Cryptography

One of Ethereum's foundational technologies is *cryptography*, which is a branch of mathematics used extensively in computer security. Cryptography means "secret writing" in Greek, but the study of cryptography encompasses more than just secret writing, which is referred to as *encryption*. Cryptography can, for example, also be used to prove knowledge of a secret without revealing that secret (e.g., with a digital signature), or to prove the authenticity of data (e.g., with digital fingerprints, also known as "hashes"). These types of cryptographic proofs are mathematical tools critical to the operation of the Ethereum platform (and, indeed, all blockchain systems), and are also extensively used in Ethereum applications.

Note that, at the time of publication, no part of the Ethereum protocol involves encryption; that is to say all communications with the Ethereum platform and between nodes (including transaction data) are unencrypted and can (necessarily) be read by anyone. This is so everyone can verify the correctness of state updates and consensus can be reached. In the future, advanced cryptographic tools, such as zero knowledge proofs and homomorphic encryption, will be available that will allow for some encrypted calculations to be recorded on the blockchain while still enabling consensus; however, while provision has been made for them, they have yet to be deployed.

In this chapter we will introduce some of the cryptography used in Ethereum: namely public key cryptography (PKC), which is used to control ownership of funds, in the form of private keys and addresses.

Keys and Addresses

As we saw earlier in the book, Ethereum has two different types of accounts: *externally owned accounts* (EOAs) and *contracts*. Ownership of ether by EOAs is established through digital *private keys*, *Ethereum addresses*, and *digital signatures*. The private keys are at the heart of all user interaction with Ethereum. In fact, account addresses are derived directly from private keys: a private key uniquely determines a single Ethereum address, also known as an *account*.

Private keys are not used directly in the Ethereum system in any way; they are never transmitted or stored on Ethereum. That is to say that private keys should remain private and never appear in messages passed to the network, nor should they be stored on-chain; only account addresses and digital signatures are ever transmitted and stored on the Ethereum system. For more information on how to keep private keys safe and secure, see [control_responsibility] and [wallets_chapter].

Access and control of funds is achieved with digital signatures, which are also created using the private key. Ethereum transactions require a valid digital signature to be included in the blockchain. Anyone with a copy of a private key has control of the corresponding account and any ether it holds. Assuming a user keeps their private key safe, the digital signatures in Ethereum transactions prove the true owner of the funds, because they prove ownership of the private key.

In public key cryptography–based systems, such as that used by Ethereum, 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; it is the latter that provides control over the account, and the former that identifies it to others. The private keys themselves are very rarely seen by Ethereum users; for the most part, they are stored (in encrypted form) in

special files and managed by Ethereum wallet software.

In the payment portion of an Ethereum transaction, the intended recipient is represented by an Ethereum address, which is used in the same way as the beneficiary account details of a bank transfer. As we will see in more detail shortly, an Ethereum address for an EOA is generated from the public key portion of a key pair. However, not all Ethereum addresses represent public–private key pairs; they can also represent contracts, which, as we will see in [smart_contracts_chapter], are not backed by private keys.

In the rest of this chapter, we will first explore basic cryptography in a bit more detail and explain the mathematics used in Ethereum. Then we will look at how keys are generated, stored, and managed. Finally, we will review the various encoding formats used to represent private keys, public keys, and addresses.

Public Key Cryptography and Cryptocurrency

Public key cryptography (also called "asymmetric cryptography") is a core part of modern-day information security. The key exchange protocol, first published in the 1970s by Martin Hellman, Whitfield Diffie, and Ralph Merkle, was a monumental breakthrough that incited the first big wave of public interest in the field of cryptography. Before the 1970s, strong cryptographic knowledge was kept secret by governments.

Public key cryptography uses unique keys to secure information. These keys are based on mathematical functions that have a special property: it is easy to calculate them, but hard to calculate their inverse. Based on these functions, cryptography enables the creation of digital secrets and unforgeable digital signatures, which are secured by the laws of mathematics.

For example, multiplying two large prime numbers together is trivial. But given the product of two large primes, it is very difficult to find the prime factors (a problem called *prime factorization*). Let's say we present the number 8,018,009 and tell you it is the product of two primes. Finding those two primes is much harder for you than it was for me to multiply them to produce 8,018,009.

Some of these mathematical functions can be inverted easily if you know some secret information. In the preceding example, if I tell you that one of the prime factors is 2,003, you can trivially find the other one with a simple division: 8,018,009 ÷ 2,003 = 4,003. Such functions are often called *trapdoor functions* because they are very difficult to invert unless you are given a piece of secret information that can be used as a shortcut to reverse the function.

A more advanced category of mathematical functions that is useful in cryptography is based on arithmetic operations on an elliptic curve. In elliptic curve arithmetic, multiplication modulo a prime is simple but division (the inverse) is practically impossible. This is called the *discrete logarithm problem* and there are currently no known trapdoors. *Elliptic curve cryptography* is used extensively in modern computer systems and is the basis of Ethereum's (and other cryptocurrencies') use of private keys and digital signatures.

NOTE

Take a look at the following resources if you're interested in reading more about cryptography and the mathematical functions that are used in modern cryptography:

- Cryptography
- Trapdoor function
- Prime factorization
- Discrete logarithm
- Elliptic curve cryptography

In Ethereum, we use public key cryptography (also known as asymmetric cryptography) to create the public–private key pair we have been talking about in this chapter. They are considered a "pair" because the public key is derived from the private key. Together, they represent an Ethereum account by providing, respectively, a publicly accessible account handle (the address) and private control over access to any ether in the account and over any authentication the account needs when using smart contracts. The private key controls access by being the unique piece of information needed to create *digital signatures*, which are required to sign transactions to spend any funds in the account. Digital signatures are also used to authenticate owners or users of contracts, as we will see in [smart_contracts_chapter].

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 trivially calculated from the private key, so storing only the private key is also possible.

A digital signature can be created to sign any message. For Ethereum transactions, the details of the transaction itself are used as the message. The mathematics of cryptography—in this case, elliptic curve cryptography—provides a way for the message (i.e., the transaction details) to be combined with the private key to create a code that can only be produced with knowledge of the private key. That code is called the digital signature. Note that an Ethereum transaction is basically a request to access a particular account with a particular Ethereum address. When a transaction is sent to the Ethereum network in order to move funds or interact with smart contracts, it needs to be sent with a digital signature created with the private key corresponding to the Ethereum address in question. Elliptic curve mathematics means that *anyone* can verify that a transaction is valid, by checking that the digital signature matches the transaction details *and* the Ethereum address to which access is being requested. The verification doesn't involve the private key at all; that remains private. However, the verification process determines beyond doubt that the transaction could have only come from someone with the private key that corresponds to the public key behind the Ethereum address. This is the "magic" of public key cryptography.

TIP

There is no encryption as part of the Ethereum protocol—all messages that are sent as part of the operation of the Ethereum network can (necessarily) be read by everyone. As such, private keys are only used to create digital signatures for transaction authentication.

Private Keys

A private key is simply a number, picked at random. Ownership and control of the private key is the root of user control over all funds associated with the corresponding Ethereum address, as well as access to contracts that authorize that address. The private key is used to create signatures required to spend ether by proving ownership of funds used in a transaction. The private key must remain secret at all times, because revealing it to third parties is equivalent to giving them control over the ether and contracts secured by that private key. The private key must also be backed up and protected from accidental loss. If it's lost, it cannot be recovered and the funds secured by it are lost forever too.

TIP The Ethereum private key is just a number. One way to pick your private keys randomly is to simply use 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 an Ethereum wallet (probably—see the next section). The public key and address 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 an Ethereum private key essentially involves picking 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 deterministic. Ethereum software uses the underlying operating system's random number generator to produce 256 random bits. 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, or press random keys on your keyboard. An alternative could be cosmic radiation noise on the computer's microphone channel.

More precisely, a private key can be any nonzero number up to a very large number slightly less than 2^{256}—a huge 78-digit number, roughly $1.158 * 10^{77}$. The exact number shares the first 38 digits with 2^{256} and is defined as the order of the elliptic curve used in Ethereum (see Elliptic Curve Cryptography Explained). To create a private key, we randomly pick a 256-bit number and check that it is within the valid range. In programming terms, this is usually achieved by feeding an even larger string of random bits (collected from a cryptographically secure source of randomness) into a 256-bit hash algorithm such as Keccak-256 or SHA-256, both of which will conveniently produce a 256-bit number. If the result is within the valid range, we have a suitable private key. Otherwise, we simply try again with another random number.

TIP 2^{256}—the size of Ethereum's private key space—is an unfathomably large number. It is approximately 10^{77} in decimal; that is, a number with 77 digits. For comparison, the visible universe is estimated to contain between 10^77 and 10^80 atoms. Therefore, at the lower range there are enough private keys to give every atom in the universe an Ethereum account. If you pick a private key randomly, there is no conceivable way anyone will ever guess it or pick it themselves.

Note that the private key generation process is an offline one; it does not require any communication with the Ethereum network, or indeed any communication with anyone at all. As

such, in order to pick a number that no one else will ever pick, it needs to be truly random. If you choose the number yourself, the chance that someone else will try it (and then run off with your ether) is too high. Using a bad random number generator (like the pseudorandom rand function in most programming languages) is even worse, because it is even more obvious and even easier to replicate. Just like with passwords for online accounts, the private key needs to be unguessable. Fortunately, you never need to remember your private key, so you can take the best possible approach for picking it: namely, true randomness.

WARNING	Do not write your own code to create a random number or use a "simple" random number generator offered by your programming language. It is vital that you use a cryptographically secure pseudo-random number generator (such as 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 library is critical to the security of the keys.

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

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Public Keys

An Ethereum public key is a *point* on an elliptic curve, meaning it is a set of *x* and *y* coordinates that satisfy the elliptic curve equation.

In simpler terms, an Ethereum public key is two numbers, joined together. These numbers are produced from the private key by a calculation that can *only go one way*. That means that it is trivial to calculate a public key if you have the private key, but you cannot calculate the private key from the public key.

WARNING	MATH is about to happen! Don’t panic. If you start to get lost at any point in the following paragraphs, you can skip the next few sections. There are many tools and libraries that will do the math for you.

The public key is calculated from the private key using elliptic curve multiplication, which is practically irreversible: $K = k * G$, where k is the private key, G is a constant point called the *generator point*, K is the resulting public key, and * is the special elliptic curve "multiplication" operator. Note that elliptic curve multiplication is not like normal multiplication. It shares functional attributes with normal multiplication, but that is about it. For example, the reverse operation (which would be division for normal numbers), known as "finding the discrete logarithm”—i.e., calculating k if you know K—is as difficult as trying all possible values of k (a brute-force search that will likely take more time than this universe will allow for).

In simpler terms: arithmetic on the elliptic curve is different from "regular" integer arithmetic. A point (G) can be multiplied by an integer (k) to produce another point (K). But there is no such thing as *division*, so it is not possible to simply "divide" the public key K by the point G to calculate the

private key *k*. This is the one-way mathematical function described in Public Key Cryptography and Cryptocurrency.

NOTE

Elliptic curve multiplication is a type of function that cryptographers call a "one-way" function: it is easy to do in one direction (multiplication) and impossible to do in the reverse direction (division). The owner of the private key can easily create the public key and then share it with the world, knowing that no one can reverse the function and calculate the private key from the public key. This mathematical trick becomes the basis for unforgeable and secure digital signatures that prove ownership of Ethereum funds and control of contracts.

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.

A visualization of an elliptic curve is an example of an elliptic curve, similar to that used by Ethereum.

NOTE

Ethereum uses the exact same elliptic curve, called secp256k1, as Bitcoin. That makes it possible to reuse many of the elliptic curve libraries and tools from Bitcoin.

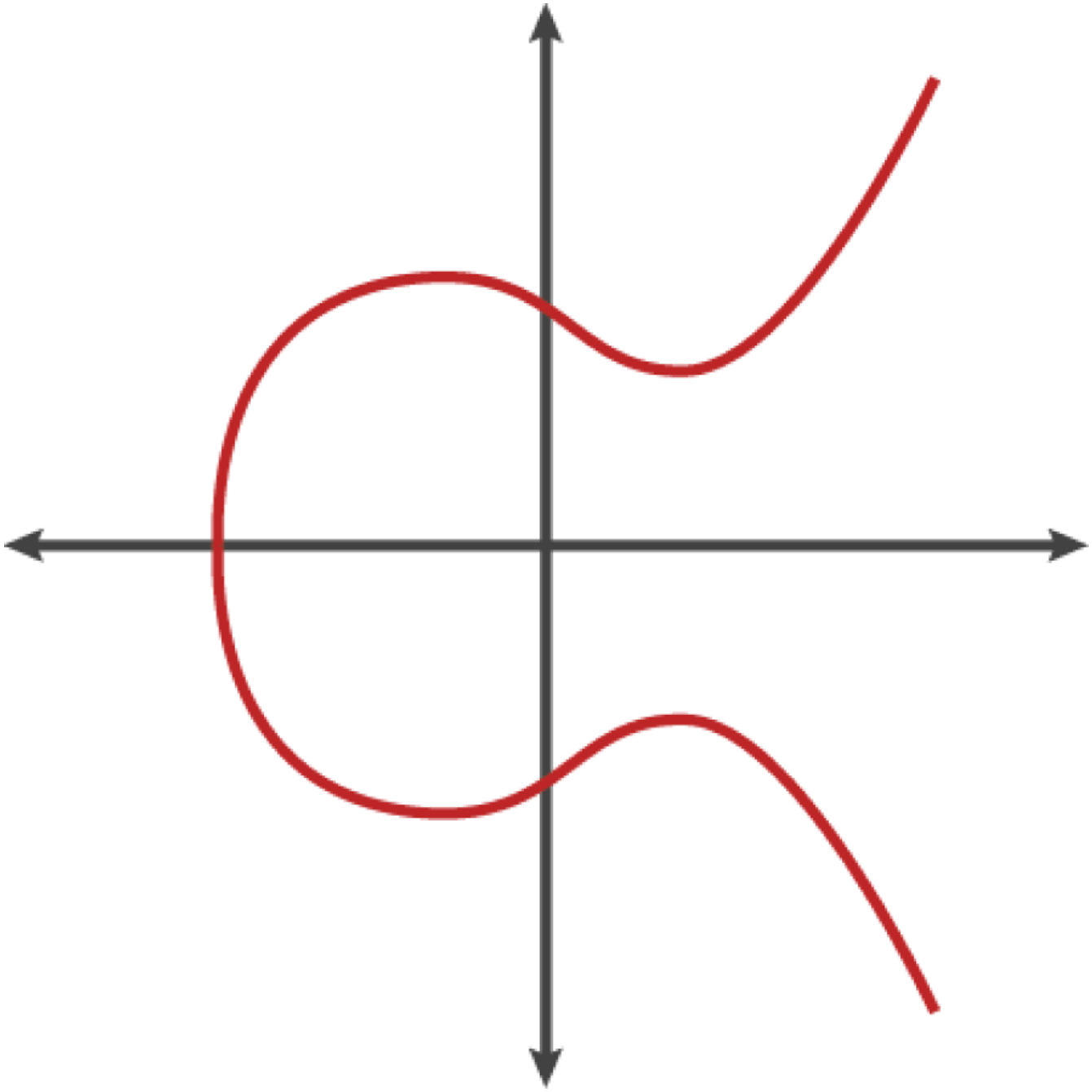

Figure 1. A visualization of an elliptic curve

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

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mrow>
      <msup><mi>y</mi> <mn>2</mn> </msup>
      <mo>=</mo>
      <mrow>
        <mo>(</mo>
        <msup><mi>x</mi> <mn>3</mn> </msup>
        <mo>+</mo>
        <mn>7</mn>
        <mo>)</mo>
      </mrow>
    </mrow>
    <mspace width="3.33333pt"/>
    <mtext>over</mtext>
    <mspace width="3.33333pt"/>
    <mrow>
      <mo>(</mo>
      <msub><mi>&#x1d53d;</mi> <mi>p</mi> </msub>
      <mo>)</mo>
    </mrow>
  </mrow>
</math>
</div>
```

or:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <msup><mi>y</mi> <mn>2</mn> </msup>
    <mspace width="3.33333pt"/>
    <mo form="prefix">mod</mo>
    <mspace width="0.277778em"/>
    <mi>p</mi>
    <mo>=</mo>
    <mrow>
      <mo>(</mo>
      <msup><mi>x</mi> <mn>3</mn> </msup>
      <mo>+</mo>
      <mn>7</mn>
      <mo>)</mo>
    </mrow>
    <mspace width="3.33333pt"/>
    <mo form="prefix">mod</mo>
    <mspace width="0.277778em"/>
    <mi>p</mi>
  </mrow>
</math>
</div>
```

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 $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, which is 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 to that of an elliptic curve over real numbers. As an example, Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17 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 Ethereum elliptic curve can be thought of as a much more complex pattern of dots on an unfathomably large grid.

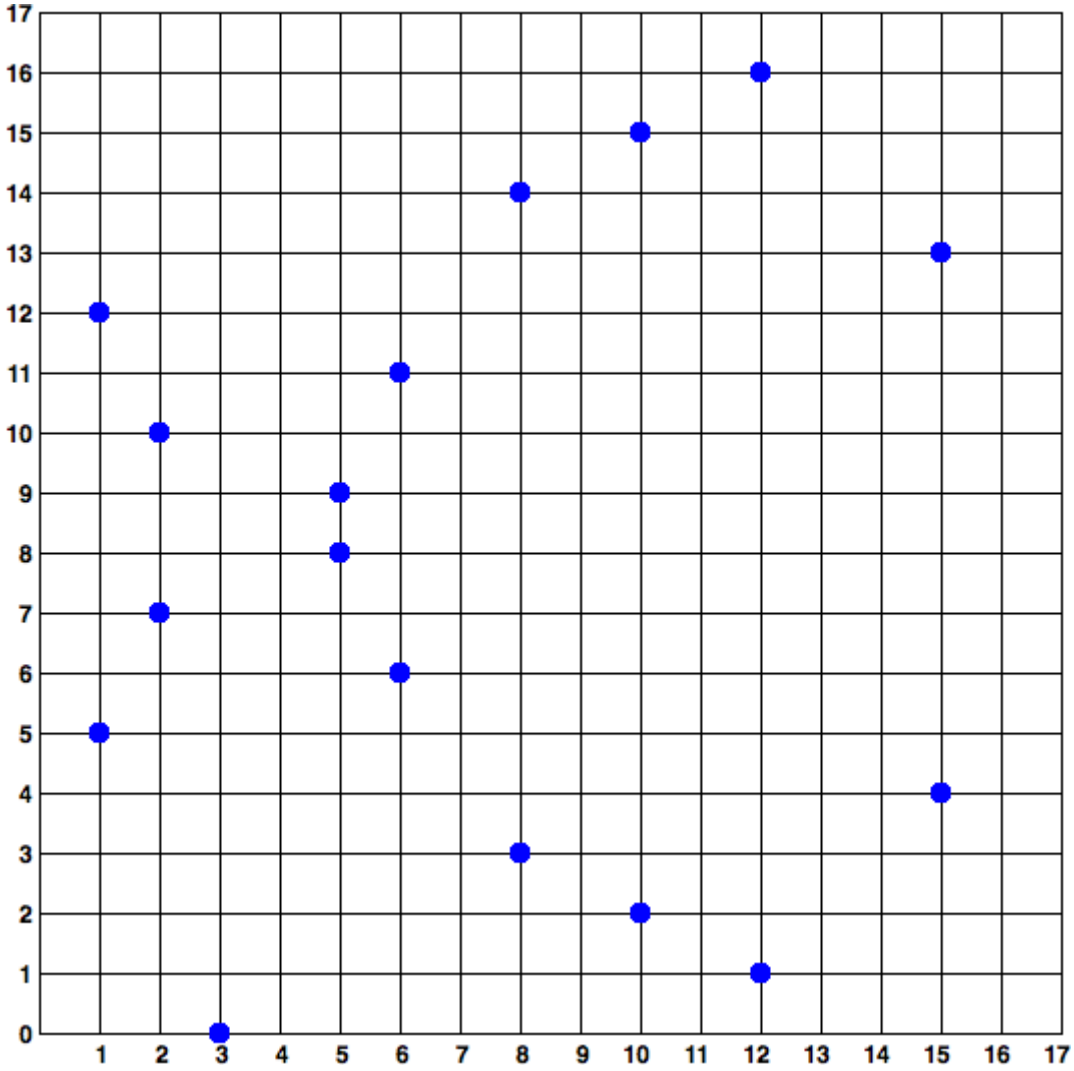

Figure 2. Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17

So, for example, the following is a point *Q* with coordinates (*x*,*y*) that is a point on the secp256k1 curve:

```
Q =
(49790390825249384486033144355916864607616083520101638681403973749255924539515,
595741321618999000458620864939210157800321752917558073992840077210503412973 60)
```

[example_1] shows how you can check this yourself using Python. The variables x and y are the coordinates of the point *Q*, as in the preceding example. The variable p is the prime order of the elliptic curve (the prime that is used for all the modulo operations). The last line of Python is the elliptic curve equation (the % operator in Python is the modulo operator). If x and y are indeed the coordinates of a point on the elliptic curve, then they satisfy the equation and the result is zero (0L is a long integer with value zero). Try it yourself, by typing **python** on a command line and copying each line (after the prompt >>>) from the listing.

```
<div data-type="example" id="example_1">
<h5>Using Python to confirm that this point is on the elliptic curve</h5>
<pre data-type="programlisting">
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.
>>> <strong>p =
115792089237316195423570985008687907853269984665640564039457584007908834 \
671663</strong>
>>> <strong>x =
4979039082524938448603314435591686460761608352010163868140397374925592453951 5</strong>
>>> <strong>y =
595741321618999000458620864939210157800321752917558073992840077210503412973 60</strong>
>>> <strong>(x ** 3 + 7 - y**2) % p</strong>
0L
</pre>
</div>
```

Elliptic Curve Arithmetic Operations

A lot of elliptic curve math looks and works very much like the integer arithmetic we learned at school. Specifically, we can define an addition operator, which instead of jumping along the number line is jumping to other points on the curve. Once we have the addition operator, we can also define multiplication of a point and a whole number, which is equivalent to repeated addition.

Elliptic curve addition is defined such that 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 (amazingly). Call this point $P_3' = (x, y)$. Then reflect in the x-axis to get $P_3 = (x, –y)$.

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

In elliptic curve math, there is also a point called the "point at infinity," which roughly corresponds to the role of the number zero in addition. On computers, it's sometimes represented by $x = y = 0$ (which doesn't satisfy the elliptic curve equation, but it's an easy separate case that can be checked). There are a couple of special cases that explain the need for the point at infinity.

In some cases (e.g., if P_1 and P_2 have the same x values but different y values), the line will be exactly vertical, in which case P_3 = the point at infinity.

If P_1 is the point at infinity, then $P_1 + P_2 = P_2$. Similarly, 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 that zero plays in "normal" arithmetic.

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 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 $k * P = P + P + P + ... + P$ (k times). Note that k is sometimes (perhaps confusingly) called an "exponent" in this case.

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:

```
<div data-type="equation">
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mi>K</mi>
    <mo>=</mo>
    <mi>k</mi>
    <mo>*</mo>
    <mi>G</mi>
  </mrow>
</math>
</div>
```

The generator point is specified as part of the secp256k1 standard; it is the same for all implementations of secp256k1, and all keys derived from that curve use the same point G. Because the generator point is always the same for all Ethereum 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 an Ethereum address (derived from K) can be shared with anyone and does not reveal the user's private key (k).

As we described in the previous section, the multiplication of $k * G$ is equivalent to repeated addition, so $G + G + G + ... + G$, repeated k times. In summary, to produce a public key K from a private key k, we add the generator point G to itself, k times.

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.

Let's apply this calculation to find the public key for the specific private key we showed you in Private Keys:

Example private key to public key calculation

```
K = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 * G
```

A cryptographic library can help us calculate K, using elliptic curve multiplication. The resulting public key K is defined as the point:

```
K = (x, y)
```

where:

```
x = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b
y = 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

In Ethereum you may see public keys represented as a serialization of 130 hexadecimal characters (65 bytes). This is adopted from a standard serialization format proposed by the industry consortium Standards for Efficient Cryptography Group (SECG), documented in Standards for Efficient Cryptography (SEC1). The standard defines four possible prefixes that can be used to identify points on an elliptic curve, listed in Serialized EC public key prefixes.

Table 1. Serialized EC public key prefixes

Prefix	Meaning	Length (bytes counting prefix)
0x00	Point at infinity	1
0x04	Uncompressed point	65
0x02	Compressed point with even y	33
0x03	Compressed point with odd y	33

Ethereum only uses uncompressed public keys; therefore the only prefix that is relevant is (hex) 04. The serialization concatenates the *x* and *y* coordinates of the public key:

```
04 + x-coordinate (32 bytes/64 hex) + y-coordinate (32 bytes/64 hex)
```

Therefore, the public key we calculated earlier is serialized as:

```
046e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0 \
c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

Elliptic Curve Libraries

There are a couple of implementations of the secp256k1 elliptic curve that are used in cryptocurrency-related projects:

OpenSSL

The OpenSSL library offers a comprehensive set of cryptographic primitives, including a full implementation of secp256k1. For example, to derive the public key, the function EC_POINT_mul can be used.

libsecp256k1

Bitcoin Core's libsecp256k1 is a C-language implementation of the secp256k1 elliptic curve and

other cryptographic primitives. It was written from scratch to replace OpenSSL in Bitcoin Core software, and is considered superior in both performance and security.

Cryptographic Hash Functions

Cryptographic hash functions are used throughout Ethereum. In fact, hash functions are used extensively in almost all cryptographic systems—a fact captured by cryptographer Bruce Schneier, who said, "Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography."

In this section we will discuss hash functions, explore their basic properties, and see how those properties make them so useful in so many areas of modern cryptography. We address hash functions here because they are part of the transformation of Ethereum public keys into addresses. They can also be used to create *digital fingerprints*, which aid in the verification of data.

In simple terms, a *hash function* is “any function that can be used to map data of arbitrary size to data of fixed size.” The input to a hash function is called a *pre-image*, the *message*, or simply the *input data*. The output is called the *hash*. *Cryptographic hash functions* are a special subcategory that have specific properties that are useful to secure platforms, such as Ethereum.

A cryptographic hash function is a *one-way* hash function that maps data of arbitrary size to a fixed-size string of bits. The "one-way" nature means that it is computationally infeasible to recreate the input data if one only knows the output hash. The only way to determine a possible input is to conduct a brute-force search, checking each candidate for a matching output; given that the search space is virtually infinite, it is easy to understand the practical impossibility of the task. Even if you find some input data that creates a matching hash, it may not be the original input data: hash functions are "many-to-one" functions. Finding two sets of input data that hash to the same output is called finding a *hash collision*. Roughly speaking, the better the hash function, the rarer hash collisions are. For Ethereum, they are effectively impossible.

Let’s take a closer look at the main properties of cryptographic hash functions. These include:

Determinism

A given input message always produces the same hash output.

Verifiability

Computing the hash of a message is efficient (linear complexity).

Noncorrelation

A small change to the message (e.g., a 1-bit change) should change the hash output so extensively that it cannot be correlated to the hash of the original message.

Irreversibility

Computing the message from its hash is infeasible, equivalent to a brute-force search through all possible messages.

Collision protection

It should be infeasible to calculate two different messages that produce the same hash output.

Resistance to hash collisions is particularly important for avoiding digital signature forgery in Ethereum.

The combination of these properties make cryptographic hash functions useful for a broad range of security applications, including:

- Data fingerprinting
- Message integrity (error detection)
- Proof of work
- Authentication (password hashing and key stretching)
- Pseudorandom number generators
- Message commitment (commit–reveal mechanisms)
- Unique identifiers

We will find many of these in Ethereum as we progress through the various layers of the system.

Ethereum's Cryptographic Hash Function: Keccak-256

Ethereum uses the *Keccak-256* cryptographic hash function in many places. Keccak-256 was designed as a candidate for the SHA-3 Cryptographic Hash Function Competition held in 2007 by the National Institute of Standards and Technology. Keccak was the winning algorithm, which became standardized as Federal Information Processing Standard (FIPS) 202 in 2015.

However, during the period when Ethereum was developed, the NIST standardization was not yet finalized. NIST adjusted some of the parameters of Keccak after the completion of the standards process, allegedly to improve its efficiency. This was occurring at the same time as heroic whistleblower Edward Snowden revealed documents that imply that NIST may have been improperly influenced by the National Security Agency to intentionally weaken the Dual_EC_DRBG random-number generator standard, effectively placing a backdoor in the standard random number generator. The result of this controversy was a backlash against the proposed changes and a significant delay in the standardization of SHA-3. At the time, the Ethereum Foundation decided to implement the original Keccak algorithm, as proposed by its inventors, rather than the SHA-3 standard as modified by NIST.

WARNING	While you may see "SHA-3" mentioned throughout Ethereum documents and code, many if not all of those instances actually refer to Keccak-256, not the finalized FIPS-202 SHA-3 standard. The implementation differences are slight, having to do with padding parameters, but they are significant in that Keccak-256 produces different hash outputs from FIPS-202 SHA-3 for the same input.

Which Hash Function Am I Using?

How can you tell if the software library you are using implements FIPS-202 SHA-3 or Keccak-256, if both might be called "SHA-3"?

An easy way to tell is to use a *test vector*, an expected output for a given input. The test most commonly used for a hash function is the *empty input*. If you run the hash function with an empty

string as input you should see the following results:

```
Keccak256("") =
  c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470

SHA3("") =
  a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

Regardless of what the function is called, you can test it to see whether it is the original Keccak-256 or the final NIST standard FIPS-202 SHA-3 by running this simple test. Remember, Ethereum uses Keccak-256, even though it is often called SHA-3 in the code.

NOTE	Due to the confusion created by the difference between the hash function used in Ethereum (Keccak-256) and the finalized standard (FIP-202 SHA-3), there is an effort underway to rename all instances of sha3 in all code, opcodes, and libraries to keccak256. See EIP-59 for details.

Next, let's examine the first application of Keccak-256 in Ethereum, which is to produce Ethereum addresses from public keys.

Ethereum Addresses

Ethereum addresses are *unique identifiers* that are derived from public keys or contracts using the Keccak-256 one-way hash function.

In our previous examples, we started with a private key and used elliptic curve multiplication to derive a public key:

Private key *k*:

```
k = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Public key *K* (*x* and *y* coordinates concatenated and shown as hex):

```
K = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e...
```

NOTE	It is worth noting that the public key is not formatted with the prefix (hex) 04 when the address is calculated.

We use Keccak-256 to calculate the *hash* of this public key:

```
Keccak256(K) = 2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Then we keep only the last 20 bytes (least significant bytes), which is our Ethereum address:

```
001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Most often you will see Ethereum addresses with the prefix 0x that indicates they are hexadecimal-encoded, like this:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Ethereum Address Formats

Ethereum addresses are hexadecimal numbers, identifiers derived from the last 20 bytes of the Keccak-256 hash of the public key.

Unlike Bitcoin addresses, which are encoded in the user interface of all clients to include a built-in checksum to protect against mistyped addresses, Ethereum addresses are presented as raw hexadecimal without any checksum.

The rationale behind that decision was that Ethereum addresses would eventually be hidden behind abstractions (such as name services) at higher layers of the system and that checksums should be added at higher layers if necessary.

In reality, these higher layers were developed too slowly and this design choice led to a number of problems in the early days of the ecosystem, including the loss of funds due to mistyped addresses and input validation errors. Furthermore, because Ethereum name services were developed slower than initially expected, alternative encodings were adopted very slowly by wallet developers. We'll look at a few of the encoding options next.

Inter Exchange Client Address Protocol

The *Inter exchange Client Address Protocol* (ICAP) is an Ethereum address encoding that is partly compatible with the International Bank Account Number (IBAN) encoding, offering a versatile, checksummed, and interoperable encoding for Ethereum addresses. ICAP addresses can encode Ethereum addresses or common names registered with an Ethereum name registry. You can read more about ICAP on the Ethereum Wiki.

IBAN is an international standard for identifying bank account numbers, mostly used for wire transfers. It is broadly adopted in the European Single Euro Payments Area (SEPA) and beyond. IBAN is a centralized and heavily regulated service. ICAP is a decentralized but compatible implementation for Ethereum addresses.

An IBAN consists of a string of up to 34 alphanumeric characters (case-insensitive) comprising a country code, checksum, and bank account identifier (which is country-specific).

ICAP uses the same structure by introducing a nonstandard country code, "XE," that stands for "Ethereum," followed by a two-character checksum and three possible variations of an account identifier:

Direct

A big-endian base-36 integer comprised of up to 30 alphanumeric characters, representing the 155 least significant bits of an Ethereum address. Because this encoding fits less than the full 160 bits of a general Ethereum address, it only works for Ethereum addresses that start with one or more zero bytes. The advantage is that it is compatible with IBAN, in terms of the field length and checksum. Example: XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD (33 characters long).

Basic

Same as the Direct encoding, except that it is 31 characters long. This allows it to encode any Ethereum address, but makes it incompatible with IBAN field validation. Example: XE18CHDJBPLTBCJ03FE9O2NS0BPOJVQCU2P (35 characters long).

Indirect

Encodes an identifier that resolves to an Ethereum address through a name registry provider. It uses 16 alphanumeric characters, comprising an *asset identifier* (e.g., ETH), a name service (e.g., XREG), and a 9-character human-readable name (e.g., KITTYCATS). Example: XE##ETHXREGKITTYCATS (20 characters long), where the ## should be replaced by the two computed checksum characters.

We can use the helpeth command-line tool to create ICAP addresses. You can get helpeth by installing it with:

```
<pre data-type="programlisting">
$ <strong>npm install -g helpeth</strong>
</pre>
```

If you don't have npm, you may have to install nodeJS first, which you can do by following the instructions at https://nodeJS.org.

Now that we have helpeth, let's try creating an ICAP address with our example private key (prefixed with 0x and passed as a parameter to helpeth).

```
<pre data-type="programlisting">
$ <strong>helpeth keyDetails \
  -p 0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315</strong>

Address: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
ICAP: XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D
Public key: 0x6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b...
</pre>
```

The helpeth command constructs a hexadecimal Ethereum address as well as an ICAP address for us. The ICAP address for our example key is:

```
XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD
```

Because our example Ethereum address happens to start with a zero byte, it can be encoded using the Direct ICAP encoding method that is valid in IBAN format. You can tell because it is 33 characters long.

If our address did not start with a zero, it would be encoded with the Basic encoding, which would be 35 characters long and invalid as an IBAN.

TIP	The chances of any Ethereum address starting with a zero byte are 1 in 256. To generate one like that, it will take on average 256 attempts with 256 different random private keys before we find one that works as an IBAN-compatible "Direct" encoded ICAP address.

At this time, ICAP is unfortunately only supported by a few wallets.

Hex Encoding with Checksum in Capitalization (EIP-55)

Due to the slow deployment of ICAP and name services, a standard was proposed by Ethereum Improvement Proposal 55 (EIP-55). EIP-55 offers a backward-compatible checksum for Ethereum addresses by modifying the capitalization of the hexadecimal address. The idea is that Ethereum addresses are case-insensitive and all wallets are supposed to accept Ethereum addresses expressed in capital or lowercase characters, without any difference in interpretation.

By modifying the capitalization of the alphabetic characters in the address, we can convey a checksum that can be used to protect the integrity of the address against typing or reading mistakes. Wallets that do not support EIP-55 checksums simply ignore the fact that the address contains mixed capitalization, but those that do support it can validate it and detect errors with a 99.986% accuracy.

The mixed-capitals encoding is subtle and you may not notice it at first. Our example address is:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

With an EIP-55 mixed-capitalization checksum it becomes:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Can you tell the difference? Some of the alphabetic (A–F) characters from the hexadecimal encoding alphabet are now capital, while others are lowercase.

EIP-55 is quite simple to implement. We take the Keccak-256 hash of the lowercase hexadecimal address. This hash acts as a digital fingerprint of the address, giving us a convenient checksum. Any small change in the input (the address) should cause a big change in the resulting hash (the checksum), allowing us to detect errors effectively. The hash of our address is then encoded in the capitalization of the address itself. Let's break it down, step by step:

1. Hash the lowercase address, without the 0x prefix:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9") =
23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. Capitalize each alphabetic address character if the corresponding hex digit of the hash is greater than or equal to 0x8. This is easier to show if we line up the address and the hash:

```
Address: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9
Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Our address contains an alphabetic character d in the fourth position. The fourth character of the hash is 6, which is less than 8. So, we leave the d lowercase. The next alphabetic character in our address is f, in the sixth position. The sixth character of the hexadecimal hash is c, which is greater than 8. Therefore, we capitalize the F in the address, and so on. As you can see, we only use the first 20 bytes (40 hex characters) of the hash as a checksum, since we only have 20 bytes (40 hex characters) in the address to capitalize appropriately.

Check the resulting mixed-capitals address yourself and see if you can tell which characters were capitalized and which characters they correspond to in the address hash:

```
Address: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Detecting an error in an EIP-55 encoded address

Now, let's look at how EIP-55 addresses will help us find an error. Let's assume we have printed out an Ethereum address, which is EIP-55 encoded:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Now let's make a basic mistake in reading that address. The character before the last one is a capital F. For this example let's assume we misread that as a capital E, and we type the following (incorrect) address into our wallet:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

Fortunately, our wallet is EIP-55 compliant! It notices the mixed capitalization and attempts to validate the address. It converts it to lowercase, and calculates the checksum hash:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9") =
5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

As you can see, even though the address has only changed by one character (in fact, only one bit, as e and f are one bit apart), the hash of the address has changed radically. That's the property of hash

functions that makes them so useful for checksums!

Now, let's line up the two and check the capitalization:

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

It's all wrong! Several of the alphabetic characters are incorrectly capitalized. Remember that the capitalization is the encoding of the *correct* checksum.

The capitalization of the address we input doesn't match the checksum just calculated, meaning something has changed in the address, and an error has been introduced.

Conclusions

In this chapter we provided a brief survey of public key cryptography and focused on the use of public and private keys in Ethereum and the use of cryptographic tools, such as hash functions, in the creation and verification of Ethereum addresses. We also looked at digital signatures and how they can demonstrate ownership of a private key without revealing that private key. In [wallets_chapter], we will put these ideas together and look at how wallets can be used to manage collections of keys.