Lecture 12 - Elliptic Curve Digital Signatures

Happy Valentines Day!

Feb 14 2022

Every transaction has to be digitally signed to verify that the author of the transaction (the spender) is valid. Both Bitcoin and Ethereum use EC cryptography to do this. Specifically the ECDSA or Elliptic Curve Digital Signature Algorithm to do this.

The difference between encryption and digital signatures is that when you encrypt a message - you are protecting the content from others. With public/private systems you are also getting some form of assurance that the only person reading the message is the intended recipient.

With a digital signature you are not protecting the message content - you are providing proof that the originator is really the person that sent the message and that the message has not been tampered with.

The “signature” requires the private key. The values passed in the signature will not reveal the private key, but can be used to derive the public key. This allows a receiver to check that the correct message sender.

Bitcoin uses spec256k1 as the “curve” with a standard starting point.

Signatures and Validation

So... there are a bunch of uses of EC but the use that we want is to **sign** content to prove that the person holding the private key `N` is who they claim to be. In Bitcoin and Ethereum (and our homework) we are using this to verify transactions - this keeps other people from opening the UTxO’s that belong to you.

In our examples `p` the (Order(n)) size of our examples is 19. It has to be relatively prime and is a prime number in this case.

Our starting point, `P` is `(11,7)`. In the real Ed25519 signature system (Based on Curve25519) the modulo value is the `2**255-19` and the base point is

x=15112221349535400772501151409588531511454012693041857206046113283949847762202

y=463168356949264781694283940034751631413079938662562256157830336031652518559 60

So big numbers...

Curve25519 also defines the values that give us the exact curve (the ‘a’ and ‘b’ values).

Signing Algorithm

Input is our public and private keys. You have to have the private key to sign, you have to have the public key to verify. The agreement on curve, starting point and order need to be in place to start the this.

Output is a pair `{r,s}` that act as a verifiable signature. In our special case this is actually `{r,s,v}` where v is a 0, 1 (we will get back to `v` in a moment)

1. take the message and calculate the hash of the message. This is a 32 byte Sha256 or Keccak hash of the message. `h = hash(msg)`.

2. Generate securely a random number `k` in the range `[1..n-1]`. The value must be from 1 to `2**256` in our case. So a 32 byte random value.
3. Calculate the random point `R = k * P` and take its x-coordinate: `r = R.x`
4. Calculate the signature proof: $s = k^{-1} * (h + r * privKey) \pmod p$
5. Return the `signature {r, s}`.

Verification Algorithm

The algorithm to verify a ECDSA signature takes as input the signed message msg + the signature {r, s} produced from the signing algorithm + the public key pubKey, corresponding to the signer's private key. The output is boolean value: valid or invalid signature. The ECDSA signature verify algorithm works as follows (with minor simplifications):

1. Calculate the message hash, with the same cryptographic hash function used during the signing: `h = hash(msg)`. This can also be achieved by passing both the message, the hash and then re-hashing the message and comparing the generated hash with the passed hash. We will do it this way.
2. Calculate the modular inverse of the signature proof: $s1 = s^{-1} \pmod p$
3. Recover the random point used during the signing: `R' = (h * s1) * G + (r * s1) * pubKey`
4. Take from `R'` its x-coordinate: `r' = R'.x`
5. Calculate the signature validation result by comparing whether `r' == r`

The idea behind signature verification is to recover the point `R'` using the public key and check whether it is same point `R`, generated randomly when the message was signed.

This scheme allows for the recovery of the public key from the signature and the message. This means that a receiver of a signed message will not need to keep a huge table of all public keys around. The disadvantage is that you have to check the recovered key is the correct public key for the account. In our system (and Ethereum) are accounts are the first 24 bytes of the 32 byte public key. So when we recover the public key we chop it to 24 bytes and compare it to the account number.

Also when we recover a public key we can get 4 different answers. This is what the `v` value is used for.

Reading

A simple ERC-20 Contract: https://www.toptal.com/ethereum/create-erc20-token-tutorial

Zepplin ERC20: https://forum.openzeppelin.com/t/how-to-implement-erc20-supply-mechanisms/226

```
contract ERC20FixedSupply is ERC20 {
    constructor() public {
        _mint(msg.sender, 1000);
    }
}
```

Stocks, ICOs, Bonds, etc...

Terms with explanation:

1. Cash - this is what you buy your pizza with.
2. what is a P&L - financial statement of income and expenses that shows if the company is making or loosing money.

3. What is Inflation?
4. What is a Derivative
5. Dividends - Payment of profit to investors as “income” or a reward for owning the stock.
6. What is the “Yield”
7. ICOs (Initial Coin Offering)
8. Cost of “going public”
9. Price to Earnings Radio - a P/E ratio is the cost of share of the company divided by Earnings of that particular share of a company.
10. High speed trading
11. What is a basis point (BPS), Example 50 BSP = 0.5%
12. Stock Buy Back
13. Index Fund
14. Insider Trading
15. “Consumer Token Sale”
16. Junk Bonds - non investment grade bond. Bonds are rated for default risk.
17. Mutual Fund
18. Asset Allocation
19. Expense Ratio for a Mutual Fund
20. Prospectus
21. Pro-Forma
22. KYI - Know Your Investor (See SEC 506(d))
23. KYC - Know your customer
24. Certified Investor
25. Going public - Make a public offering. Cost etc.
26. Money Laundering