Elliptic Curve Cryptography

ECEN 5033
Jan 24, 2019

Review: RSA

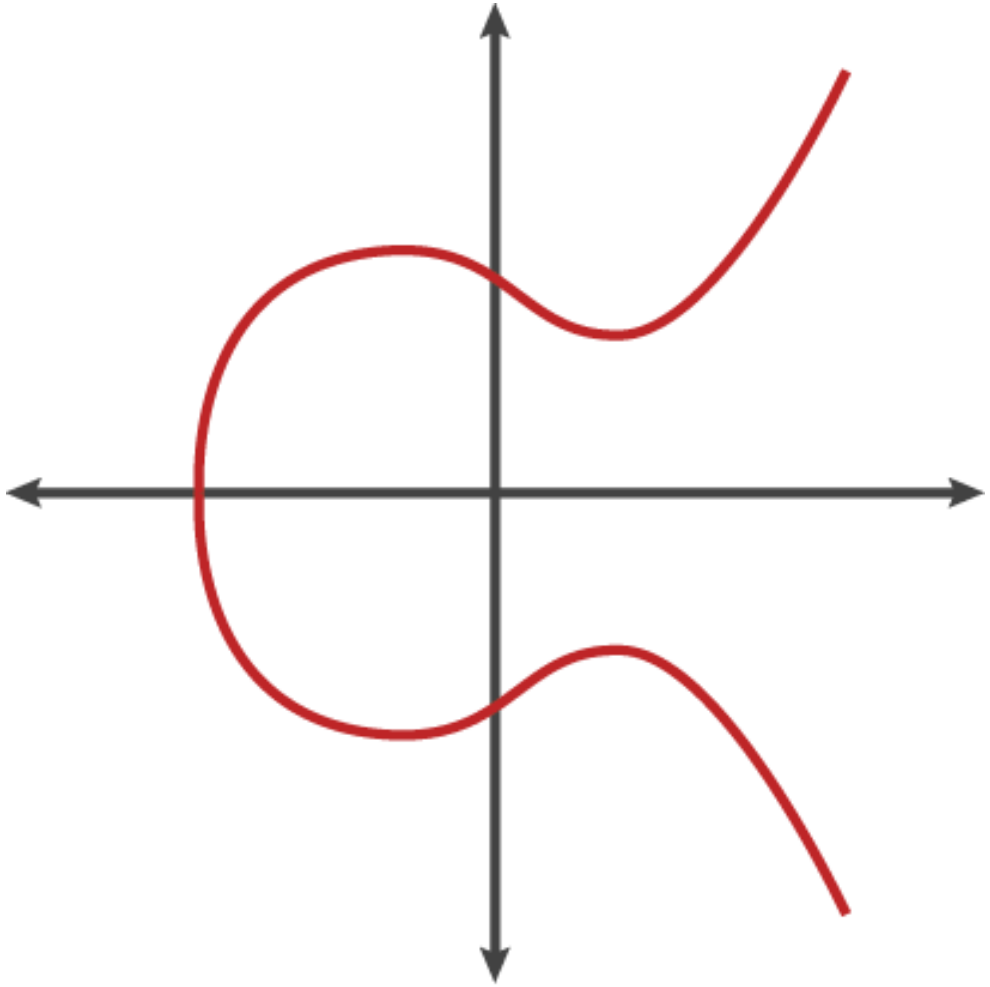
- **Key generation:**
  1. Pick large (say, 1024 bits) random primes **p** and **q**
  2. Compute **N** := **pq** (RSA uses multiplication mod **N**)
  3. Pick **e** to be relatively prime to (**p**-1)(**q**-1)
  4. Find **d** so that **ed** mod (**p**-1)(**q**-1) = 1
  5. Finally: **Public key** is (e,N)
     Private keyis (d,N)
- **To encrypt:** $E(\mathbf{x}) = \mathbf{x}^{\mathbf{e}} \bmod \mathbf{N}$
  To decrypt: $D(\mathbf{x}) = \mathbf{x}^{\mathbf{d}} \bmod \mathbf{N}$

Review: RSA

- Breaking RSA:
  - Factor N into p, q
  - Have to choose large enough p, q so that factoring is hard
    - But larger p, q means slower and larger signatures

Elliptic Curves

$$y^2 = x^3 + ax + b$$

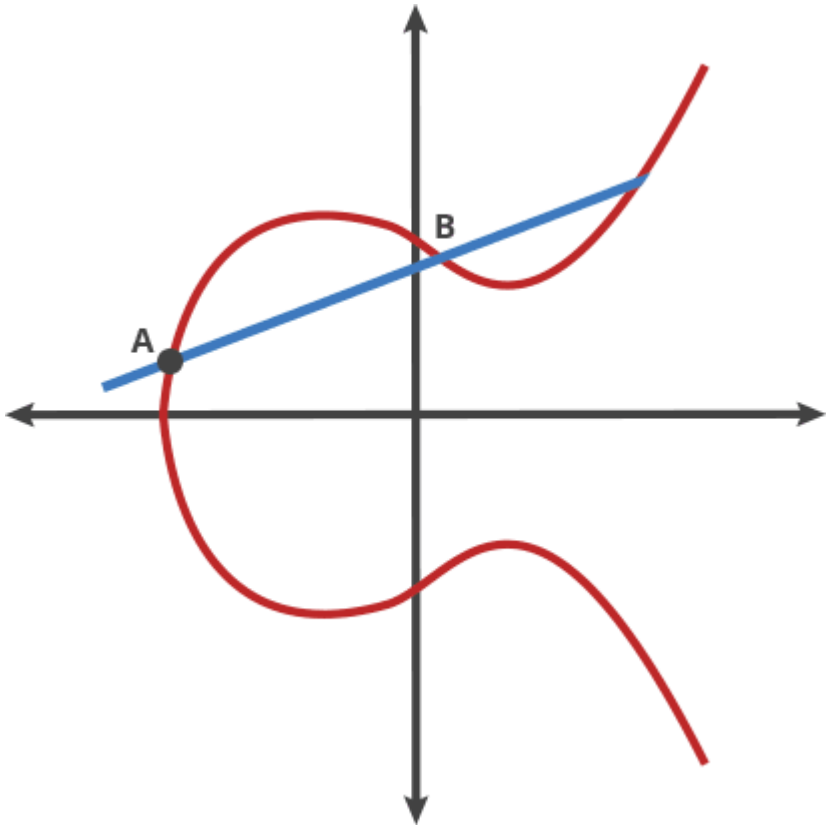


Adapted from:
https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

Point addition

$$y^2 = x^3 + ax + b$$

Can *add* points:
-Draw a line through both points, follow it until it hits the curve again, then reflect over the x-axis.
-Unique solution for any two points on the curve, gives a third point on the curve.
-Adding the same point (doubling): draw line tangent to the curve at the point



Adapted from:
https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

Point multiplication

- P + P = 2P
- P + P + P = 3P
- Repeat point addition n times:
  - nP
- Trapdoor function:
  - Easy to compute Q = dG given integer d and point G, but hard to compute d from Q and G
  - Like finding the initial state of a Billiard Ball game

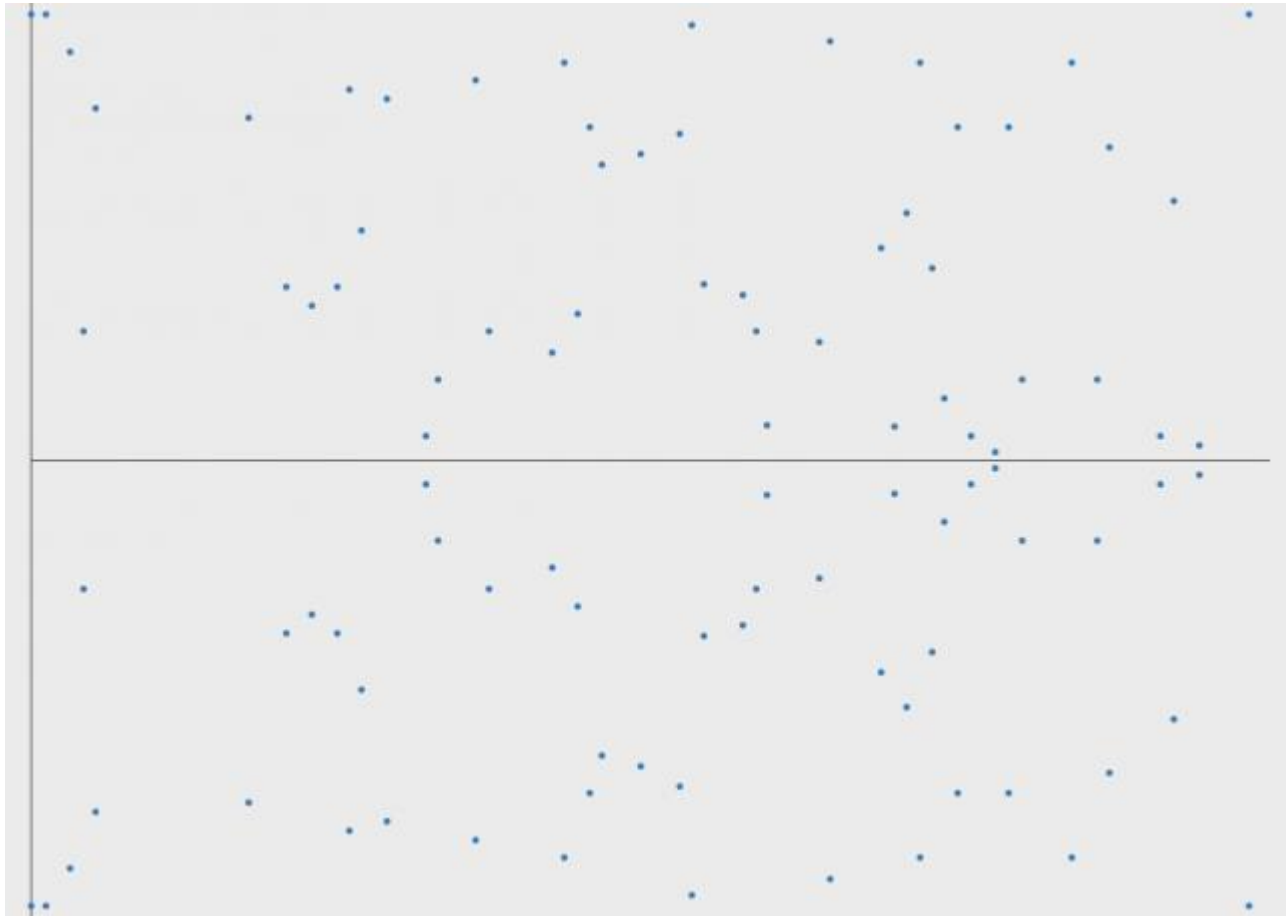
Elliptic Curve Cryptography

$$y^2 = x^3 + ax + b \textbf{ mod p}$$

Take only integer points, modulo a large prime
Looks pretty random…
But can still do point addition
(and: still symmetric about x-axis)

Adapted from:
https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

$$y^2 = x^3 - x + 1 \, mod \, 97$$



Elliptic Curve Cryptography

- As used in Bitcoin:
  - NIST Standardized curve named "secp256k1"
  - Koblitz curve: chosen to be efficient to compute on

- $y^2 = x^3 + ax + b$ **mod p**
- p = $2^{256} - 2^{32} - 977$
  - 115792089237316195423570985008687907853269984665640564039457584007908834671663
- a = 0, b = 7
- Base point G = (x, y) =

(550662630222773436695787188951685343262506034537775941755001873603891167292 40,
32670510020758816978083085130507043184471273380659243275938904335757337482424)

- secp256k1: $y^2 = x^3 + 7\ mod\ (2^{256} - 2^{32} - 977)$
- curve25519: $y^2 = x^3 + 486662x^2 + x\ mod\ (2^{255} - 19)$

How do we use point multiplication?

- Easy to compute Q = dG given integer d and point G
- Hard to compute d from Q and G

Diffie-Hellman key exchange over

- Easy to compute Q = dG given integer d and point G
- Hard to compute d from Q and G

- Alice and Bob agree on curve (e.g. secp256k1), and point G.
- Alice picks random d, computes dG (and sends dG to Bob)
- Bob picks random e, computes eG (and sends eG to Alice)

- Alice computes d(eG)
- Bob computes e(dG)
- Attacker that knows (eG) and (dG) can't compute edG!
  - Can't learn e or d from eG or dG, need at least one to compute edG

Signing using point multiplication (ECDSA)

- *Elliptic Curve Digital Signature Algorithm (ECDSA)*
- Given private key **d**, public key **dG**, message **m**
- **z** = H(**m**)
  - Some caveats about size of z vs group order **n** (size) of curve
  - We'll simply use SHA256(m) in project 1
- Generate random **k**
- Compute **kG**; set **r** = x-coordinate of that point
- **s** = k^{-1}(**z** + **rd**) mod **n**
- Signature is **(r, s)**

Verifying a signature (ECDSA)

- Given public key **Q**, message **m**, signature **(r, s)**
- **z** = H(**m**)
- $\mathbf{w} = \mathbf{s}^{-1} \bmod \mathbf{n}$
- $\mathbf{u}_1 = \mathbf{zw} \bmod \mathbf{n}$
- $\mathbf{u}_2 = \mathbf{rw} \bmod \mathbf{n}$
- Compute $\mathbf{P} = \mathbf{u}_1\mathbf{G} + \mathbf{u}_2\mathbf{Q}$
- Signature valid if x-coordinate of **P** is equal to **r**

Why does ECDSA work?

- $\mathbf{P} = \mathbf{u_1 G} + \mathbf{u_2 Q}$ - why does x-coordinate of **P** == **r**?

  $= \mathbf{zwG} + \mathbf{rwQ}$

  $= \mathbf{zs^{-1} G} + \mathbf{rs^{-1} Q}$

  $= \mathbf{zs^{-1} G} + \mathbf{rs^{-1} dG}$

  $= \mathbf{(zs^{-1} + rs^{-1} d)G}$

  $= \mathbf{s^{-1} (z+ rd)G}$

  $= \mathbf{k(z+rd)^{-1} (z+ rd)G}$

  $= \mathbf{kG}$ (which has x-coordinate = **r**)

ECDSA vs RSA

- ECDSA advantages over RSA
  - Faster (by about 20x)
  - Smaller public/private keys (256 bits vs 2048 bits)
  - Smaller signatures (512 bits vs 2048 bits)
- ECDSA disadvantages compared to RSA
  - Can't directly encrypt (but can use El Gamal's encryption algorithm)
  - Nonce reuse in signatures leaks the private key!!!

ECDSA nonce reuse

- Suppose Alice signs two messages (with hashes **z** and **z′**) with the same nonce **k**.
  - **z**, (**r**, **s**)
  - **z′**,(**r**, **s′**)
- Anyone can compute **k** from these signatures:
  - $s-s' = k^{-1}(z-z') \bmod n$
  - $k = (z-z') / (s-s') \bmod n$
- With **k**, can compute private key **d**!!!
  - $s = k^{-1}(z+rd) \bmod n$
    $d = r^{-1}(sk-z) \bmod n$