# A brief Introduction to Ed25519

Speaker: Roger

# Introduction

what is Ed25519, have you seen it before?

> 注意：GitHub 通过在 2022 年 3 月 15 日删除旧的、不安全的密钥类型来提高安全性。
>
> 自该日期起，不再支持 DSA 密钥 (`ssh-dss`)。 无法在 GitHub.com上向个人帐户添加新的 DSA 密钥。
>
> 2021 年 11 月 2 日之前带有 `valid_after` 的 RSA 密钥 (`ssh-rsa`) 可以继续使用任何签名算法。 在该日期之后生成的 RSA 密钥必须使用 SHA-2 签名算法。 一些较旧的客户端可能需要升级才能使用 SHA-2 签名。

1. 将 SSH 公钥复制到剪贴板。

   如果您的 SSH 公钥文件与示例代码不同，请修改文件名以匹配您当前的设置。 在复制密钥时，请勿添加任何新行或空格。

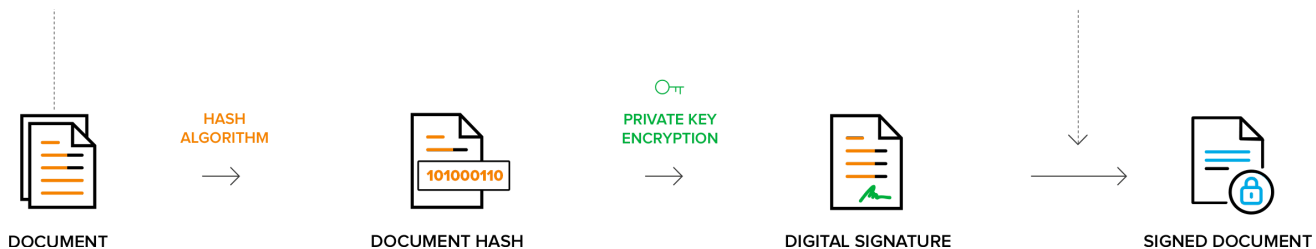
```
$ cat ~/.ssh/id_ed25519.pub
# Then select and copy the contents of the id_ed25519.pub file
# displayed in the terminal to your clipboard
```

So what is Ed25519, is it safe? 🧐

# Digital signature

suppose Alice wants to send a message to Bob, and they both want to find a way to check the authenticity and integrity of the message. digital signature is a technology that works in this way.

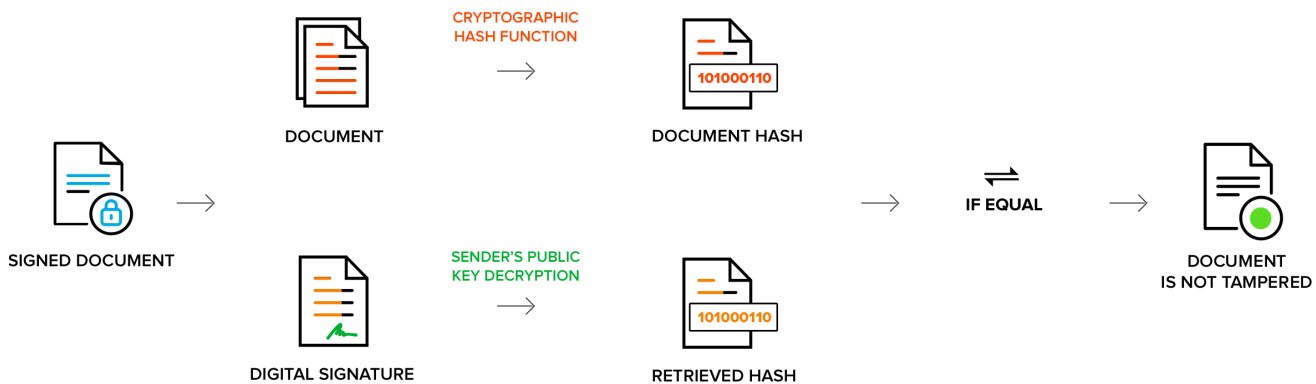
For Alice, she can using a mathematical algorithm to generate two keys: a **public key** and a **private key**. When she signs a document, a cryptographic hash is generated for the document. That cryptographic hash is then encrypted using the the **private key**, which is stored in a secure HSM box



# Digital signature

suppose Alice wants to send a message to Bob, and they both want to find a way to check the authenticity and integrity of the message. digital signature is a technology that works in this way.

Bob can decrypt the encrypted hash with the sender's **public key** certificate. A cryptographic hash is again generated on the recipient's end. Both cryptographic hashes are compared to check its authenticity. If they match, the document hasn't been tampered with and is considered valid



# Digital signature

there are several way to generate digital signatures, RSA and ECC are two very representative methods.

## RSA vs ECC: Side by Side Comparison

| RSA | ECC |
| --- | --- |
| A well-established method of public-key cryptography. | A newer public-key cryptography method compared to RSA. |
| Works on the principle of the prime factorization method. | Works on the mathematical representation of elliptic curves. |
| RSA can run faster than ECC thanks to its simplicity. | ECC requires bit more time as it's complex in nature. |
| RSA has been found vulnerable and is heading towards the end of its tenure. | ECC is more secure than RSA and is in its adaptive phase. Its usage is expected to scale up in the near future. |
| RSA requires much bigger key lengths to implement encryption. | ECC requires much shorter key lengths compared to RSA. |

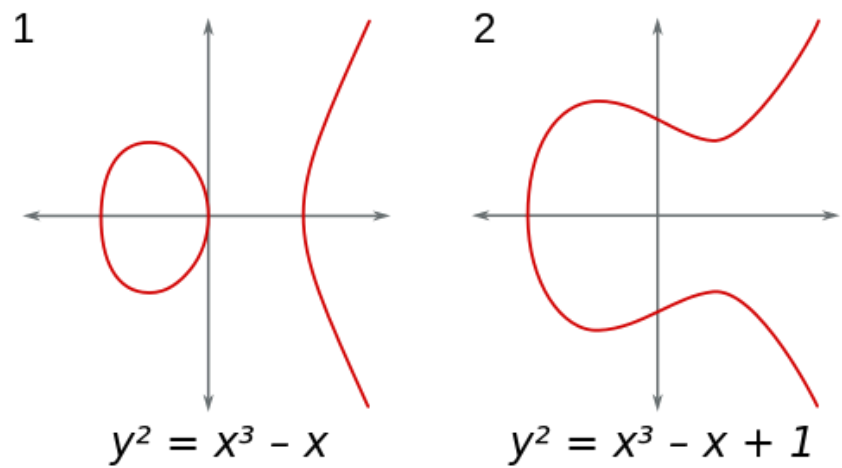
# Elliptic curve

first, let's take a look at what an elliptic curve is.

A elliptic curve is a plane algebraic curve which consists of solutions $(x, y)$ for

$$y^3 = x^3 + ax + b$$

and an infinity point $\mathcal{O}$, where $a, b$ satisfy $4a^3 + 27b^2 \neq 0$

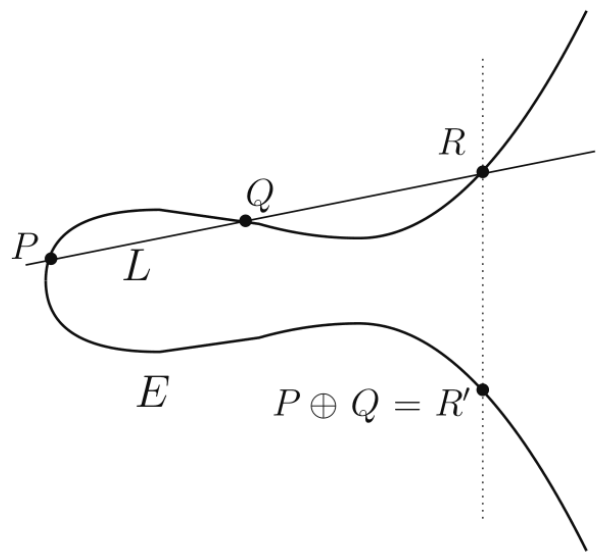


# Elliptic curve

we can define "addition" in elliptic curve.

Suppose we have two points $P, Q$, then the result of $P \oplus Q$ is obtained as follows:

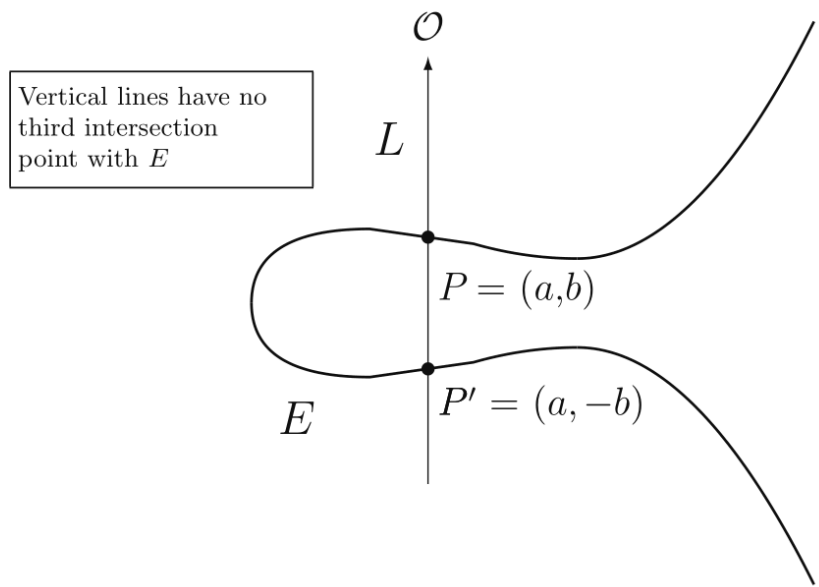
1. Connecting $P$ and $Q$ forms a straight line, which intersects the curve at point $R$
2. The point $R'$ symmetric to $R$ with respect to the x-axis is the result



# Elliptic curve

we can also define the infinity point in elliptic curve

So far, our definition of addition is not yet well-defined. It is easy to see that not all pairs of points can be added. Sometimes, the line passing through P and Q intersects the curve at only two points. In this case, we can define the result of this situation as the point at infinity.



# Elliptic curve

to apply elliptic curves in cryptography, we need a discrete definition.

Suppose $p$ is a prime number, then the prime field $\mathbb{F}_p$ is defined as a field consisting of $p$ elements, where addition and multiplication operations are performed modulo $p$.

So the elliptic curve over $\mathbb{F}_p$ is given by the equation

$$y^3 = x^3 + ax + b(mode\ p)$$

Its set of points is:

$$E(\mathbb{F}_p) = \{(x, y) : x, y \in \mathbb{F}_p, y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

where all operations are performed modulo p

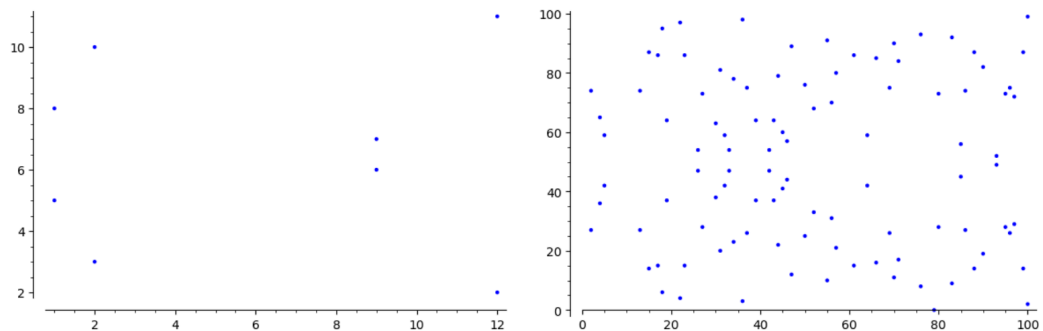
# Elliptic curve

let's examine the number of elements on the elliptic curve.

To find the set of points on an elliptic curve over a finite field, we can enumerate x and then look for y that satisfies the equation.

According to *Hasse's theorem on elliptic curves*, we can obtain a rough estimate.

$$|card\ E(\mathbb{F}_p) - (p+1)| \leq 2\sqrt{p}$$




When a=3, b=8,the distribution of points is like above. The left diagram is $\mathbb{F}_{13}$ , the right is $\mathbb{F}_{101}$

# Elliptic-curve cryptography

now let's apply elliptic curves in cryptography.

Public-key cryptography is based on the intractability of certain mathematical problems. This problem in the elliptic-curve is *Elliptic Curve Discrete Logarithm Problem*(ECDLP)

Let $E$ be an elliptic curve over the finite field $\mathbb{F}_p$, and let $P, Q$ be two point on the epliptic curve $E$. The Elliptic Curve Discrete Logarithm Problem is defined as finding an integer $n$ that $Q = nP$. We denote

$$n = log_p(Q)$$

and call $n$ the **elliptic curve discrete logarithm** of $Q$ with respect to base $P$

# Elliptic-curve cryptography

example: Schnorr Signature

In the situation of digital signatures, the signer should avoid disclosing their private key

Note that if signer has a information $nP$, he can apply a linear transformation to change $n$ to $(en+k)$. And calulate the discrete logarithm of $(en+k)P$ is easy to singer(absolutely, it is $en+k$). So we can propose a digital signatures method

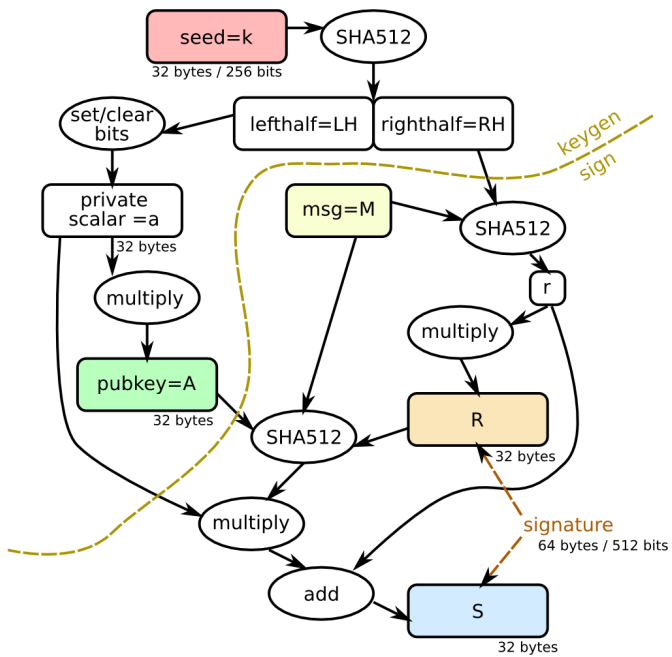
Suppose signer Alice keep **private key** $n$ and publicize **public key** $Q=nP$($P$ is base point), then for recipient Bob, the signing and verification process is as follows

1. Alice informs the Bob that the message to be signed is $m$
2. Alice randomly generates $k$ and calculates $R=kP$
3. Alice use a hash function to calculate $e=H(R_x||m)$ ($||$ means concatenation)
4. Alice calulates $s=(k+en)mod\ p$, and use $(e,s)$ as signature
5. Bob calulates $r_v=sP-eQ$, and then calculates $e_v=H(r_v||m)$
6. If $e_v=e$ then the signature is verified

# Ed25519

finally, let's focus on the Ed25519

Ed25519 is the *Edwards-curve Digital Signature Algorithm*(EdDSA) signature scheme using SHA-512 (SHA-2) and Curve25519. And EdDSA is a digital signature scheme using a variant of Schnorr signature based on *twisted Edwards curves*



# Ed25519

twisted Edwards curves

A twisted Edwards curve over a field $\mathbb{F}$ is defined as above

$$ax^2 + y^2 = 1 + dx^2y^2$$

where $a, d$ are distinct non-zero elements of $\mathbb{F}$

For Curve25519, $a = -1, d = -\frac{121665}{121666}$, so Curve25519 is

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

over field $\mathbb{F}_p$ where $p = 2^{255} - 19$

It is **birationally equivalent** to an elliptic curve

$$y^2 = x^3 + 486662x^2 + x$$

# Ed25519

encoding

For point at Curve25519, if we konw $x$, we can calculate two different $y$. So, if we want to efficiently transmit information, we need to encode the point's coordinates into a 256-bit string.

First, encode the **y-coordinate** as a little-endian string of 32 octets. The most significant bit of the final octet is always zero due to the cleverly chosen $p$. To form the encoding of the point, copy the least significant bit of the x-coordinate **to the most significant bit of the final octet**.

Then we get the coordinates's code with **32 bytes(256 bits)**, where 255 bits represent y-coordinate, and 1 bit represents sign bit

# Ed25519

key generation

The **private key** $k$ is 32 octets (256 bits) of cryptographically secure random data.

We use *SHA-512* hash the $k$ to generate a 512 bits $s = H(k) = (h_0, h_1, ..., h_{511})$. Then use the left 256 bits to generate **public key A**:

$$A = s \cdot B$$

where $B$ is the curve's base point, which is $(15112221349535400772501151409588531511454012693041857206046113283949847762202, 46316835694926478169428394003475163141307993866256225615783033603165251855960)$. And $s$ is calculated as above:

$$s = \left(2^{254} + \sum_{i=3}^{253} 2^i \cdot h^i\right)$$

**Notice**: We do not use all 256 bits. Instead, we clear bits $h_0$, $h_1$, $h_2$ and $h_{255}$ and set bit $h_{254}$. And this operation is a convention (RFC8032).

# Ed25519

sign

Calulate

$$r = H(h_{255}||...||h_{511}||M)$$

where $h_{255}||...||h_{511}$ is the right 256 bits of $s$, and $M$ is the message

And calulate

$$R = r \cdot B$$

Then calculate

$$S = (r + s \cdot H(R||A||M))\ (mod\ L)$$

where $L = 2^{252} + 27742317777372353535851937790883648493$

Use $(R, S) = (r \cdot B, r + s \cdot H(R||A||M))$ as the signature

# Ed25519

verify

For a recipient, he needs to confirm

$$(2^c \cdot S) \cdot B = 2^c \cdot R + (2^c \cdot H(R||A||M) \cdot A)$$

where $c = 3$

Because

$$\begin{aligned}(2^c \cdot S) \cdot B &= 2^c(r + H(R||A||M) \cdot s) \cdot B \\ &= 2^c \cdot r \cdot B + 2^c \cdot H(R||A||M) \cdot s \cdot B \\ &= 2^c \cdot R + (2^c \cdot H(R||A||M) \cdot A)\end{aligned}$$

Typically, if $S \geq L$, then the signature must be invalid

# Implementation

let's take tweetnacl-js as an example to see how to implement Ed25519

```
nacl.sign.detached.verify = function(msg, sig, publicKey) {
  checkArrayTypes(msg, sig, publicKey);
  if (sig.length !== crypto_sign_BYTES)
    throw new Error('bad signature size');
  if (publicKey.length !== crypto_sign_PUBLICKEYBYTES)
    throw new Error('bad public key size');
  var sm = new Uint8Array(crypto_sign_BYTES + msg.length);
  var m = new Uint8Array(crypto_sign_BYTES + msg.length);
  var i;
  for (i = 0; i < crypto_sign_BYTES; i++) sm[i] = sig[i];
  for (i = 0; i < msg.length; i++) sm[i+crypto_sign_BYTES] = msg[i];
  return (crypto_sign_open(m, sm, sm.length, publicKey) >= 0);
};
```

# Implementation

let's take tweetnacl-js as an example to see how to implement Ed25519

```
function crypto_sign_open(m, sm, n, pk) {
  ......
  if (unpackneg(q, pk)) return -1;             // decode A to q
  for (i = 0; i < n; i++) m[i] = sm[i];        // m = R||S||M
  for (i = 0; i < 32; i++) m[i+32] = pk[i];    // m = R||A||M
  crypto_hash(h, m, n);              // calculate hash value H
  reduce(h);                         // H modulo L
  scalarmult(p, q, h);               // calculate H*A

  scalarbase(q, sm.subarray(32));    // calculate S*B(sm.subarray(32) is S)
  add(p, q);                         // p = p + q
  pack(t, p);                        // encode p to t

  n -= 64;
  if (crypto_verify_32(sm, 0, t, 0)) {  // verify t is equal to sm[0:31](R)
    for (i = 0; i < n; i++) m[i] = 0;
    return -1;
  }

  for (i = 0; i < n; i++) m[i] = sm[i + 64];
  return n;
}
```

# Implementation

let's take tweetnacl-js as an example to see how to implement Ed25519

```
function crypto_sign_open(m, sm, n, pk) {
  ......
  if (unpackneg(q, pk)) return -1;             // decode A to q
  for (i = 0; i < n; i++) m[i] = sm[i];        // m = R||S||M
  for (i = 0; i < 32; i++) m[i+32] = pk[i];    // m = R||A||M
  crypto_hash(h, m, n);                // calculate hash value H
  reduce(h);                           // H modulo L
  scalarmult(p, q, h);                 // calculate H*A

  scalarbase(q, sm.subarray(32));   // calculate S*B(sm.subarray(32) is S)
  add(p, q);                           // p = p + q (why?)
  pack(t, p);                          // encode p to t

  n -= 64;
  if (crypto_verify_32(sm, 0, t, 0)) {  // verify t is equal to sm[0:31](R)
    for (i = 0; i < n; i++) m[i] = 0;
    return -1;
  }

  for (i = 0; i < n; i++) m[i] = sm[i + 64];
  return n;
}
```

# Implementation

let's take tweetnacl-js as an example to see how to implement Ed25519

```
function crypto_sign(sm, m, n, sk) { ........
  crypto_hash(d, sk, 32);                          // d = H(sk)
  d[0] &= 248;                                     // clear h_0, h_1, h_2
  d[31] &= 127;                                    // clear h_{255}
  d[31] |= 64;                                     // set h_{254}
  for (i = 0; i < n; i++) sm[64 + i] = m[i];       // sm = 64btyes || M
  for (i = 0; i < 32; i++) sm[32 + i] = d[32 + i]; // sm = 32btyes || h_{255}-h_{511} || M
  crypto_hash(r, sm.subarray(32), n+32);           // r = H(h_{255}||...||h_{511}||M)
  reduce(r);                                       // r = r mod L
  scalarbase(p, r);                                // p = r * B, which is R
  pack(sm, p);                                     // encode p to sm
  for (i = 32; i < 64; i++) sm[i] = sk[i];         // sm = p || sk || M
  crypto_hash(h, sm, n + 64);                      // h = H(R||A||M)
  reduce(h);                                       // h = h mod L
  for (i = 0; i < 64; i++) x[i] = 0;
  for (i = 0; i < 32; i++) x[i] = r[i];            // x[0:31] = r
  for (i = 0; i < 32; i++) { for (j = 0; j < 32; j++) {
      x[i+j] += h[i] * d[j];                       // calulate s * h
    }}
  modL(sm.subarray(32), x);                        // S = r - s * h, let sm = R || S
  return smlen;
}
```

# Implementation

let's take tweetnacl-js as an example to see how to implement Ed25519

```
function crypto_sign(sm, m, n, sk) { ........
  crypto_hash(d, sk, 32);                          // d = H(sk)
  d[0] &= 248;                                     // clear h_0, h_1, h_2
  d[31] &= 127;                                    // clear h_{255}
  d[31] |= 64;                                     // set h_{254}
  for (i = 0; i < n; i++) sm[64 + i] = m[i];       // sm = 64btyes || M
  for (i = 0; i < 32; i++) sm[32 + i] = d[32 + i]; // sm = 32btyes || h_{255}-h_{511} || M
  crypto_hash(r, sm.subarray(32), n+32);           // r = H(h_{255}||...||h_{511}||M)
  reduce(r);                                       // r = r mod L
  scalarbase(p, r);                                // p = r * B, which is R
  pack(sm, p);                                     // encode p to sm
  for (i = 32; i < 64; i++) sm[i] = sk[i];         // sm = p || sk || M
  crypto_hash(h, sm, n + 64);                      // h = H(R||A||M)
  reduce(h);                                       // h = h mod L
  for (i = 0; i < 64; i++) x[i] = 0;
  for (i = 0; i < 32; i++) x[i] = r[i];            // x[0:31] = r
  for (i = 0; i < 32; i++) { for (j = 0; j < 32; j++) {
      x[i+j] += h[i] * d[j];                       // calulate s * h
    }}
  modL(sm.subarray(32), x);                        // S = r - s * h, let sm = R || S
  return smlen;
}
```

# Implementation

let's take tweetnacl-js as an example to see how to implement Ed25519

```
function pack(r, p) {
  var tx = gf(), ty = gf(), zi = gf();
  inv25519(zi, p[2]);
  M(tx, p[0], zi);                  // p[0]: x-cordinate
  M(ty, p[1], zi);                  // p[1]: y-cordinate
  pack25519(r, ty);
  r[31] ^= par25519(tx) << 7;
}

function crypto_sign_keypair(pk, sk, seeded) {
  ........
  if (!seeded) randombytes(sk, 32);
  crypto_hash(d, sk, 32);
  d[0] &= 248;                      // clear h_0, h_1, h_2
  d[31] &= 127;                     // clear h_{255}
  d[31] |= 64;                      // set h_{254}
  scalarbase(p, d);
  pack(pk, p);                      // encode p to pk

  for (i = 0; i < 32; i++) sk[i+32] = pk[i];
  return 0;
}
```

# Implementation

in solana, Ed25519 can be utilized in multiple areas

## Terminology

The following terms are used throughout the Solana documentation and development ecosystem.

### account #

A record in the Solana ledger that either holds data or is an executable program.

Like an account at a traditional bank, a Solana account may hold funds called lamports. Like a file in Linux, it is addressable by a key, often referred to as a public key or pubkey.

The key may be one of:

- an ed25519 public key
- a program-derived account address (32byte value forced off the ed25519 curve)
- a hash of an ed25519 public key with a 32 character string

# Implementation

in solana, Ed25519 can be utilized in multiple areas

```
import { Keypair } from "@solana/web3.js";
import nacl from "tweetnacl";
import { decodeUTF8 } from "tweetnacl-util";

const keypair = Keypair.fromSecretKey(
  Uint8Array.from([
    174, 47, 154, 16, 202, 193, 206, 113, 199, 190, 53, 133, 169, 175, 31, 56,
    222, 53, 138, 189, 224, 216, 117, 173, 10, 149, 53, 45, 73, 251, 237, 246,
    15, 185, 186, 82, 177, 240, 148, 69, 241, 227, 167, 80, 141, 89, 240, 121,
    121, 35, 172, 247, 68, 251, 226, 218, 48, 63, 176, 109, 168, 89, 238, 135,
  ]),
);

const message = "The quick brown fox jumps over the lazy dog";
const messageBytes = decodeUTF8(message);

const signature = nacl.sign.detached(messageBytes, keypair.secretKey);
const result = nacl.sign.detached.verify(
  messageBytes,
  signature,
  keypair.publicKey.toBytes(),
);
```

# References


- https://www.zoho.com/sign/how-it-works/electronic-signature/digital-signature.html
- https://cheapsslsecurity.com/p/ecc-vs-rsa-comparing-ssl-tls-algorithms/
- https://www.ruanx.net/elliptic-curve/
- https://aandds.com/blog/eddsa.html
- https://courses.zjusec.com/2023/topic/crypto-lab3
- https://en.wikipedia.org/wiki/Elliptic-curve_cryptography
- https://datatracker.ietf.org/doc/html/rfc8032
- http://tweetnacl.cr.yp.to/tweetnacl-20140917.pdf
- https://solana.com/developers/cookbook

END