section 05: elliptic curve digital signature algorithm (ECDSA)

intro to signing algorithms

a singning algotithm has two functionalities:

1. **sign:**

   `sign(message, privkey) -> signature`

2. **verify:**

   `verify(message, signature, pubkey) -> bool`

   if the message was signed by the corresponding privkey AND the signature corresponds to the message, returns true. else returns false.

the signature is tied to the message itself. ie: you can't change the data in the message and still have a valid signature. when the signature is being verified, you compare it with the original message and the pubkey (see `verify` above).

think of it like a check. you have to sign *something*, it doesn't exist on the platonic world of forms.

ECDSA

signature generation algorithm

let's implement `sign()`. here's the steps to implement it:

1. calculate `z = HASH(m)`. in bitcoin's case, `HASH(m) = SHA256(SHA256(m))`;
2. pick a random number in the interval [1, n-1]. let's define this number as `k`. (ie: a nonce);
3. calculate the EC point `R`: `(Rx, Ry) = k * G`;
4. let `r = Rx mod n`. if `r = 0`, go back to step 2;
5. let $s = (\frac{1}{k}(z + r * d_A)) \mod n$, where d_A is the private key. if `s = 0`, go back to step 2;
6. the signature is the pair `(r, s)`. `(r, -s mod n)` is also a valid signature.

important: the nonce (`k`) *MUST NOT* be reused in different signatures. if so, d_A can be derived, given two signatures using the same nonce for different, but known, messages.

signature verification algorithm

let's implement `verify()`. here's the steps to implement it:

1. check that `r` and `s` are smaller than n;

2. calculate `z = SHA256(SHA256(m))`;
3. calculate `u = (z * s^-1) mod n`;
4. calculate `v = (r * s^-1) mod n`;
5. calculate the point `R = u * G + v * Pubkey_Point`;
6. if r is equal to Rx, then the signature is valid.

additive and multiplicative inverses on finite fields

when we wanted to find the other “conjugate point” on the EC, we took the additive inverse, via this operation:

`y2 = p - y1`,

where `p` is the field size.

when we do this, this will hold:

`y1 + y2 % p = 0` (zero is the identity of addition).

cool, but how do we get `1/k`, the multiplicative inverse?

we are trying to find the value that makes this statement true:

`(y1 * y2) % p = 1` (one is the identity of multiplication)

we want `1/k` such that `k * 1/k % n = 1`:

we can use python’s `pow()` function for this:

`k1 = pow(k, -1, n)`

distinguished encoding rules (DER) format

signatures made with `openssl` are encoded with DER. the `openssl` dependency was dropped long ago, and this encoding scheme was deprecated on taproot signatures. this signature contains 2 integers, that are either 32 or 33 bytes long.

it uses TLV (type-lenght-value). let’s take the transaction with id `14bec8ddd0624ba15a02e27ddfc8d0e98e4b3ef54f099330c6f7a47ce3861ffe` as an example. in it’s witness section, it has these two values:

```
// DER encoded signature
304402206572f867ff2e14fedb82996fcb02972799e6c1eb29fe1264a804b15379c76eb
2022016ebe63d732bc45102d8cbefa37558f27fa830df7edf2f09254dccf44f6e55e801

// compressed pubkey
034fbfee1786927128c1b0b8864268cb91463ce85e19f67def6569df19bfc6ecaa
```

let’s break down the signature. the first byte is the *type byte*, the second byte is the *lenght byte*, and then you have the data itself:

```
30 -> type byte [1]
44 -> lenght byte
    data:
    02 -> type byte
    20 -> lenght byte
    data (r):
        6572f867ff2e14fedb82996fcb02972799e6c1eb29fe1264a804b15379c76eb2

    02 -> type byte
    20 -> lenght byte
    data (s):
        16ebe63d732bc45102d8cbefa37558f27fa830df7edf2f09254dccf44f6e55e8

01 -> sighash flag [2]

[1]: the value 30 means it a compound value;
ie: (r, s), so we unwrap another TLV.

[2]: the sighash determines what the signature
applies to:
    0x01: ALL inputs, ALL outputs (ALL)
    0x02: ALL inputs, NO outputs (NONE)
    0x03: ALL inputs, ONLY ONE output with the same
          index as the signed input (SINGLE)
```