DIGITAL SIGNATURES

JOHN NEWBERY

@jfnewbery

github.com/jnewbery

ABOUT ME

Live in New York

Work for Chaincode Labs

Contribute to Bitcoin Core

github.com/jnewbery

DIGITAL SIGNATURES

- Electronic coins & digital signatures
- Finite Fields
- Elliptic Curves
- Schnorr Signatures
- ECDSA
- Further Reading

CAVEAT AUDITOR!

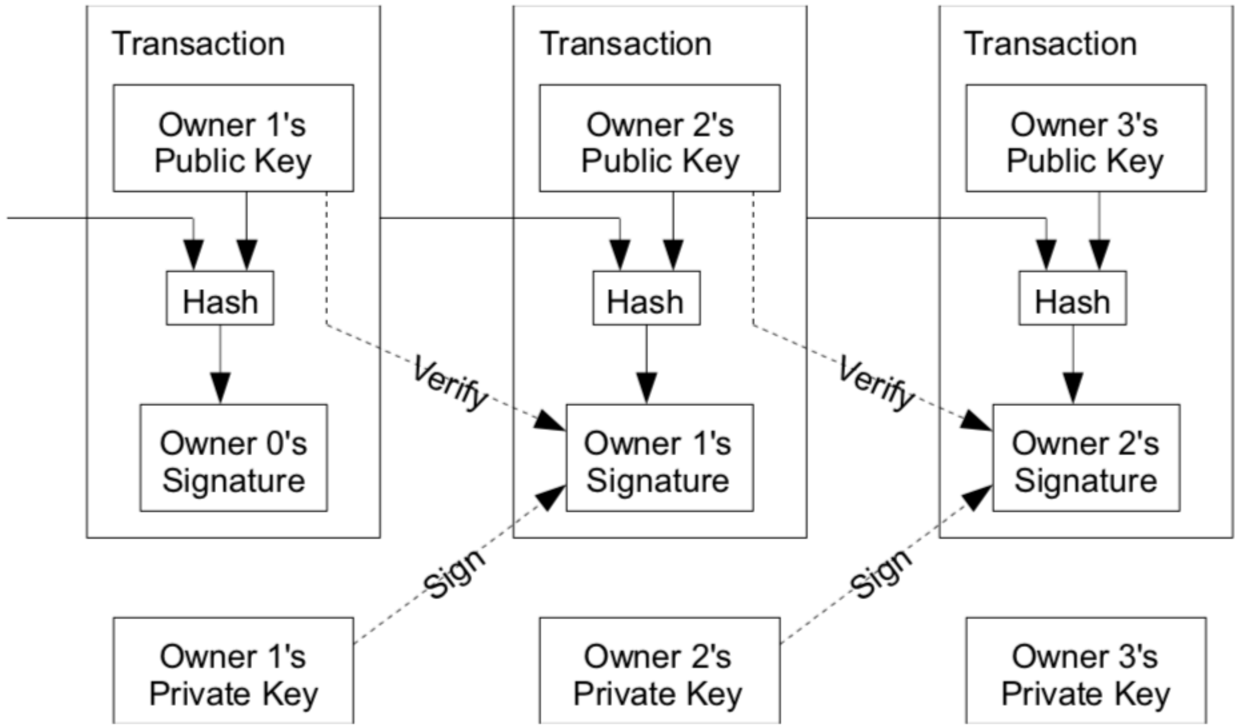
- I am not a cryptologist!
- This is only an overview - no formal proofs
- I will use some terms loosely eg:
  - *zero knowledge* instead of *honest verifier zero knowledge*
  - *proof* instead of *argument*

ELECTRONIC COINS

ELECTRONIC COINS

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.



SENDING A COIN (SIMPLIFIED)

- A transaction consists of:
  - one or more transaction inputs (txins), which contain:
    - a reference to the transaction output (txout) that is being spent
    - a digital signature proving that the owner of the private key authorised the transaction
  - one or more transactions outputs (txouts), which contain:
    - the amount
    - the public key of the recipient of the txout

VERIFYING A TRANSACTION (SIMPLIFIED)

- All Bitcoin nodes verify all transactions:
  - Check that each txin points to an unspent txout
  - Check that the total amount in the txouts does not exceed the total amount from the txins
  - Check that each txin contains a valid signature for the public key from the txout referenced

DIGITAL SIGNATURES

- *Digital signatures* are used to transfer ownership of coins
- A digital signature proves that the owner of the coin authorised the transfer:
    - only someone with the private key can sign the transaction (authentication)
    - no-one can change the transaction after it has been signed (integrity)

WHAT IS A DIGITAL SIGNATURE?

- Digital signatures make use of asymmetric cryptography
- The user has a public key (which is known to everyone) and a corresponding private key (which is kept secret)
- Only someone with the private key can create a valid signature over a message
- Anyone with the public key and message can verify that the signature is valid

DIGITAL SIGNATURES AND BITCOIN

- Bitcoin uses *Eliptic Curve Digital Signature Algorithm (ECDSA)* over the *secp256k1* curve
- A better digital signature algorithm is the *Schnorr signature algorithm*
- In the future, the Bitcoin protocol may be extended to allow Schnorr signatures

THE DISCRETE LOG PROBLEM

- ECDSA is an application of the *discrete log problem*
- In some systems it is easy to 'multiply' but difficult to 'divide'
- Discrete logs are defined for *cyclic groups* with a generator G. The problem is:
  - for a given H in the group, what is the scalar x such that $xG = H$
- Bitcoin uses the group of points on the *elliptic curve secp256k1* defined over a finite field of integers

+	0	1	a	b
0	0	1	a	b
1	1	0	b	a
a	a	b	0	1
b	b	a	1	0

*	0	1	a	b
0	0	0	0	0
1	0	1	a	b
a	0	a	b	1
b	0	b	1	a

FINITE FIELDS

GROUP

- A *group* is a set of objects along with a *binary operator* +
- The binary operator has the following properties:
  - closure: $\forall a, b \in G, a + b \in G$
  - identity: $\exists 0 \in G \mid 0 + a = a + 0 = a \ \forall a \in G$
  - inverse: $\forall a \in G, \exists (-a) \mid a + (-a) = (-a) + a = 0$
  - associativity: $\forall a, b, c \in G, (a + b) + c = a + (b + c)$
- Some groups (called commutative/Abelian groups) also have:
  - commutativity: $\forall a, b \in G, a + b = b + a$

CYCLIC GROUP

- A group is cyclic if there is a generator element:

$$\exists g \mid \forall a \in G, \exists n \mid a = g + g + g + \ldots (n \ \ times)$$

- The integers modulo p for any prime p is a cyclic group:

$$Z/pZ = \{0,1,2,\ldots,p-1\}$$

FIELD

- A field is a commutative group with a second *binary operator* x
- The second binary operator is also closed, has an identity, has inverses (except for zero) and is associative and commutative
- The binary operations are also distributive:

$$\forall a, b, c \in G, a \times (b + c) = (a \times b) + (a \times c)$$

- We can add, subtract, multiply and divide over a field

EXAMPLE FIELDS

- The real numbers, with addition and multiplication defined as normal (infinite)
- The rational numbers, with addition and multiplication defined as normal (infinite)
- The integers from 0 to (n -1), with addition and multiplication defined modulo n (finite)

THE FINITE FIELD F_P

- We use the finite field $F_P = \{0,1,2,\ldots,p-1\}$
- eg $F_{13} = \{0,1,2,\ldots,12\}$

$$4 + 5 = 9$$

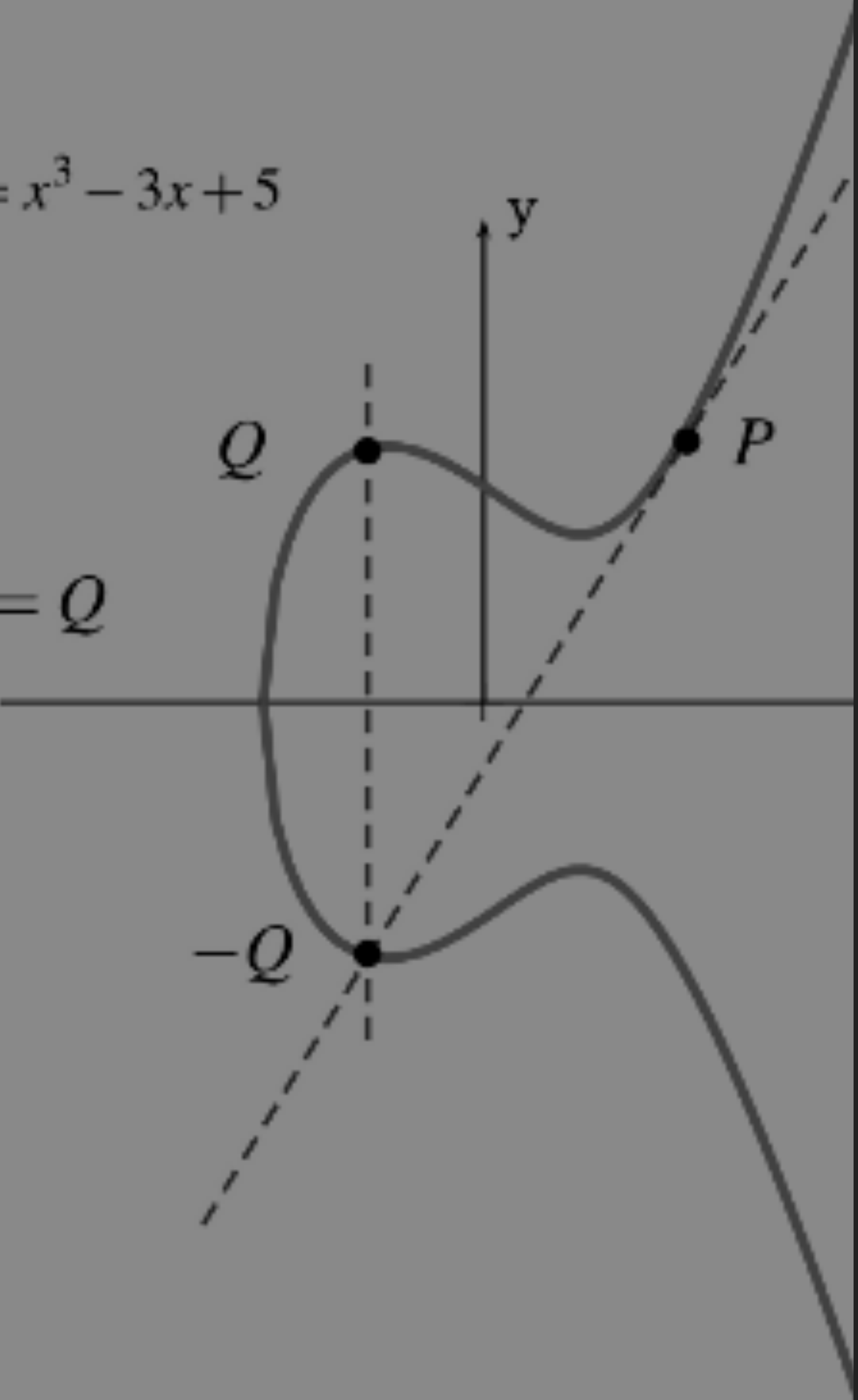
$$8 + 9 = 17 = 4 \ (mod\ 13)$$

$$4 - 8 = -4 = 9 \ (mod\ 13)$$

$$5 \times 3 = 15 = 2 \ (mod\ 13)$$

$$5 \div 3 = 5 \times \frac{1}{3} = 5 \times 9 = 45 = 6 \ (mod\ 13)$$

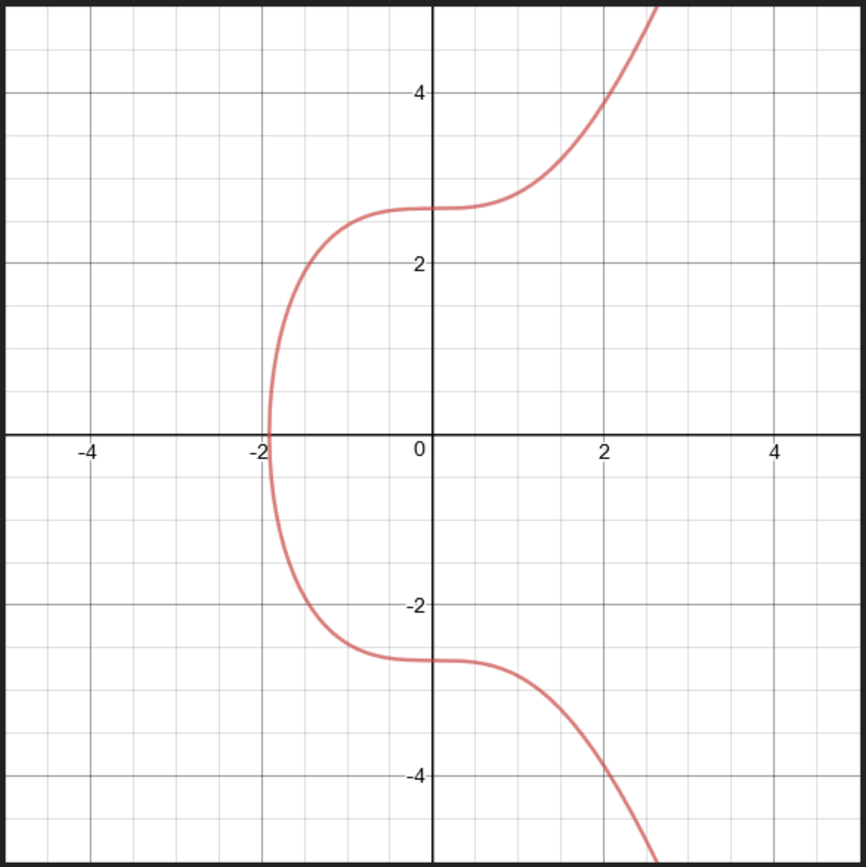
$$5^3 = 125 = 8 \ (mod\ 13)$$



ELLIPTIC CURVES

ELLIPTIC CURVES

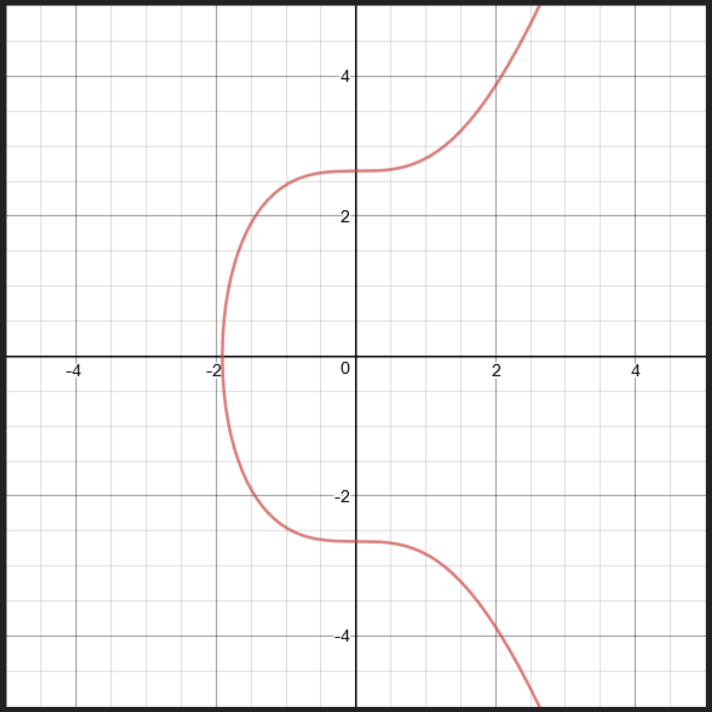
- An elliptic curve is a curve of the form: $y^2 = x^3 + ax + b$
- In Bitcoin, we use the sec256k1 curve: $y^2 = x^3 + 7$
- (ie a=0 and b=7)



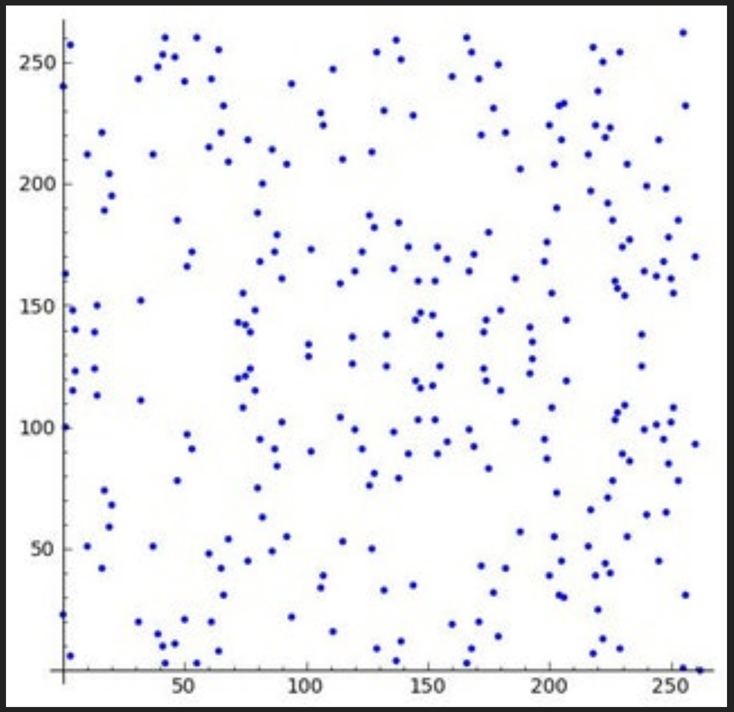
ELLIPTIC CURVES OVER A FINITE FIELD

- Instead of defining secp256k1 over the reals, we define it over a finite field of integers mod p
- p is $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

OVER REALS

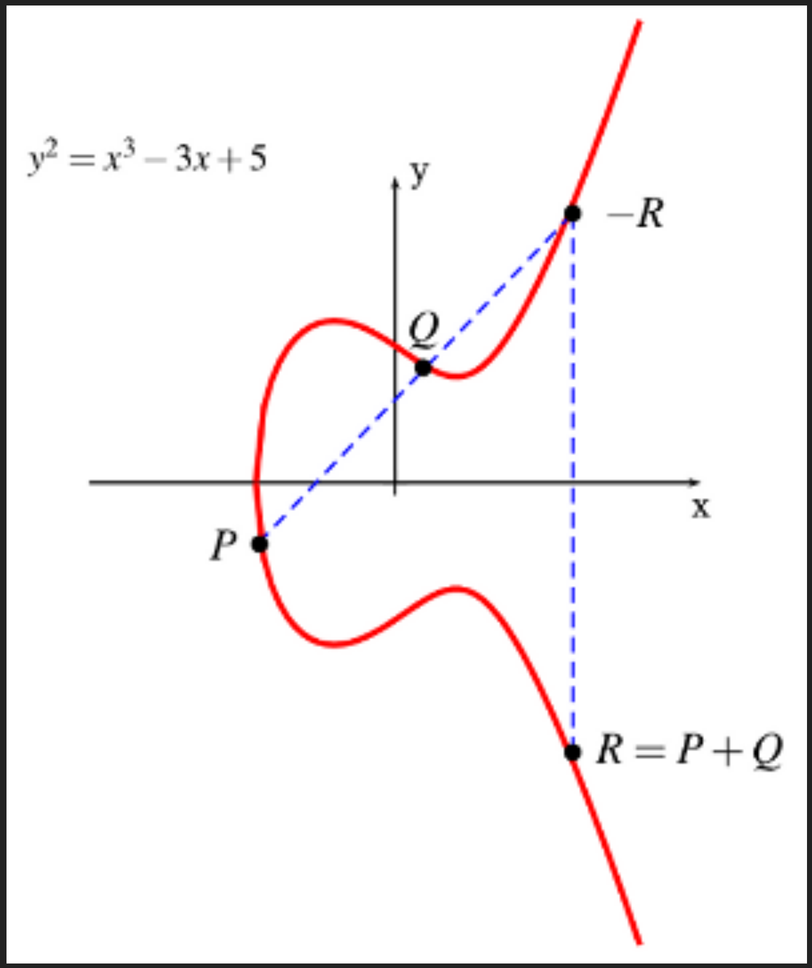


OVER F_P



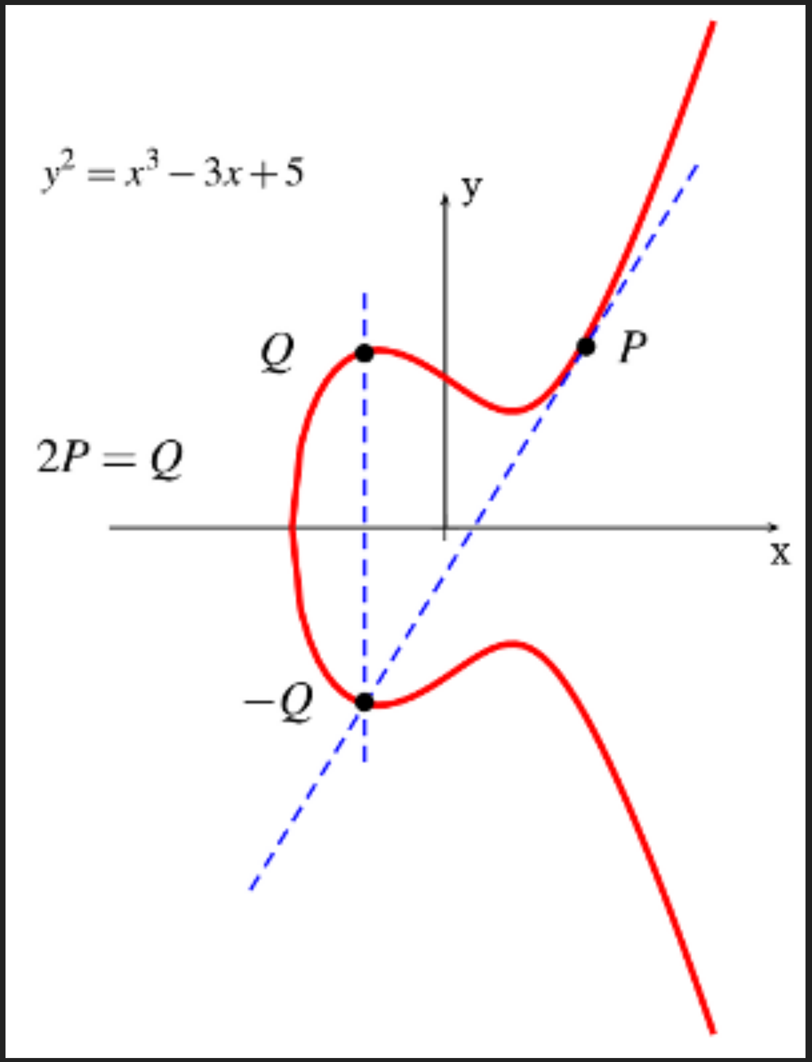
DEFINING A GROUP OPERATION FOR THE ELLIPTIC CURVE

- We can define a binary operation + for the elliptic curve
- To add two points:
  - take the line meeting the two points
  - find where the line intersects the curve again
  - reflect through the x axis



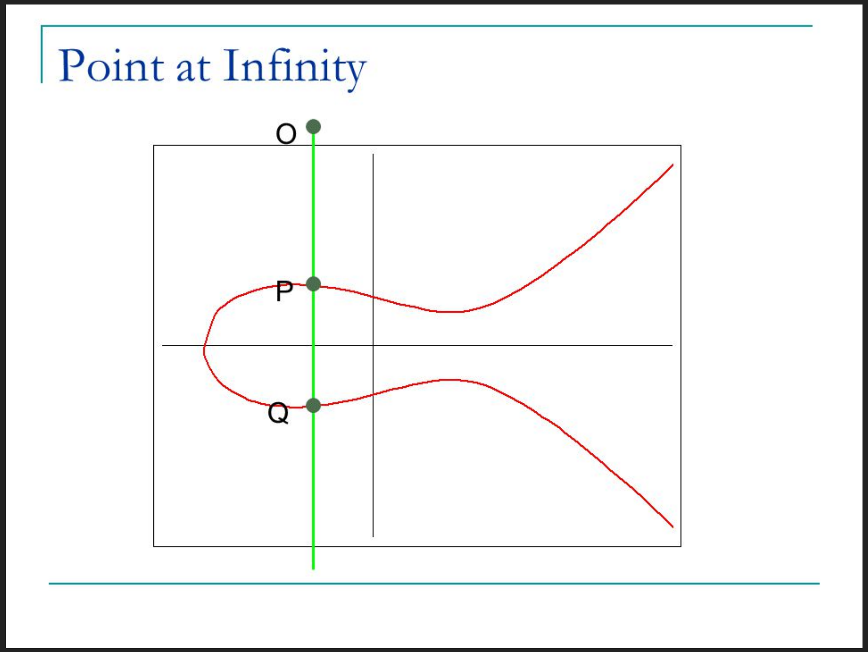
DEFINING A GROUP OPERATION FOR THE ELLIPTIC CURVE

- To double a point:
  - take the tangent at that point
  - find where the tangent meets the curve again
  - reflect through the x axis



DEFINING A GROUP OPERATION FOR THE ELLIPTIC CURVE

- A point's inverse is the reflection in the x axis
- Adding a point to its inverse yields our group identity: the 'point at infinity'
- Adding the point at infinity to any point P yields P

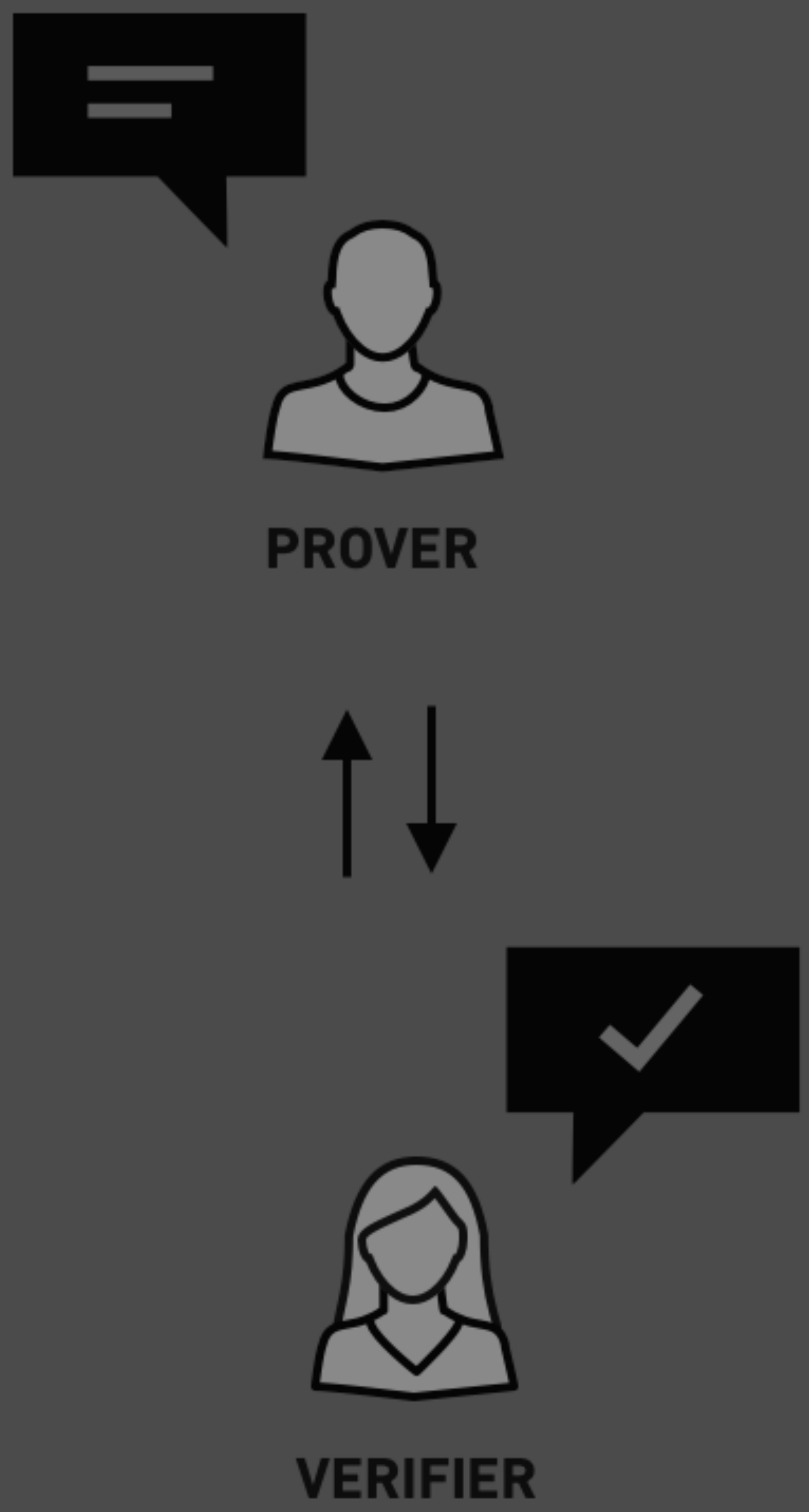


GENERATING A CYCLIC GROUP

- Take any point G on the curve
- Repeatedly add it to itself until you reach G again
- The set of points generated is a cyclic group
- For secp256k1, we use the generator point:
  G = (55066263022277343669578718895168534326250603453777594175500187360389116729240,
  32670510020758816978083085130507043184471273380659243275938904335757337482424)
- This is the group we use for our discrete log problem

DISCRETE LOG PROBLEM FOR AN ELIPTIC CURVE

- The private key is a scalar x, which is a 256 bit number in the range [0, … , n-1] where n is the order of the group
- The public key is a point P on the curve where $P = xG$
- It’s easy to go from x to P
- it’s *computationally difficult* to go from P to x



SCHNORR SIGNATURES

SCHNORR IDENTIFICATION PROTOCOL

- A prover can prove to a verifier that she knows the private key x corresponding to a public key P without revealing x
- The verifier learns nothing about x from the proof (except the fact that the prover knows x)
- This is called a *proof in zero knowledge*

ZERO-KNOWLEDGE PROOF

- A *zero-knowledge proof* requires three properties:
  - *Completeness* - the proof convinces the verifier
  - *Zero-knowledgness* - the proof doesn't leak information
  - *Soundness* - a proof can only be produced by a prover who knows the private key

SCHNORR IDENTIFICATION PROTOCOL - THE STEPS

- The identification protocol has 3 steps:
  - 1. commitment - the prover picks a *nonce* scalar k and commits to it by sending $K = kG$ to the verifier
  - 2. challenge - the verifier sends a *challenge* scalar e
  - 3. response - the prover sends the *response* scalar $s = k + ex$

SCHNORR IDENTIFICATION PROTOCOL - COMPLETENESS

- The verifier is convinced that the prover knows x if the identity holds:

$$\begin{aligned} sG &= kG + exG \\ &= K + eP \end{aligned}$$

- The verifier can do this because he knows s, G, K, e and P

SCHNORR IDENTIFICATION PROTOCOL - ZERO-KNOWLEDGENESS

- The *transcript* of the 3 step protocol is: (K, e, s)
- If the verifier colludes with the prover and tells her what e_{fake} is before she provides a K, then she can choose s_{fake} randomly and set $K_{fake} = s_{fake}G - e_{fake}P$
- The transcript $(K_{fake}, e_{fake}, s_{fake})$ is indistinguisable from a real transcript
- If we can *simulate* a fake proof transcript without knowledge of x, then it follows that a real proof transcript leaks no knowledge of x

SCHNORR IDENTIFICATION PROTOCOL - SOUNDNESS (1)

- If the prover can produce a proof reliably for any challenge e, she *must* know x
- Imagine being able to pause, fast-forward or rewind the prover's operation. The verifier could 'fork' the prover:
  1. wait for the prover's commitment K
  2. send challenge e_1 and receive response s_1
  3. rewind to the challenge step
  4. send challenge e_2 and receive response s_2

SCHNORR IDENTIFICATION PROTOCOL - SOUNDNESS (2)

- The verifier now has: $s_1 = k + e_1 x$ and $s_2 = k + e_2 x$
- The verifier can calculate $$x = \frac{s_1 - s_2}{e_1 - e_2}$$
- The verifier has extracted the private key x from the prover
- The prover therefore must have had the private key!
- If this doesn't convince you, imagine the prover 'forking' himself

NON-INTERACTIVE SCHNORR IDENTIFICATION PROTOCOL

- That the verifier's only role was to provide a 'random' challenge
- If we can replace the verifier with a *random oracle* that simply provides a random number *after* the commitment step, then we don't need a verifier
- We treat a hash function as a *random oracle*
- *After* has a special meaning - the prover can't know the output to a hash function before evaluating it
- This is called a Fiat-Shamir transform

NON-INTERACTIVE SCHNORR IDENTIFICATION PROTOCOL

- The identification protocol has 3 steps:
  1. The prover picks a *nonce* scalar k
  2. The prover calculates $e = H(kG)$
  3. The prover computes the scalar $s = k + ex$
- The proof is (s,e)
- Anyone can verify the proof by calculating $kG = sG - exG$ and verifying $e = H(kG)$

SIGNATURE OF KNOWLEDGE OVER A MESSAGE

- Since H is a random oracle and returns different values for different inputs, the prover can add extra inputs to H
- The result is a *signature of knowledge* over a message
- eg the prover can set $e = H(m \| kG)$
- The prover calculates s in the normal way: $s = k + ex$
- The verifier then calculates $kG = sG - exG$
and verifying that $e = H(m \| kG)$

ECDSA

ECDSA

- ECDSA is a different digital signature algorithm
- It also uses the DLP over elliptic curves
- ECDSA was developed (and later used in Bitcoin) because Schnorr signatures were encumbered by a patent
- There are several disadvantages compared to Schnorr:
  - Signatures are not linear (makes threshold and adaptor signatures much more difficult)
  - There is no security proof for ECDSA
  - ECDSA signatures are malleable

ECDSA SIGNING

- The prover signs a message m as follows:
  1. set z as the leftmost bits of $H(m)$
  2. pick a random nonce scalar k
  3. set $K = kG$ and r as the x coordinate of K
  4. set $s = k^{-1}(z + rx)$
- The signature is the pair (r, s)

ECDSA VERIFYING

- The signature can be verified as follows:
    1. set *z* as the leftmost bits of *H(m)*
    2. set $u = \frac{z}{s}$ and $v = \frac{r}{s}$
    3. if the x co-ordinate of $uG + vP$ is equal to *r*, then the signature is valid

FURTHER READING

FURTHER READING

- Borromean Ring Signatures, *Greg Maxwell and Andrew Poelstra*: https://github.com/Blockstream/borromean_paper
- Confidential Transactions and Bulletproofs, *Adam Gibson*: https://joinmarket.me/blog/blog/from-zero-knowledge-proofs-to-bulletproofs-paper/
- Zero-knowledge proofs, *Matthew Green*: https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/
- Schnorr Signatures, *Pieter Wuille*: https://www.youtube.com/watch?v=YSUVRj8iznU