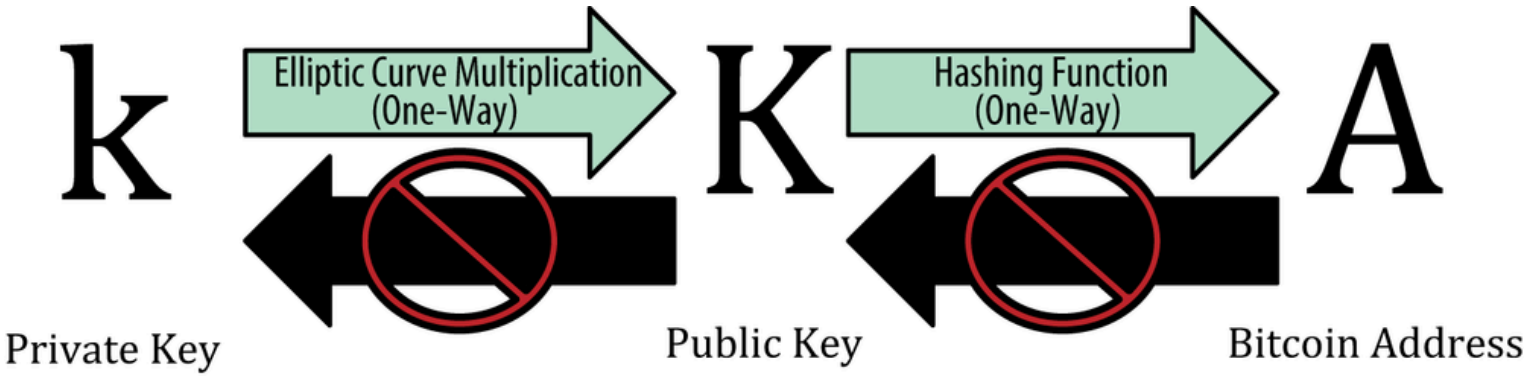

Figure 4-1. Private key, public key, and bitcoin address

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, because revealing it to third

> **TIP**
>
> The bitcoin private key is just a number. You can pick your private 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. The public key can then be generated from the private key.

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}, slightly less than 2^{256}) defined as the order of the elliptic curve used in bitcoin (see Elliptic Curve Cryptography Explained). 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 SHA256 hash algorithm that 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 write your own code to create a random number or use a "simple" random number generator offered by your programming language. Use a cryptographically secure pseudo-random number generator (CSPRNG) with a seed from a source of sufficient entropy. Study the documentation of the random number generator library you choose to make sure it is cryptographically secure. Correct implementation of the CSPRNG is critical to the security of the keys.

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```

> **TIP**
>
> The size of bitcoin's private key space, 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 Chapter 3), 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` command shows the private key in a Base58 checksum-encoded format called the *Wallet Import Format* (WIF), which we will examine in more detail in Private key 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.

> **TIP**
>
> The `dumpprivkey` command is not generating a private key from a public key, as this is impossible. The command simply reveals the private key that is already known to the wallet and which was generated by the getnewaddress command.

You can also use the Bitcoin Explorer command-line tool (see Libbitcoin and Bitcoin Explorer) to generate and display private keys with the commands `seed`, `ec-new` and `ec-to-wif`:

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, known as "finding the discrete logarithm"—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.

Figure 4-2 is an example of an elliptic curve, similar to that used by bitcoin.

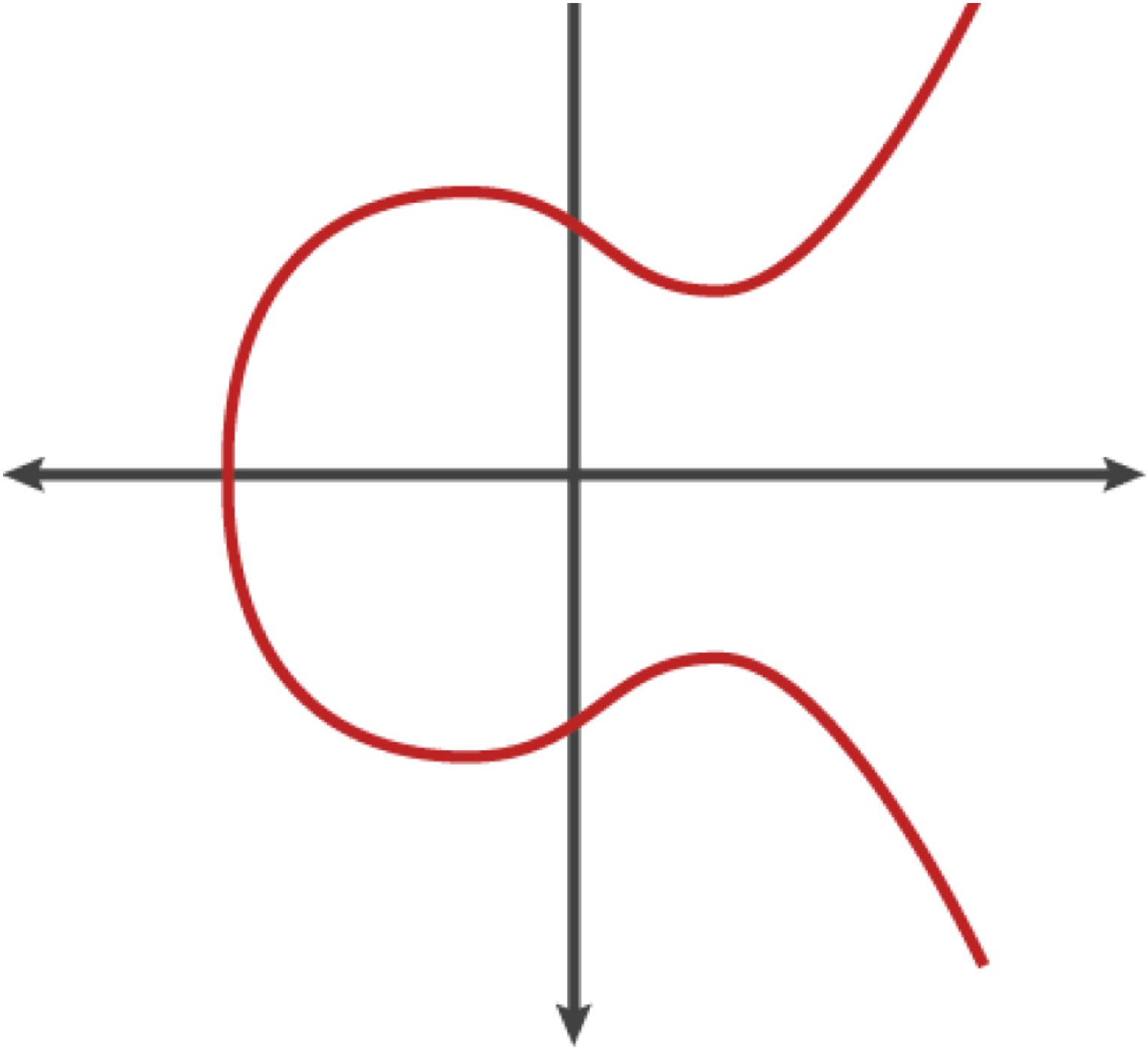

Figure 4-2. An elliptic curve

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:

or

$$y^2 \bmod p = (x^3 + 7) \bmod p$$

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. As an example, Figure 4-3 shows 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.

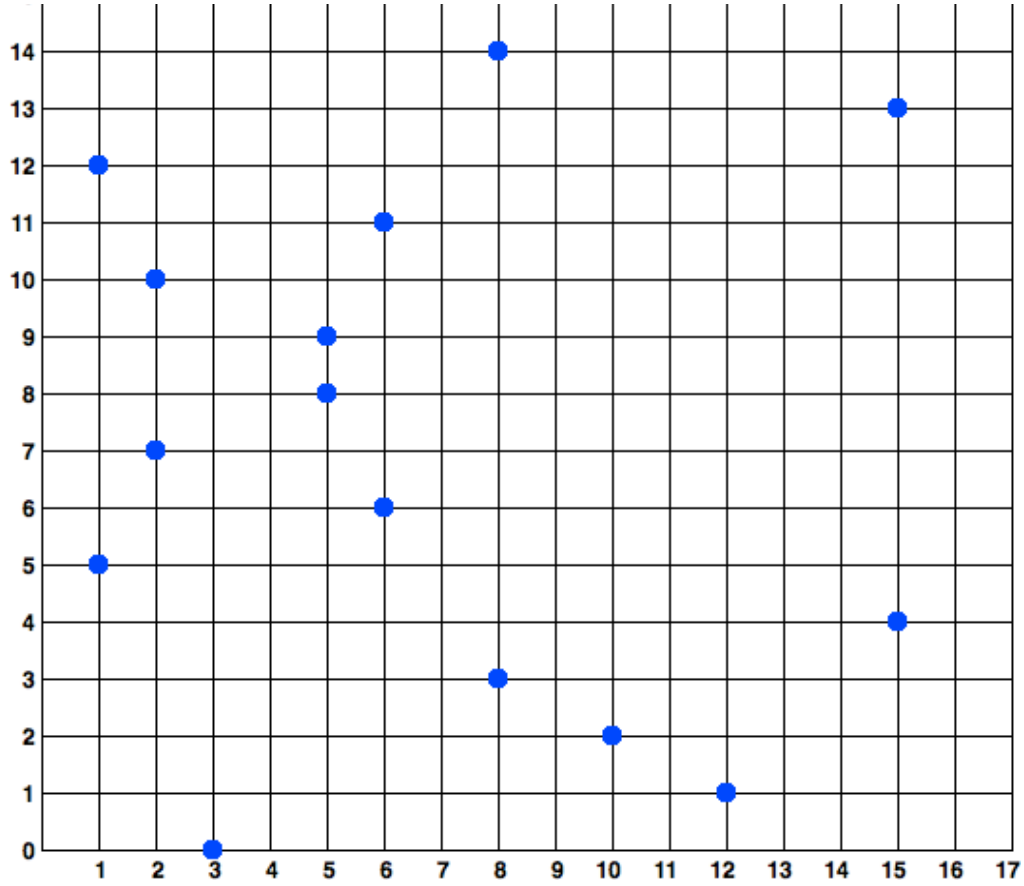

Figure 4-3. Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17

So, for example, the following is a point P with coordinates (x,y) that is a point on the `secp256k1` curve. You can check this yourself using Python:

```
P = (55066263022277343669578718895168534326250603453777594175500187360389116729240
```

```
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7 - y**2) % p
0
```

There is also a + operator, called "addition," which has some properties similar to the traditional addition of real numbers that grade school children learn. Given two points P_1 and P_2 on the elliptic curve, there is a third point $P_3 = P_1 + P_2$, also on the elliptic curve.

Geometrically, this third point P_3 is calculated by drawing a line between P_1 and P_2. This line will intersect the elliptic curve in exactly one additional place. Call this point $P_3' = (x, y)$. Then reflect in the x-axis to get $P_3 = (x, –y)$.

There are a couple of special cases that explain the need for the "point at infinity."

If P_1 and P_2 are the same point, the line "between" P_1 and P_2 should extend to be the tangent on the curve at this point P_1. This tangent will intersect the curve in exactly one new point. You can use techniques from calculus to determine the slope of the tangent line. These techniques curiously work, even though we are restricting our interest to points on the curve with two integer coordinates!

In some cases (i.e., if P_1 and P_2 have the same x values but different y values), the tangent line will be exactly vertical, in which case P3 = "point at infinity."

If P_1 is the "point at infinity," then the sum $P_1 + P_2 = P_2$. Similary, if P_2 is the point at infinity, then $P_1 + P_2 = P_1$. This shows how the point at infinity plays the role of 0.

It turns out that + is associative, which means that (A + B) + C = A + (B + C). That means we can write A + B + C without parentheses without any ambiguity.

Now that we have defined addition, we can define multiplication in the standard way that extends addition. For a point P on the elliptic curve, if k is a whole number, then kP = P + P + P + … + P (k times). Note that k is sometimes confusingly called an "exponent" in this case.

termined 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:

$$K = k * G$$

where k is the private key, G is the generator point, and K is the resulting public key, a point on the curve. Because the generator point is always the same for all bitcoin users, a private key k multiplied with G will always result in the same public key K. The relationship between k and K is fixed, but can only be calculated in one direction, from k to K. That's why a bitcoin address (derived from K) can be shared with anyone and does not reveal the user's private key (k).

> **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.

Implementing the elliptic curve multiplication, we take the private key k generated previously and multiply it with the generator point G to find the public key K:

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

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

```
K = (x, y)

where,

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

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.

Figure 4-4 shows the process for deriving G, 2G, 4G, as a geometric operation on the curve.

TIP

Most bitcoin implementations use the OpenSSL cryptographic library to do the elliptic curve math. For example, to derive the public key, the function `EC_POINT_mul()` is used.

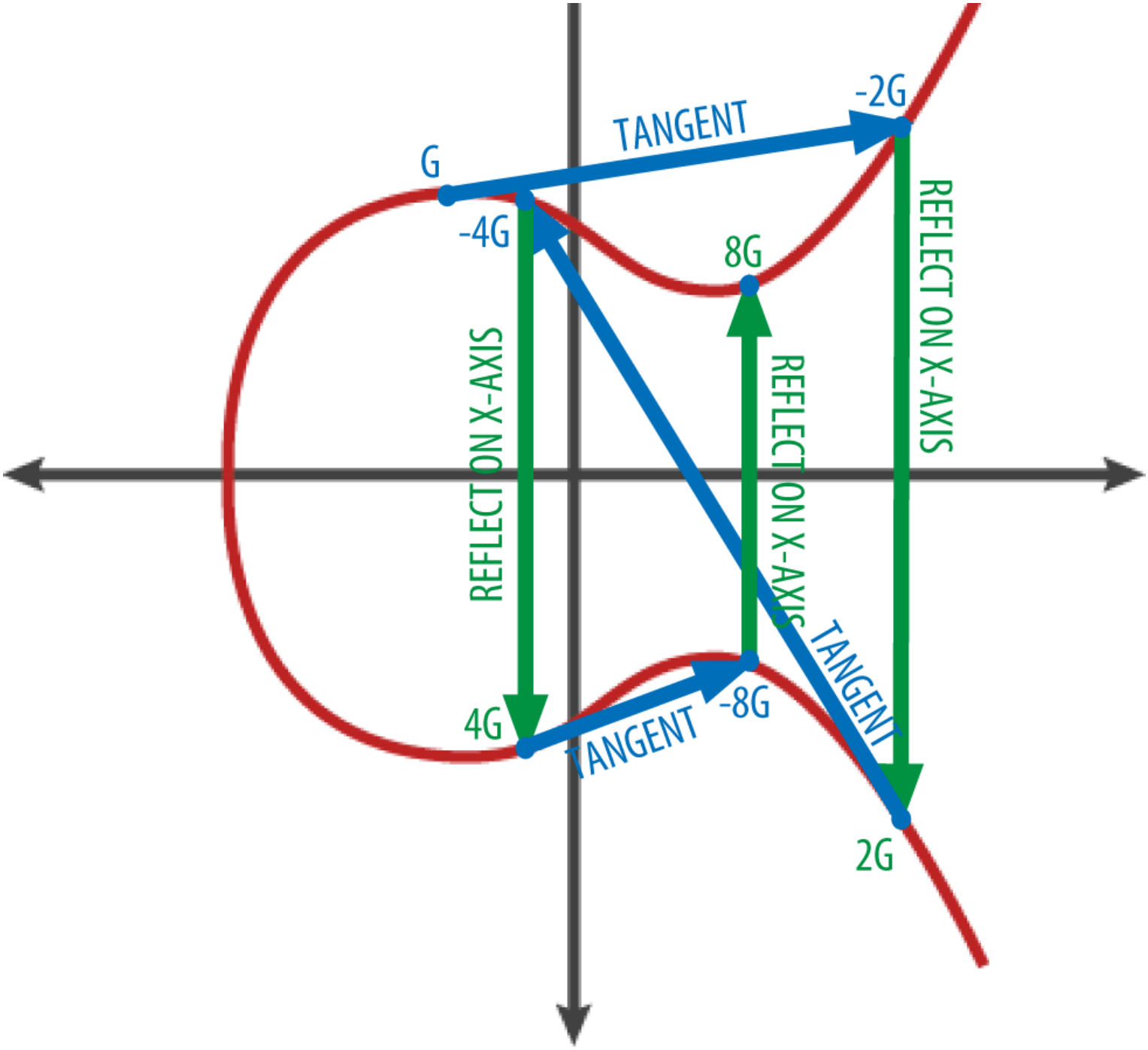

Figure 4-4. Elliptic curve cryptography: Visualizing the multiplication of a point G by an integer k on an elliptic curve

Bitcoin Addresses

A bitcoin address is a string of digits and characters that can be shared with anyone who wants to send you money. Addresses produced from public keys consist of a string of numbers and letters, beginning with the digit "1". Here's an example of a bitcoin address:

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 check, the bitcoin address is the beneficiary, which is what we write on the line after "Pay to the order of." On a paper check, that beneficiary can sometimes be the name of a bank account holder, but can also include corporations, institutions, or even cash. Because paper checks do not need to specify an account, but rather use an abstract name as the recipient of funds, that makes paper checks 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 Pay-to-Script-Hash (P2SH). For now, let's examine the simple case, a bitcoin address that represents, and is derived from, a public key.

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, in 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:

$$A = RIPEMD160(SHA256(K))$$

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

TIP

A bitcoin address is *not* the same as a public key. Bitcoin addresses are derived from a public key using a one-way function.

Bitcoin addresses are almost always presented to users in an encoding called "Base58Check" (see Base58 and Base58Check Encoding), which uses 58 characters (a Base58 number system) and a

crypted key, or a script hash. In the next section we will examine the mechanics of Base58Check encoding and decoding, and the resulting representations. Figure 4-5 illustrates the conversion of a public key into a bitcoin address.

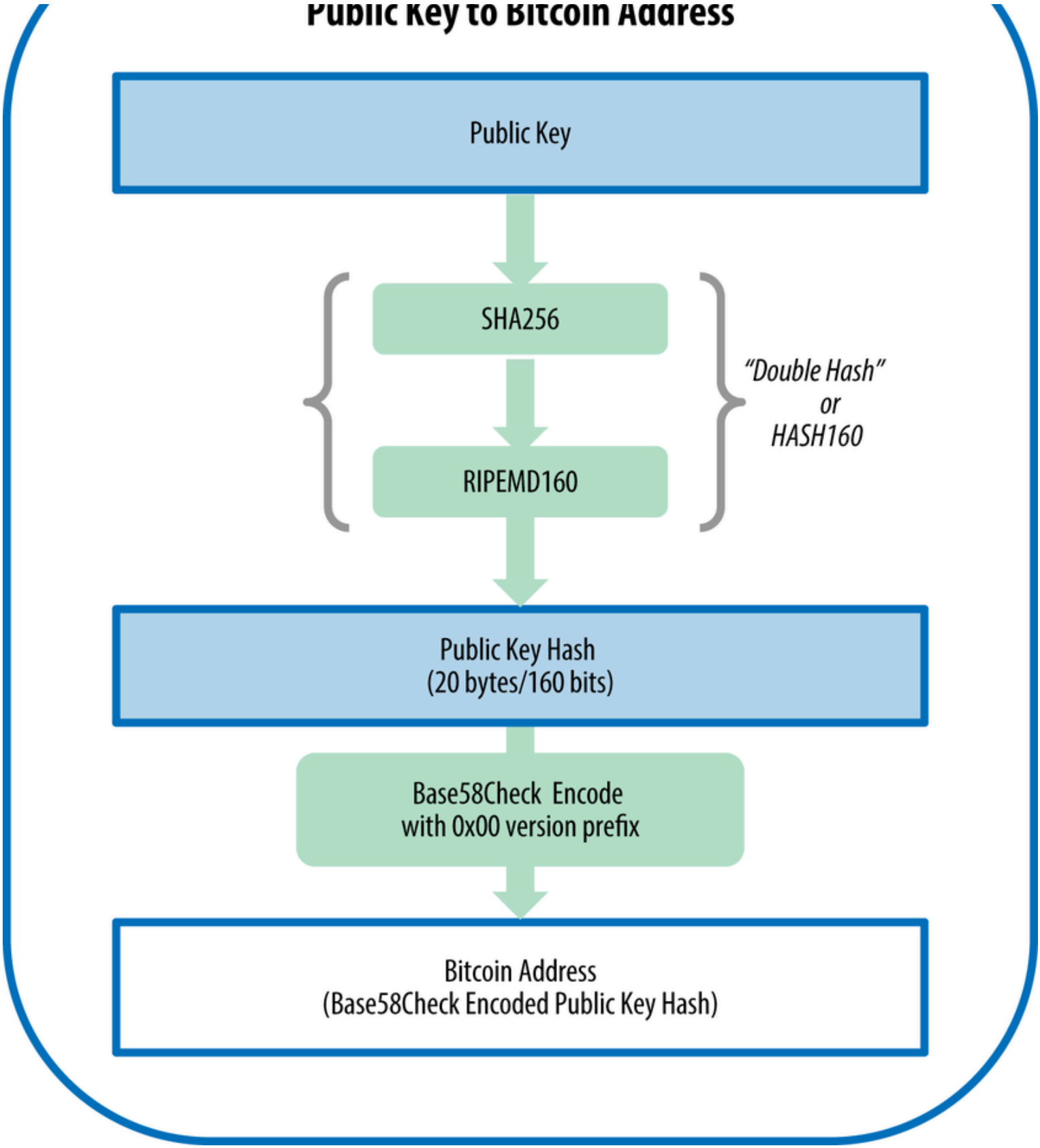

Figure 4-5. Public key to bitcoin address: conversion of a public key into a bitcoin address

use mixed-alphanumeric representations with a base (or radix) higher than 10. For example, whereas the traditional decimal system uses the 10 numerals 0 through 9, the hexadecimal system uses 16, 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. Base58 is a text-based binary-encoding format developed for use in bitcoin and used in many other cryptocurrencies. It offers a balance between compact representation, readability, and error detection and prevention. Base58 is a subset of Base64, using the upper- and lowercase letters and numbers, but omitting some characters that are frequently mistaken for one another and can appear identical when displayed in certain fonts. Specifically, Base58 is Base64 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) just mentioned.

Example 4-1. bitcoin's Base58 alphabet

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

To add extra security against typos or transcription errors, Base58Check is a Base58 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 that 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 in Table 4-1.

```
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 is composed of three items: a prefix, the data, and a checksum. This result is encoded using the Base58 alphabet described previously. Figure 4-6 illustrates the Base58Check encoding process.

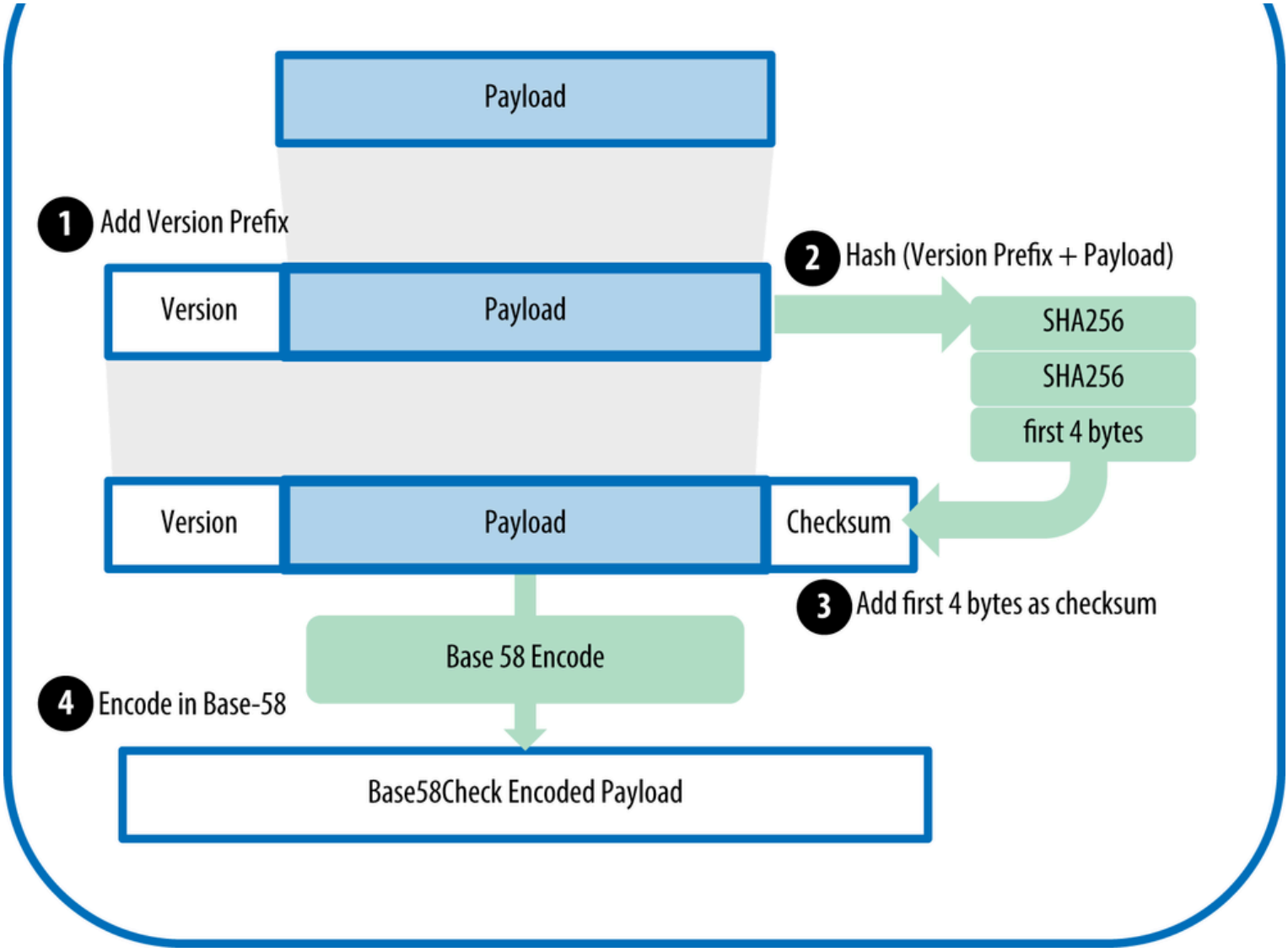

Figure 4-6. Base58Check encoding: a Base58, versioned, and checksummed format for unambiguously encoding bitcoin data

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 Base58 contain specific characters at the beginning of the Base58Check-encoded payload. These characters make 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 Base58 characters are shown in Table 4-1.

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

Let's look at the complete process of creating a bitcoin address, from a private key, to a public key (a point on the elliptic curve), to a double-hashed address and finally, the Base58Check encoding. The C++ code in Example 4-2 shows the complete step-by-step process, from private key to Base58Check-encoded bitcoin address. The code example uses the libbitcoin library introduced in Alternative Clients, Libraries, and Toolkits for some helper functions.

Example 4-2. Creating a Base58Check-encoded bitcoin address from a private key

```
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Private secret key.
    bc::ec_secret secret;
    bool success = bc::decode_base16(secret,
        "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");
    assert(success);
    // Get public key.
    bc::ec_point public_key = bc::secret_to_public_key(secret);
```

```
    // Normally you can use:
    //   bc::payment_address payaddr;
    //   bc::set_public_key(payaddr, public_key);
    //   const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    const bc::short_hash hash = bc::bitcoin_short_hash(public_key);

    bc::data_chunk unencoded_address;
    // Reserve 25 bytes
    //   [ version:1  ]
    //   [ hash:20    ]
    //   [ checksum:4 ]
    unencoded_address.reserve(25);
    // Version byte, 0 is normal BTC address (P2PKH).
    unencoded_address.push_back(0);
    // Hash data
    bc::extend_data(unencoded_address, hash);
    // Checksum is computed by hashing data, and adding 4 bytes from hash.
    bc::append_checksum(unencoded_address);
    // Finally we must encode the result in Bitcoin's base58 encoding
    assert(unencoded_address.size() == 25);
    const std::string address = bc::encode_base58(unencoded_address);

    std::cout << "Address: " << address << std::endl;
    return 0;
}
```

The code uses a predefined private key so that it produces the same bitcoin address every time it is run, as shown in Example 4-3.

Example 4-3. Compiling and running the addr code

```
# Compile the addr.cpp code
$ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)
# Run the addr executable
$ ./addr
Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK
```

tions all encode the same number, even though they look different. These formats are primarily used to make it easy for people to read and transcribe keys without introducing errors.

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. Table 4-2 shows three common formats used to represent private keys.

Table 4-2. Private key representations (encoding formats)

Type	Prefix	Description
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base58 with version prefix of 128 and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

Table 4-3 shows the private key generated in these three formats.

Table 4-3. Example: Same key, different formats

Format	Private Key
Hex	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

We use the `wif-to-ec` command from Bitcoin Explorer (see Libbitcoin and Bitcoin Explorer) to show that both WIF keys represent the same private key:

```
$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

$ bx wif-to-ec KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
```

Decode from Base58Check

The Bitcoin Explorer commands (see Libbitcoin and Bitcoin Explorer) make it easy to write shell scripts and command-line "pipes" that manipulate bitcoin keys, addresses, and transactions. You can use Bitcoin Explorer to decode the Base58Check format on the command line.

We use the `base58check-decode` command to decode the uncompressed key:

```
$ bx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
wrapper
{
    checksum 4286807748
    payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
    version 128
}
```

The result contains the key as payload, the Wallet Import Format (WIF) version prefix 128, and a checksum.

Notice that the "payload" of the compressed key is appended with the suffix `01`, signalling that the derived public key is to be compressed.

```
$ bx base58check-decode KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
wrapper
{
    checksum 2339607926
```

Encode from hex to Base58Check

To encode into Base58Check (the opposite of the previous command), we use the `base58check-encode` command from Bitcoin Explorer (see Libbitcoin and Bitcoin Explorer) and provide the hex private key, followed by the Wallet Import Format (WIF) version prefix 128:

```
bx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a52
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Encode from hex (compressed key) to Base58Check

To encode into Base58Check as a "compressed" private key (see Compressed private keys), we append the suffix `01` to the hex key and then encode as above:

```
$ bx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a
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 Compressed public keys).

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`.

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

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`:

```
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A<?pdf-cr?>0
```

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 Public Keys, a public key is a point (x,y) on an elliptic curve. Because 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: because 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 in Figure 4-2, 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 because 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

from the *x* coordinate and uncompress the public key to the full coordinates of the point. Public key compression is illustrated in Figure 4-7.

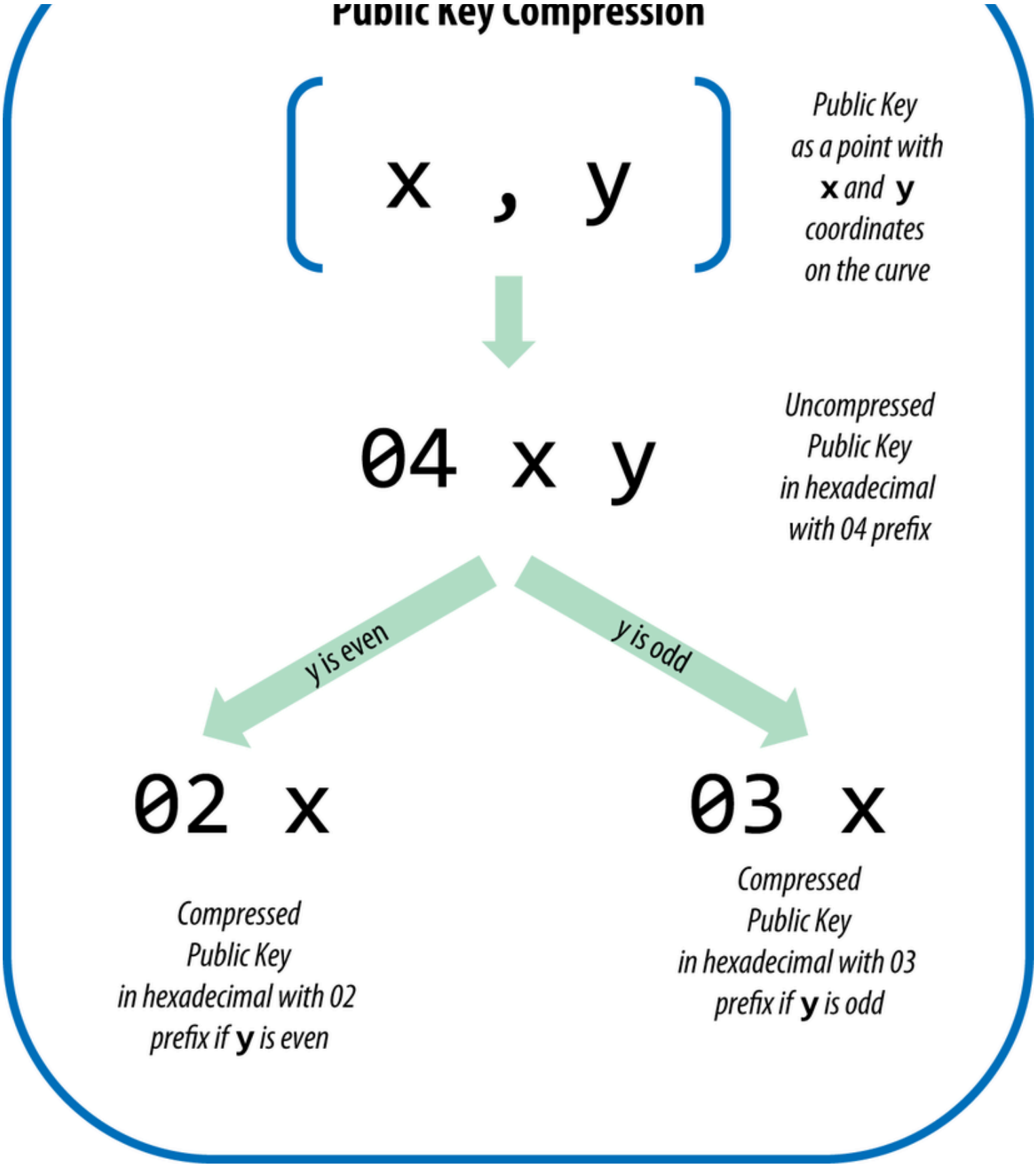

Figure 4-7. Public key compression

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
```

This compressed public key 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) that 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 that 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 term "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 be-

lic 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 (with a K or 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 (with a 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, 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.

Table 4-4 shows the same key, encoded in WIF and WIF-compressed formats.

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. In Example 4-4, we use the pybitcointools library (imported as "bitcoin") to generate and display keys and addresses in various formats.

Example 4-4. Key and address generation and formatting with the pybitcointools library

```
import bitcoin

# Generate a random private key
valid_private_key = False
```

```
    valid_private_key = 0 < decoded_private_key < bitcoin.N

print "Private Key (hex) is: ", private_key
print "Private Key (decimal) is: ", decoded_private_key

# Convert private key to WIF format
wif_encoded_private_key = bitcoin.encode_privkey(decoded_private_key, 'wif')
print "Private Key (WIF) is: ", wif_encoded_private_key

# Add suffix "01" to indicate a compressed private key
compressed_private_key = private_key + '01'
print "Private Key Compressed (hex) is: ", compressed_private_key

# Generate a WIF format from the compressed private key (WIF-compressed)
wif_compressed_private_key = bitcoin.encode_privkey(
    bitcoin.decode_privkey(compressed_private_key, 'hex'), 'wif')
print "Private Key (WIF-Compressed) is: ", wif_compressed_private_key

# Multiply the EC generator point G with the private key to get a public key point
public_key = bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
print "Public Key (x,y) coordinates is:", public_key

# Encode as hex, prefix 04
hex_encoded_public_key = bitcoin.encode_pubkey(public_key,'hex')
print "Public Key (hex) is:", hex_encoded_public_key

# Compress public key, adjust prefix depending on whether y is even or odd
(public_key_x, public_key_y) = public_key
if (public_key_y % 2) == 0:
    compressed_prefix = '02'
else:
    compressed_prefix = '03'
hex_compressed_public_key = compressed_prefix + bitcoin.encode(public_key_x, 16)
print "Compressed Public Key (hex) is:", hex_compressed_public_key

# Generate bitcoin address from public key
print "Bitcoin Address (b58check) is:", bitcoin.pubkey_to_address(public_key)

# Generate compressed bitcoin address from compressed public key
```

Example 4-5 shows the output from running this code.

Example 4-5. Running key-to-address-ecc-example.py

```
$ 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:
 045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec243bcefdd43470
Compressed Public Key (hex) is:
 025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec
Bitcoin Address (b58check) is:
 1thMirt546nngXqyPEz532S8fLwbozud8
Compressed Bitcoin Address (b58check) is:
 14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3
```

Example 4-6 is another example, using the Python ECDSA library for the elliptic curve math and without using any specialized bitcoin libraries.

Example 4-6. A script demonstrating elliptic curve math used for bitcoin keys

```
import ecdsa
import os
from ecdsa.util import string_to_number, number_to_string

# secp256k1, http://www.oid-info.com/get/1.3.132.0.10
_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2FL
```

```
_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L
_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L
curve_secp256k1 = ecdsa.ellipticcurve.CurveFp(_p, _a, _b)
generator_secp256k1 = ecdsa.ellipticcurve.Point(curve_secp256k1, _Gx, _Gy, _r)
oid_secp256k1 = (1, 3, 132, 0, 10)
SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve_secp256k1, generator_secp256k1, oid_se
ec_order = _r

curve = curve_secp256k1
generator = generator_secp256k1

def random_secret():
    convert_to_int = lambda array: int("".join(array).encode("hex"), 16)

    # Collect 256 bits of random data from the OS's cryptographically secure random gene
    byte_array = os.urandom(32)

    return convert_to_int(byte_array)

def get_point_pubkey(point):
    if point.y() & 1:
        key = '03' + '%064x' % point.x()
    else:
        key = '02' + '%064x' % point.x()
    return key.decode('hex')

def get_point_pubkey_uncompressed(point):
    key = '04' + \
          '%064x' % point.x() + \
          '%064x' % point.y()
    return key.decode('hex')


# Generate a new private key.
secret = random_secret()
print "Secret: ", secret

# Get the public key point.
point = secret * generator
print "EC point:", point
```

```
point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec_order)
assert point1 == point
```

Example 4-7 shows the output produced by running this script.

> **NOTE**
>
> The example above uses `os.urandom`, which reflects a cryptographically secure random number generator (CSRNG) provided by the underlying operating system. In the case of an UNIX-like operating system such as Linux, it draws from `/dev/urandom`; and in the case of Windows, calls `CryptGenRandom()`. If a suitable randomness source is not found, `NotImplementedError` will be raised. While the random number generator used here is for demonstration purposes, it is *not* appropriate for generating production-quality bitcoin keys as it is not implemented with sufficient security.

Example 4-7. Installing the Python ECDSA library and running the ec_math.py script

```
$ # Install Python PIP package manager
$ sudo apt-get install python-pip
$ # Install the Python ECDSA library
$ sudo pip install ecdsa
$ # Run the script
$ python ec-math.py
Secret:  38090835015954358862481132628887443905906204995912378278060168703580660 2
EC point: (700488535318671794898577504976069662723825834713229354546245955400072 6
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a152287
```

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.

pairs of private/public keys (see Private and Public Keys). Users sign transactions with the keys, thereby proving they own the transaction outputs (their coins). The coins are stored on the blockchain in the form of transaction-ouputs (often noted as vout or txout).

Nondeterministic (Random) Wallets

In the first bitcoin clients, wallets were simply collections of randomly generated private keys. This type of wallet is called a *Type-0 nondeterministic wallet*. For example, the Bitcoin Core client pre-generates 100 random private keys when first started and generates more keys as needed, using 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, back up, 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 nondeterministic wallet is a poor choice of wallet, especially if you want to avoid address re-use because that means managing many keys, which creates the need for frequent backups. Although the Bitcoin Core client includes a Type-0 wallet, using this wallet is discouraged by developers of Bitcoin Core. Figure 4-8 shows a nondeterministic wallet, containing a loose collection of random keys.

Deterministic (Seeded) Wallets

Deterministic, or "seeded" wallets are wallets that contain private keys that are all derived from a common seed, through the use of a one-way hash function. The seed is a randomly generated number that is combined with other data, such as an index number or "chain code" (see Hierarchical Deterministic Wallets (BIP0032/BIP0044)) 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.

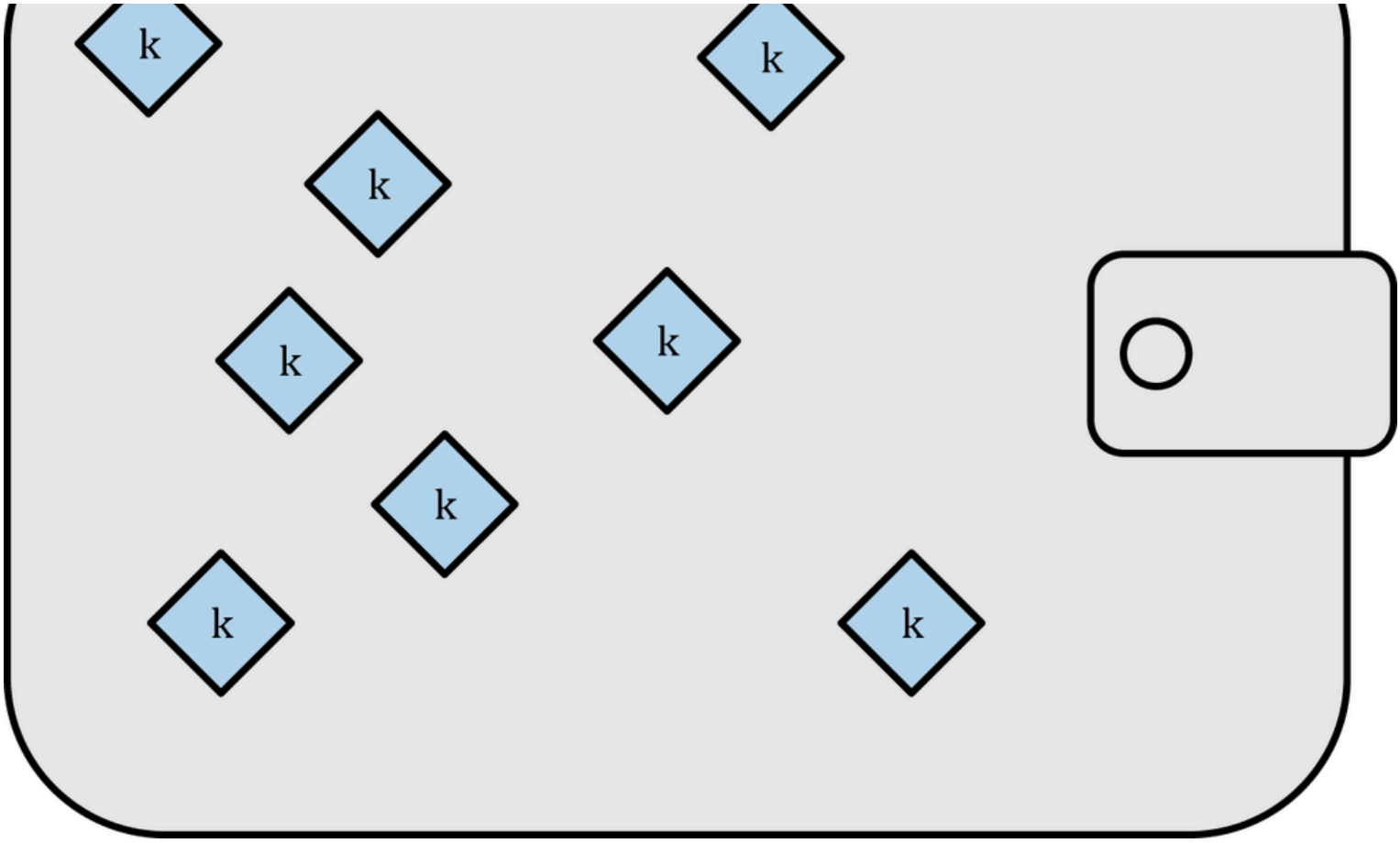

Figure 4-8. Type-0 nondeterministic (random) wallet: a collection of randomly generated keys

Mnemonic Code Words

Mnemonic codes are English word sequences that represent (encode) a random number used as a seed to derive a deterministic wallet. 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 to 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. Mnemonic code words make it easier for users to back up wallets because they are easy to read and correctly transcribe, as compared to a random sequence of numbers.

Mnemonic codes are defined in Bitcoin Improvement Proposal 39 (see [bip0039]), currently in Draft status. Note that BIP0039 is a draft proposal and not a standard. Specifically, there is a different standard, with a different set of words, used by the Electrum wallet and predating BIP0039. BIP0039

BIP0039 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 predefined words.
5. Produce 12 to 24 words representing the mnemonic code.

Table 4-5 shows the relationship between the size of entropy data and the length of mnemonic codes in words.

Table 4-5. 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

Tables 4-6 and 4-7 show some examples of mnemonic codes and the seeds they produce.

Table 4-6. 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 4-7. 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)	3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343

most advanced form of deterministic wallets is the *hierarchical deterministic wallet* or *HD wallet* defined by the BIP0032 standard. Hierarchical deterministic wallets contain keys derived in a tree structure, such that a parent key can derive a sequence of children keys, each of which can derive a sequence of grandchildren keys, and so on, to an infinite depth. This tree structure is illustrated in Figure 4-9.

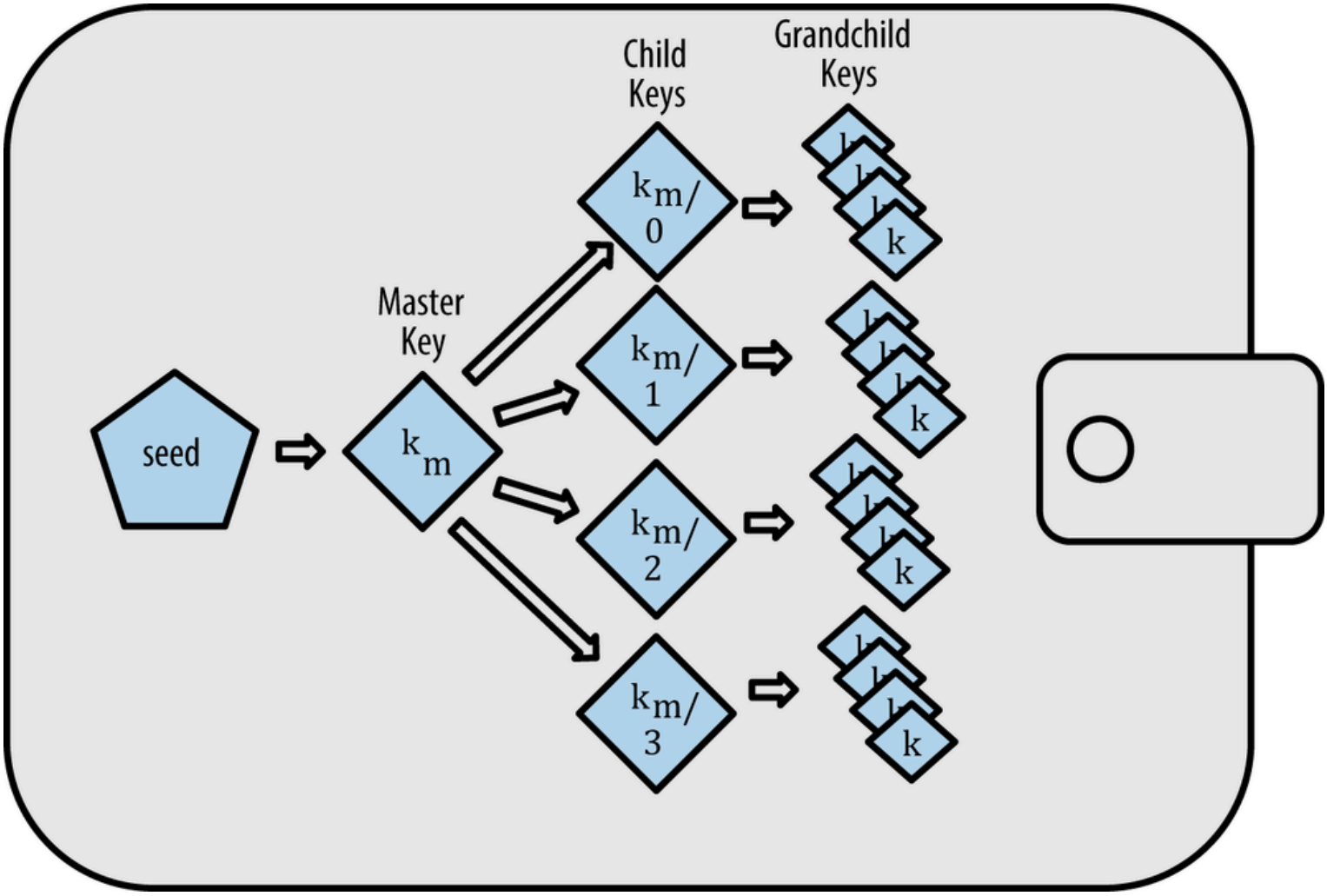

Figure 4-9. Type-2 hierarchical deterministic wallet: a tree of keys generated from a single seed

TIP

If you are implementing a bitcoin wallet, it should be built as an HD wallet following the BIP0032 and BIP0044 standards.

payments. Branches of keys can also be used in a corporate setting, allocating different branches to departments, subsidiaries, specific functions, or accounting categories.

The second advantage of HD wallets is that users can create a sequence of public keys without having access to the corresponding private keys. This allows HD wallets to be used on an insecure server or in a receive-only capacity, issuing a different public key for each transaction. The public keys do not need to be preloaded or derived in advance, yet the server doesn't have the private keys that can spend the funds.

HD wallet creation from a seed

HD wallets are created from a single *root seed*, which is a 128-, 256-, or 512-bit random number. Everything else in the HD wallet is deterministically derived from this root seed, which makes it possible to re-create the entire HD wallet from that seed in any compatible HD wallet. This makes it easy to back up, restore, export, and import HD wallets containing thousands or even millions of keys by simply transferring only the root seed. The root seed is most often represented by a *mnemonic word sequence*, as described in the previous section Mnemonic Code Words, to make it easier for people to transcribe and store it.

The process of creating the master keys and master chain code for an HD wallet is shown in Figure 4-10.

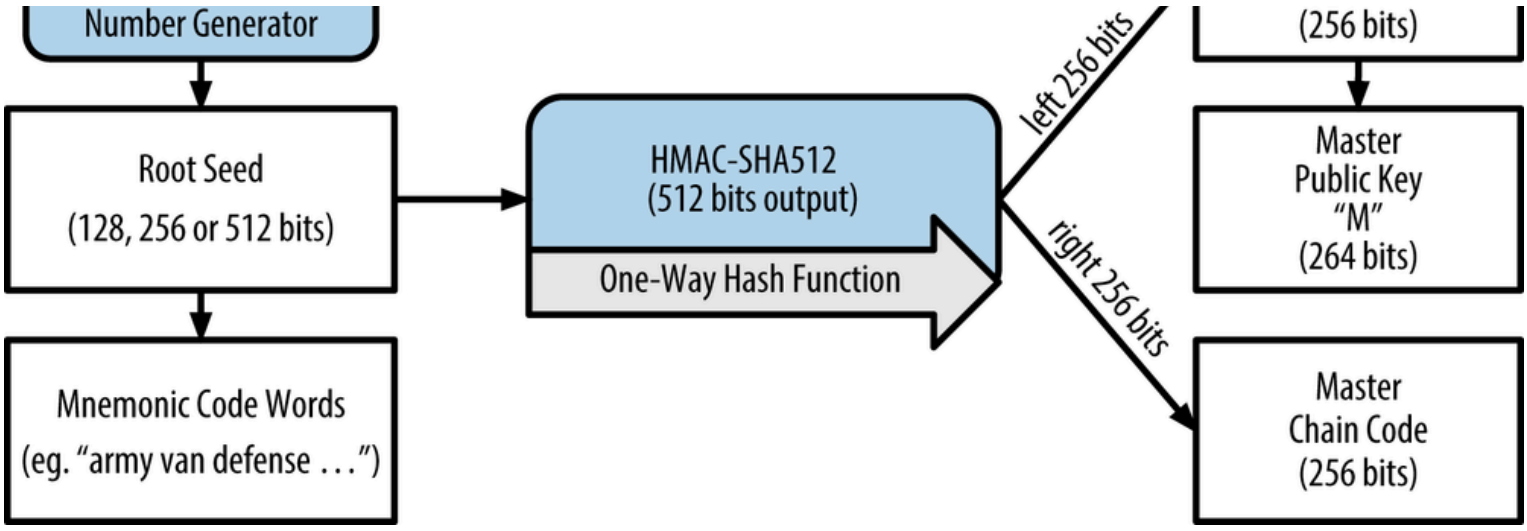

Figure 4-10. Creating master keys and chain code from a root seed

The root seed is input into the HMAC-SHA512 algorithm and the resulting hash is used to create a *master private key* (m) and a *master chain code*. The master private key (m) then generates a corresponding master public key (M), using the normal elliptic curve multiplication process `m * G` that we saw earlier in this chapter. The chain code is used to introduce entropy in the function that creates child keys from parent keys, as we will see in the next section.

Private child key derivation

Hierarchical deterministic wallets use a *child key derivation* (CKD) function to derive children keys from parent keys.

The child key derivation functions are based on a one-way hash function that combines:

- A parent private or public key (ECDSA uncompressed key)
- A seed called a chain code (256 bits)
- An index number (32 bits)

The chain code is used to introduce seemingly random data to the process, so that the index is not sufficient to derive other child keys. Thus, having a child key does not make it possible to find its siblings, unless you also have the chain code. The initial chain code seed (at the root of the tree) is made from random data, while subsequent chain codes are derived from each parent chain code.

SHA512 algorithm to produce a 512-bit hash. The resulting hash is split into two halves. The right-half 256 bits of the hash output become the chain code for the child. The left-half 256 bits of the hash and the index number are added to the parent private key to produce the child private key. In Figure 4-11, we see this illustrated with the index set to 0 to produce the 0'th (first by index) child of the parent.

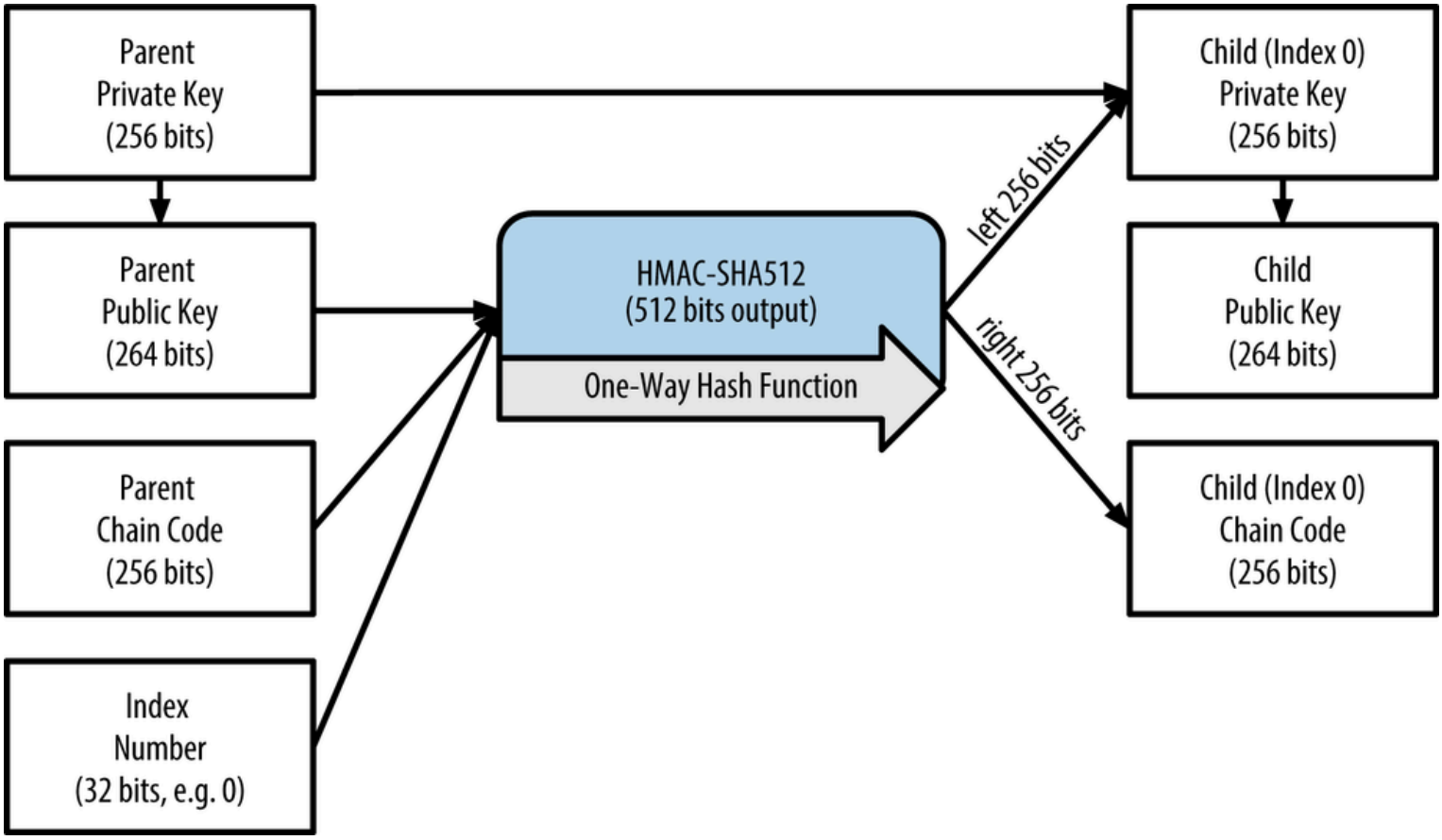

Figure 4-11. Extending a parent private key to create a child private key

Changing the index allows us to extend the parent and create the other children in the sequence, e.g., Child 0, Child 1, Child 2, etc. Each parent key can have 2 billion children keys.

Repeating the process one level down the tree, each child can in turn become a parent and create its own children, in an infinite number of generations.

tion function is a one-way function, the child key cannot be used to find the parent key. The child key also cannot be used to find any siblings. If you have the n_{th} child, you cannot find its siblings, such as the n–1 child or the n+1 child, or any other children that are part of the sequence. Only the parent key and chain code can derive all the children. Without the child chain code, the child key cannot be used to derive any grandchildren either. You need both the child private key and the child chain code to start a new branch and derive grandchildren.

So what can the child private key be used for on its own? It can be used to make a public key and a bitcoin address. Then, it can be used to sign transactions to spend anything paid to that address.

> **TIP**
>
> A child private key, the corresponding public key, and the bitcoin address are all indistinguishable from keys and addresses created randomly. The fact that they are part of a sequence is not visible, outside of the HD wallet function that created them. Once created, they operate exactly as "normal" keys.

Extended keys

As we saw earlier, the key derivation function can be used to create children at any level of the tree, based on the three inputs: a key, a chain code, and the index of the desired child. The two essential ingredients are the key and chain code, and combined these are called an *extended key*. The term "extended key" could also be thought of as "extensible key" because such a key can be used to derive children.

Extended keys are stored and represented simply as the concatenation of the 256-bit key and 256-bit chain code into a 512-bit sequence. There are two types of extended keys. An extended private key is the combination of a private key and chain code and can be used to derive child private keys (and from them, child public keys). An extended public key is a public key and chain code, which can be used to create child public keys, as described in Generating a Public Key.

Think of an extended key as the root of a branch in the tree structure of the HD wallet. With the root of the branch, you can derive the rest of the branch. The extended private key can create a complete branch, whereas the extended public key can only create a branch of public keys.

own branch in the tree structure. Sharing an extended key gives access to the entire branch.

Extended keys are encoded using Base58Check, to easily export and import between different BIP0032-compatible wallets. The Base58Check coding for extended keys uses a special version number that results in the prefix "xprv" and "xpub" when encoded in Base58 characters, to make them easily recognizable. Because the extended key is 512 or 513 bits, it is also much longer than other Base58Check-encoded strings we have seen previously.

Here's an example of an extended private key, encoded in Base58Check:

```
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE
```

Here's the corresponding extended public key, also encoded in Base58Check:

```
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrt
```

Public child key derivation

As mentioned previously, a very useful characteristic of hierarchical deterministic wallets is the ability to derive public child keys from public parent keys, *without* having the private keys. This gives us two ways to derive a child public key: either from the child private key, or directly from the parent public key.

An extended public key can be used, therefore, to derive all of the *public* keys (and only the public keys) in that branch of the HD wallet structure.

This shortcut can be used to create very secure public-key-only deployments where a server or application has a copy of an extended public key and no private keys whatsoever. That kind of deployment can produce an infinite number of public keys and bitcoin addresses, but cannot spend any of the money sent to those addresses. Meanwhile, on another, more secure server, the extended private key can derive all the corresponding private keys to sign transactions and spend the money.

will not have any private keys that would be vulnerable to theft. Without HD wallets, the only way to do this is to generate thousands of bitcoin addresses on a separate secure server and then preload them on the ecommerce server. That approach is cumbersome and requires constant maintenance to ensure that the ecommerce server doesn't "run out" of keys.

Another common application of this solution is for cold-storage or hardware wallets. In that scenario, the extended private key can be stored on a paper wallet or hardware device (such as a Trezor hardware wallet), while the extended public key can be kept online. The user can create "receive" addresses at will, while the private keys are safely stored offline. To spend the funds, the user can use the extended private key on an offline signing bitcoin client or sign transactions on the hardware wallet device (e.g., Trezor). Figure 4-12 illustrates the mechanism for extending a parent public key to derive child public keys.

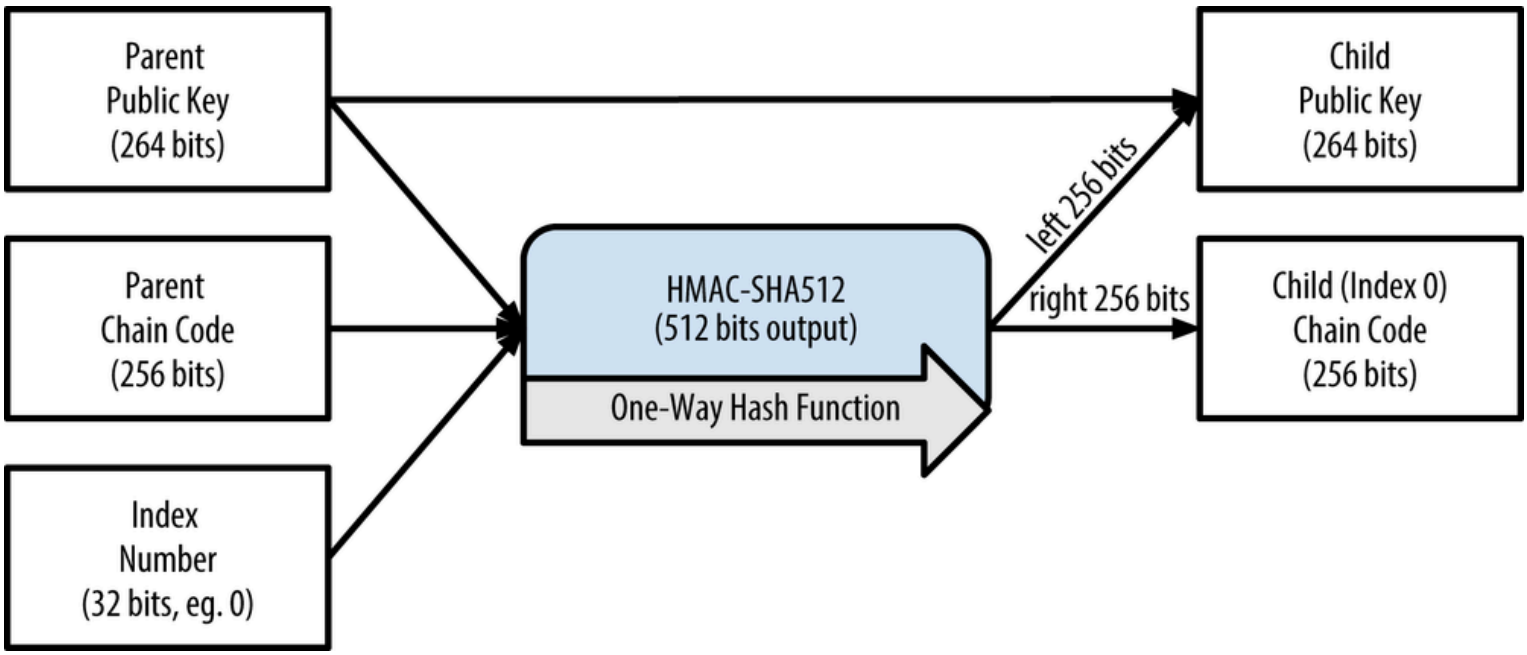

Figure 4-12. Extending a parent public key to create a child public key

Hardened child key derivation

The ability to derive a branch of public keys from an extended public key is very useful, but it comes with a potential risk. Access to an extended public key does not give access to child private keys. However, because the extended public key contains the chain code, if a child private key is known, or somehow leaked, it can be used with the chain code to derive all the other child private keys. A

To counter this risk, HD wallets use an alternative derivation function called *hardened derivation*, which "breaks" the relationship between parent public key and child chain code. The hardened derivation function uses the parent private key to derive the child chain code, instead of the parent public key. This creates a "firewall" in the parent/child sequence, with a chain code that cannot be used to compromise a parent or sibling private key. The hardened derivation function looks almost identical to the normal child private key derivation, except that the parent private key is used as input to the hash function, instead of the parent public key, as shown in the diagram in Figure 4-13.

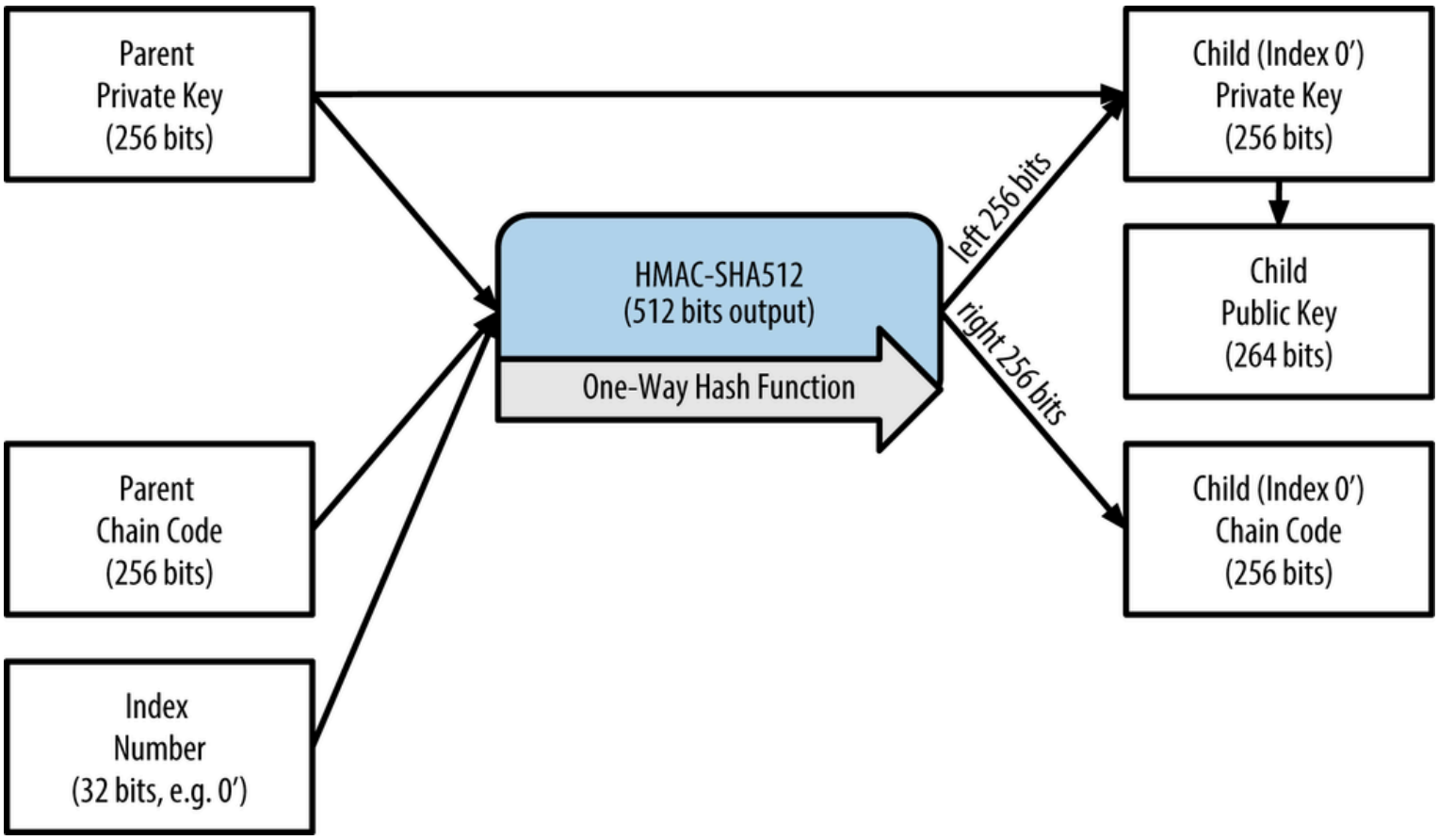

Figure 4-13. Hardened derivation of a child key; omits the parent public key

When the hardened private derivation function is used, the resulting child private key and chain code are completely different from what would result from the normal derivation function. The resulting "branch" of keys can be used to produce extended public keys that are not vulnerable, because the chain code they contain cannot be exploited to reveal any private keys. Hardened derivation is therefore used to create a "gap" in the tree above the level where extended public keys are used.

keys are always derived through the hardened derivation, to prevent compromise of the master keys.

Index numbers for normal and hardened derivation

The index number used in the derivation function is a 32-bit integer. To easily distinguish between keys derived through the normal derivation function versus keys derived through hardened derivation, this index number is split into two ranges. Index numbers between 0 and 2^{31}–1 (0x0 to 0x7FFFFFFF) are used *only* for normal derivation. Index numbers between 2^{31} and 2^{32}–1 (0x80000000 to 0xFFFFFFFF) are used *only* for hardened derivation. Therefore, if the index number is less than 2^{31}, that means the child is normal, whereas if the index number is equal or above 2^{31}, the child is hardened.

To make the index number easier to read and display, the index number for hardened children is displayed starting from zero, but with a prime symbol. The first normal child key is therefore displayed as 0, whereas the first hardened child (index 0x80000000) is displayed as 0'. In sequence then, the second hardened key would have index 0x80000001 and would be displayed as 1', and so on. When you see an HD wallet index i', that means 2^{31}+i.

HD wallet key identifier (path)

Keys in an HD wallet are identified using a "path" naming convention, with each level of the tree separated by a slash (/) character (see Table 4-8). Private keys derived from the master private key start with "m". Public keys derived from the master public key start with "M". Therefore, the first child private key of the master private key is m/0. The first child public key is M/0. The second grandchild of the first child is m/0/1, and so on.

The "ancestry" of a key is read from right to left, until you reach the master key from which it was derived. For example, identifier m/x/y/z describes the key that is the z-th child of key m/x/y, which is the y-th child of key m/x, which is the x-th child of m.

m/0	The first (0) child private key from the master private key (m)
m/0/0	The first grandchild private key of the first child (m/0)
m/0'/0	The first normal grandchild of the first *hardened* child (m/0')
m/1/0	The first grandchild private key of the second child (m/1)
M/23/17/0/0	The first great-great-grandchild public key of the first great-grandchild of the 18th grandchild of the 24th child

Navigating the HD wallet tree structure

The HD wallet tree structure offers tremendous flexibility. Each parent extended key can have 4 billion children: 2 billion normal children and 2 billion hardened children. Each of those children can have another 4 billion children, and so on. The tree can be as deep as you want, with an infinite number of generations. With all that flexibility, however, it becomes quite difficult to navigate this infinite tree. It is especially difficult to transfer HD wallets between implementations, because the possibilities for internal organization into branches and subbranches are endless.

Two Bitcoin Improvement Proposals (BIPs) offer a solution to this complexity, by creating some proposed standards for the structure of HD wallet trees. BIP0043 proposes the use of the first hardened child index as a special identifier that signifies the "purpose" of the tree structure. Based on BIP0043, an HD wallet should use only one level-1 branch of the tree, with the index number identifying the structure and namespace of the rest of the tree by defining its purpose. For example, an HD wallet using only branch m/i'/ is intended to signify a specific purpose and that purpose is identified by index number "i".

Extending that specification, BIP0044 proposes a multiaccount structure as "purpose" number `44'` under BIP0043. All HD wallets following the BIP0044 structure are identified by the fact that they

```
m / purpose' / coin_type' / account' / change / address_index
```

The first-level "purpose" is always set to `44'`. The second-level "coin_type" specifies the type of cryptocurrency coin, allowing for multicurrency HD wallets where each currency has its own subtree under the second level. There are three currencies defined for now: Bitcoin is m/44'/0', Bitcoin Testnet is m/44'/1'; and Litecoin is m/44'/2'.

The third level of the tree is "account," which allows users to subdivide their wallets into separate logical subaccounts, for accounting or organizational purposes. For example, an HD wallet might contain two bitcoin "accounts": m/44'/0'/0' and m/44'/0'/1'. Each account is the root of its own subtree.

On the fourth level, "change," an HD wallet has two subtrees, one for creating receiving addresses and one for creating change addresses. Note that whereas the previous levels used hardened derivation, this level uses normal derivation. This is to allow this level of the tree to export extended public keys for use in a nonsecured environment. Usable addresses are derived by the HD wallet as children of the fourth level, making the fifth level of the tree the "address_index." For example, the third receiving address for bitcoin payments in the primary account would be M/44'/0'/0'/0/2. Table 4-9 shows a few more examples.

Table 4-9. BIP0044 HD wallet structure examples

HD path	Key described
M/44'/0'/0'/0/2	The third receiving public key for the primary bitcoin account
M/44'/0'/3'/1/14	The fifteenth change-address public key for the fourth bitcoin account
m/44'/2'/0'/0/1	The second private key in the Litecoin main account, for signing transactions

generating and extending BIP0032 deterministic keys, as well as displaying them in different formats:

```
$ bx seed | bx hd-new > m # create a new master private key from a seed and store in f
$ cat m # show the master extended private key
xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdfGvzghDt39DHPFpovvtWZaRgY5uPwV7RpEgHs7
$ cat m | bx hd-public # generate the M/0 extended public key
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrt
$ cat m | bx hd-private # generate the m/0 extended private key
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE
$ cat m | bx hd-private | bx hd-to-wif # show the private key of m/0 as a WIF
L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBUN
$ cat m | bx hd-public | bx hd-to-address # show the bitcoin address of M/0
1CHCnCjgMNb6digimckNQ6TBVcTWBAmPHK
$ cat m | bx hd-private | bx hd-private --index 12 --hard | bx hd-private --index
xprv9yL8ndfdPVeDWJenF18oiHguRUj8jHmVrqqD97YQHeTcR3LCeh53q5PXPkLsy2kRaqgwoS6YZBLat
```

Advanced Keys and Addresses

In the following sections we will look at advanced forms of keys and addresses, such as encrypted private keys, script and multisignature addresses, vanity addresses, and paper wallets.

Encrypted Private Keys (BIP0038)

Private keys must remain secret. The need for *confidentiality* of the private keys is a truism that is quite difficult to achieve in practice, because 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 might 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 under-

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 kept 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 as input 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 Bit Address (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 back up 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 encrypted keys in Table 4-10 using bitaddress.org to see how you can get the decrypted key by entering the passphrase.

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. Although 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 Pay-to-Script-Hash (P2SH)). 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

```
$ echo dup hash160 [ 89abcdefabbaabbaabbaabbaabbaabbaabbaabba ] equalverify check
$ bx script-encode < script | bx sha256 | bx ripemd160 | bx base58check-encode --
3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM
```

> **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 Chapter 5.

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

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 more easily find the private key of an address starting with a vanity pattern than you can any other address.

In Chapter 1, 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 Base58 alphabet. The search for a pattern like "1Kids" can be seen as searching for an address in the range from `1Kids11111111111111111111111111` to `1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzz`. There are approximately 58^{29} (approximately $1.4 * 10^{51}$) addresses in that range, all starting with "1Kids". Table 4-11 shows the range of addresses that have the prefix 1Kids.

From	1Kids11111111111111111111111111
	1Kids11111111111111111111111112
	1Kids11111111111111111111111113
	…
To	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzz

Let's look at the pattern "1Kids" as a number and see how frequently we might find this pattern in a bitcoin address (see Table 4-12). An average desktop computer PC, without any specialized hardware, can search approximately 100,000 keys per second.

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,

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 Vanity Pool. 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 at the time of this writing), Eugenia can outsource the search for a seven-character pattern vanity address and get results in a few hours instead of having to run a CPU search for months.

Generating a vanity address is a brute-force exercise: try a random key, check the resulting address to see if it matches the desired pattern, repeat until successful. Example 4-8 shows an example of a "vanity miner," a program designed to find vanity addresses, written in C++. The example uses the libbitcoin library, which we introduced in Alternative Clients, Libraries, and Toolkits.

Example 4-8. Vanity address miner

```
#include <bitcoin/bitcoin.hpp>

// The string we are searching for
const std::string search = "1kid";

// Generate a random secret key. A random 32 bytes.
bc::ec_secret random_secret(std::default_random_engine& engine);
// Extract the Bitcoin address from an EC secret.
std::string bitcoin_address(const bc::ec_secret& secret);
// Case insensitive comparison with the search string.
bool match_found(const std::string& address);

int main()
{
    // random_device on Linux uses "/dev/urandom"
    // CAUTION: Depending on implementation this RNG may not be secure enough!
    // Do not use vanity keys generated by this example in production
    std::random_device random;
    std::default_random_engine engine(random());

    // Loop continuously...
    while (true)
    {
```

```
        std::string address = bitcoin_address(secret);
        // Does it match our search string? (1kid)
        if (match_found(address))
        {
            // Success!
            std::cout << "Found vanity address! " << address << std::endl;
            std::cout << "Secret: " << bc::encode_hex(secret) << std::endl;
            return 0;
        }
    }
    // Should never reach here!
    return 0;
}

bc::ec_secret random_secret(std::default_random_engine& engine)
{
    // Create new secret...
    bc::ec_secret secret;
    // Iterate through every byte setting a random value...
    for (uint8_t& byte: secret)
        byte = engine() % std::numeric_limits<uint8_t>::max();
    // Return result.
    return secret;
}

std::string bitcoin_address(const bc::ec_secret& secret)
{
    // Convert secret to pubkey...
    bc::ec_point pubkey = bc::secret_to_public_key(secret);
    // Finally create address.
    bc::payment_address payaddr;
    bc::set_public_key(payaddr, pubkey);
    // Return encoded form.
    return payaddr.encoded();
}

bool match_found(const std::string& address)
{
    auto addr_it = address.begin();
    // Loop through the search string comparing it to the lower case
    // character of the supplied address.
```

```
    // Reached end of search string, so address matches.
    return true;
}
```

> **NOTE**
>
> The example above uses `std::random_device`. Depending on the implementation it may reflect a cryptographically secure random number generator (CSRNG) provided by the underlying operating system. In the case of UNIX-like operating system such as Linux, it draws from `/dev/urandom`. While the random number generator used here is for demonstration purposes, it is *not* appropriate for generating production-quality bitcoin keys as it is not implemented with sufficient security.

The example code must be compiled using a C compiler and linked against the libbitcoin library (which must be first installed on that system). To run the example, run the vanity-miner++ executable with no parameters (see Example 4-9) and it will attempt to find a vanity address starting with "1kid".

Example 4-9. Compiling and running the vanity-miner example

```
$ # Compile the code with g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the example
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Run it again for a different result
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
# Use "time" to see how long it takes to find a result
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real    0m8.868s
```

The example code will take a few seconds to find a match for the three-character pattern "kid", as we can see when we use the `time` Unix command to measure the execution time. Change the `search` pattern in the source code and see how much longer it takes for four- or five-character patterns!

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 his own address, thereby diverting donations to himself. 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 perhaps inspect the first few characters "1J7mdg" 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, as shown in Table 4-13.

Original Random Address	1J7mdg5rbQyUHENYdx39WVWK7s5pE6X2y
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 six characters (two more), expending an effort that is 3,364 times (58 × 58) higher than the effort Eugenia expended for her four-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 because 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.

Table 4-14. Simplest form of a paper wallet—a printout of the bitcoin address and private key.

Public Address	Private Key (WIF)
1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x	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 CD-ROM 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 fireproof safe and "send" bitcoin to their bitcoin address, to implement a simple yet highly effective "cold storage" solution. Figure 4-14 shows a paper wallet generated from the bitaddress.org site.

Figure 4-14. 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

physically retrieved from a safe or other physically secured storage. Figure 4-15 shows a paper wallet with an encrypted private key (BIP0038) created on the bitaddress.org site.

Figure 4-15. An example of an encrypted paper wallet from bitaddress.org. The passphrase is "test."

WARNING

Although 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 some wallets might generate a change address if you spend less than the whole amount. Additionally, if the computer you use to sign the transaction is compromised, you risk exposing the private key. By spending the entire balance of a paper wallet only once, you reduce the risk of key compromise. If you need only a small amount, send any remaining funds to a new paper wallet in the same transaction.

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. Figures 4-16 through 4-18 show various examples of paper wallets with security and backup features.

Figure 4-16. An example of a paper wallet from bitcoinpaperwallet.com with the private key on a folding flap.

Figure 4-17. 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 4-18. An example of a paper wallet with additional copies of the keys on a backup "stub."