Bonus Lecture #3: Digital Signatures in Blockchain Protocols (Part 2 of 2)

COMS 4995-001: The Science of Blockchains

URL: https://timroughgarden.org/s25/

Tim Roughgarden

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   - in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   - signature depends on both sk and the message being signed
3. *Verification algorithm:* maps msg + sig + pk → "yes"/"no".
   - anyone who knows pk can verify correctness of an alleged signature

Goals for Bonus Lecture #3

1. Schnorr signatures.
   - used in Bitcoin since the Taproot upgrade in 2021, EdDSA in Solana
2. tl;dr of elliptic curves.
   - groups where discrete log appears harder than in Z_p^*
   - basis of all signature schemes used in blockchain protocols
3. tl;dr of ECDSA signatures.
   - what users use to sign transactions in Bitcoin and Ethereum
4. tl;dr of BLS signatures.
   - used by Ethereum validators to sign consensus-layer messages

Signatures Based on Exponentiation

General approach to key generation:

- sk := random t-bit string, pk := g^{sk}
  - repeated squaring ➔ can compute pk from sk with ≤ 2t group operations

Signatures Based on Exponentiation

General approach to key generation:

- sk := random t-bit string, pk := g^{sk}
  - repeated squaring➔can compute pk from sk with ≤ 2t group operations
- g is generator of a cyclic group with order/size q ≥ 2^t
  - group: has a well-defined binary operation, each element has inverse
  - ex: each element of Z_7^* a power of 3, each element of Z_{11}^* a power of 2

Signatures Based on Exponentiation

General approach to key generation:

- sk := random t-bit string, pk := g^{sk}
  - repeated squaring➔can compute pk from sk with ≤ 2t group operations
- g is generator of a cyclic group with order/size q ≥ 2^t
  - group: has a well-defined binary operation, each element has inverse
  - ex: each element of Z_7^* a power of 3, each element of Z_{11}^* a power of 2

Discrete log (DL) assumption: there is no (randomized) polynomial-time algorithm that can recover x from g and g^x (with non-negligible probability). [necessary condition for security]

Computing Discrete Logarithms

Problem: recover x from g and g^x.

- note: can be done efficiently in some groups (e.g., addition modulo p)

Computing Discrete Logarithms

Problem: recover x from g and g^x.

- note: can be done efficiently in some groups (e.g., addition modulo p)

Facts: solvable with $O(\sqrt{q})$ group operations (q = order of G)

- to get 128 bits of security (standard target), need ≥ 256-bit private keys

Computing Discrete Logarithms

Problem: recover x from g and g^x.

– note: can be done efficiently in some groups (e.g., addition modulo p)

Facts: solvable with $O(\sqrt{q})$ group operations (q = order of G)

– to get 128 bits of security (standard target), need ≥ 256-bit private keys

- solvable with $O((\ln q)^3)$ group operations on quantum computer

– big quantum computers ➔ discrete log approach to signatures broken

Computing Discrete Logarithms

Problem: recover x from g and g^x.

– note: can be done efficiently in some groups (e.g., addition modulo p)

Facts: solvable with $O(\sqrt{q})$ group operations (q = order of G)

– to get 128 bits of security (standard target), need ≥ 256-bit private keys

- solvable with $O((\ln q)^3)$ group operations on quantum computer
  – big quantum computers ➔ discrete log approach to signatures broken
- in Z_p^*, can solve with $\approx \exp\{1.92 \times (\ln p)^{1/3} \times (\ln \ln p)^{2/3}\}$ ops
  – for 128 bits of security, need ≥ 3072-bit private keys
  – same story for RSA signatures (can use GNFS for factoring, as well)

What Could Go Wrong?

Failure modes for signature schemes:

What Could Go Wrong?

Failure modes for signature schemes:

1. can extract pk from sk
   - silly example: if G = addition modulo p
   - address by choosing group where the discrete log problem is hard

What Could Go Wrong?

Failure modes for signature schemes:

1. can extract pk from sk
   - silly example: if G = addition modulo p
   - address by choosing group where the discrete log problem is hard
2. can compute signatures without knowing sk
   - silly example: signature = message (or f(message))

What Could Go Wrong?

Failure modes for signature schemes:

1. can extract pk from sk
   - silly example: if G = addition modulo p
   - address by choosing group where the discrete log problem is hard
2. can compute signatures without knowing sk
   - silly example: signature = message (or f(message))
3. signature leaks (computationally recoverable) info about sk
   - silly example: signature = sk
     - or anything from which sk can be easily extracted

Schnorr Signatures

Schnorr Signatures (in One Slide)

- let G = cyclic group with generator g, prime order $q \approx 2^t$
- key generation:
  - sk = random x in {0,1,2,…,q-1}
  - pk = g^x
- to sign:
  - choose random b in {0,1,2,…,q-1}
  - set a := $h(m \,||\, g^b)$ [h = cryptographic hash function, acts like random]
  - output as signature ($r:=g^b$, $s:=(ax+b) \bmod q$) [≈ 2t bits]
- to verify: accept signature (r,s) ⇔ $g^s = (pk)^{h(m \,||\, r)} \cdot r$

Deriving Schnorr Signatures

- let G = cyclic group with generator g, prime order q ≈ 2^t
- key generation:
  - sk = random x in {0,1,2,…,q-1}
  - pk = g^x

Deriving Schnorr Signatures

- let G = cyclic group with generator g, prime order $q \approx 2^t$
- key generation:
  - sk = random x in {0,1,2,…,q-1}
  - pk = g^x
- to sign/verify: let m = message
  - goal: design a signing function f(x,m) such that:

Deriving Schnorr Signatures

- let G = cyclic group with generator g, prime order $q \approx 2^t$
- key generation:
  - sk = random x in {0,1,2,…,q-1}
  - pk = g^x
- to sign/verify: let m = message
  - goal: design a signing function f(x,m) such that:
    - given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
      - even though it only knows g^x and not x itself

Deriving Schnorr Signatures

- let G = cyclic group with generator g, prime order $q \approx 2^t$
- key generation:
  - sk = random x in {0,1,2,…,q-1}
  - pk = g^x
- to sign/verify: let m = message
  - goal: design a signing function f(x,m) such that:
    - given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
      - even though it only knows g^x and not x itself
    - can't reverse engineer x from s=f(x,m) (and m, and g^x)

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
     - even though it only knows g^x and not x itself
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point:

Deriving Schnorr Signatures

- goal: design a signing function $f(x,m)$ such that:
  1. given $pk = g^x$, m, and s, verification algorithm can check if $s = f(x,m)$
     – even though it only knows g^x and not x itself
  2. can't reverse engineer x from $s=f(x,m)$ (and m, and g^x)

Starting point: $f(x,m) := m \cdot x \pmod{q}$.

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
     – even though it only knows g^x and not x itself
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point: f(x,m) := m • x (mod q).

- bad news: (2) fails [given s:=f(x,m) and m, can extract x = s/m (mod q)]

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
     - even though it only knows g^x and not x itself
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point: f(x,m) := m • x (mod q).

- bad news: (2) fails [given s:=f(x,m) and m, can extract x = s/m (mod q)]
- good news: (1) holds (i.e., can verify using g^x but not x):

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
     – even though it only knows g^x and not x itself
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point: f(x,m) := m • x (mod q).

- bad news: (2) fails [given s:=f(x,m) and m, can extract x = s/m (mod q)]
- good news: (1) holds (i.e., can verify using g^x but not x):
  – $s = m \cdot x \pmod q \Leftrightarrow g^s = g^{m \cdot x}$

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
     – even though it only knows g^x and not x itself
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point: f(x,m) := m • x (mod q).

- bad news: (2) fails [given s:=f(x,m) and m, can extract x = s/m (mod q)]
- good news: (1) holds (i.e., can verify using g^x but not x):
  – $s = m \cdot x \pmod q \Leftrightarrow g^s = g^{m \cdot x}$
  – verification algorithm accepts $\Leftrightarrow g^s = (pk)^m$

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point: f(x,m) := m • x (mod q).

- bad news: (2) fails [given s:=f(x,m) and m, can extract x = s/m (mod q)]
- good news: (1) holds (i.e., can verify using g^x but not x):
  - s = m • x (mod q) ⇔ $g^s = g^{m \cdot x}$
  - verification algorithm accepts ⇔ $g^s = (pk)^m$

Note: if m a multiple of q, 0 always a valid signature (for any sk).

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Starting point: f(x,m) := m • x (mod q).

- bad news: (2) fails [given s:=f(x,m) and m, can extract x = s/m (mod q)]
- good news: (1) holds (i.e., can verify using g^x but not x):
  - $s = m \cdot x \pmod q \Leftrightarrow g^s = g^{m \cdot x}$
  - verification algorithm accepts $\Leftrightarrow g^s = (pk)^m$

Note: if m a multiple of q, 0 always a valid signature (for any sk).

- fix: use h(m) instead of m, where h = a cryptographic hash function

Deriving Schnorr Signatures

- goal: design a signing function $f(x,m)$ such that:
  1. given $pk = g^x$, m, and s, verification algorithm can check if $s = f(x,m)$
  2. can't reverse engineer x from $s=f(x,m)$ (and m, and g^x)

Revised starting point: $f(x,m) := h(m) \cdot x \pmod q$. [h = CHF]

- bad news: (2) fails [given $s:=f(x,m)$ and m, can extract $x = s/h(m) \pmod q$]
- good news: (1) holds (i.e., can verify using g^x but not x):
  – $s = m \cdot x \pmod q \Leftrightarrow g^s = g^{h(m) \cdot x}$
  – verification algorithm accepts $\Leftrightarrow g^s = (pk)^{h(m)}$

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
    1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
    2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Next idea: $f(x,m) := (h(m) \cdot x + b) \bmod q$.   [h = CHF]

Deriving Schnorr Signatures

- goal: design a signing function $f(x,m)$ such that:
  1. given $pk = g^x$, m, and s, verification algorithm can check if $s = f(x,m)$
  2. can't reverse engineer x from $s=f(x,m)$ (and m, and g^x)

Next idea: $f(x,m) := (h(m) \cdot x + b) \bmod q$. [$h$ = CHF]

- b must be secret [else, given s and m, can extract $x = (s-b)/h(m) \pmod q$]

Deriving Schnorr Signatures

- goal: design a signing function $f(x,m)$ such that:
  1. given $pk = g^x$, m, and s, verification algorithm can check if $s = f(x,m)$
  2. can't reverse engineer x from $s=f(x,m)$ (and m, and g^x)

Next idea: $f(x,m) := (h(m) \cdot x + b) \bmod q$. [$h$ = CHF]

- b must be secret [else, given s and m, can extract $x = (s-b)/h(m) \pmod q$]
- so choose b at random from $\{0,1,2,\ldots,q-1\}$
  - called a "nonce" [for "number used once"]

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Next idea: f(x,m) := (h(m) • x + b) mod q.   [h = CHF]

- b must be secret [else, given s and m, can extract x = (s-b)/h(m) (mod q)]
- so choose b at random from {0,1,2,…,q-1}
  - called a "nonce" [for "number used once"]
- question: how can verification algorithm check that s = f(x,m)?

Deriving Schnorr Signatures

- goal: design a signing function f(x,m) such that:
  1. given pk = g^x, m, and s, verification algorithm can check if s = f(x,m)
  2. can't reverse engineer x from s=f(x,m) (and m, and g^x)

Next idea: f(x,m) := (h(m) • x + b) mod q. [h = CHF]

- b must be secret [else, given s and m, can extract x = (s-b)/h(m) (mod q)]
- so choose b at random from {0,1,2,…,q-1}
  - called a "nonce" [for "number used once"]
- question: how can verification algorithm check that s = f(x,m)?
  - insight: using that s = h(m) • x + b (mod q) ⇔ $g^s = g^{h(m) \cdot x+b}$, see that verification algorithm only needs to know g^b, not b itself

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where:

Deriving Schnorr Signatures (con’d)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}

Deriving Schnorr Signatures (con’d)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where:
  - r := g^b

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where:
  - $r := g^b$
  - $s := (h(m) \cdot x + b) \bmod q$
    - intuitively, not enough info to extract x from s (one equation, two unknowns)

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where:
  - $r := g^b$
  - $s := (h(m) \cdot x + b) \bmod q$
    - intuitively, not enough info to extract x from s (one equation, two unknowns)

Proposed verification algorithm: [given m, pk, and (r,s)]

- accept $\Leftrightarrow g^s = (pk)^{h(m)} \cdot r$

Deriving Schnorr Signatures (con’d)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Notes:

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Notes:

- signatures not unique (one per choice of nonce)

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Notes:

- signatures not unique (one per choice of nonce)
- signature size = 2 group elements (q ≈ 2^{256} ➔ signature ≈ 512 bits)

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Notes:

- signatures not unique (one per choice of nonce)
- signature size = 2 group elements ($q \approx 2^{256}$ ➔ signature ≈ 512 bits)
- reuse a nonce ➔ can extract x (two equations, two unknowns)

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⬄ $g^s = (pk)^{h(m)} \cdot r$

Notes:

- signatures not unique (one per choice of nonce)
- signature size = 2 group elements ($q \approx 2^{256}$ ➔ signature ≈ 512 bits)
- reuse a nonce ➔ can extract x (two equations, two unknowns)
- $h(m_1)=h(m_2)$ ➔ same signatures (r,s) valid for both m_1 and m_2

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Issue:

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Issue: For any m, pk, and s, can forge a valid signature (r,s) by taking $r = g^{-x \cdot h(m)+s}$. [can compute r without knowing x, why?]

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where $r := g^b$ and $s := (h(m) \cdot x + b) \bmod q$

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Issue: For any m, pk, and s, can forge a valid signature (r,s) by taking $r = g^{-x \cdot h(m)+s}$. [can compute r without knowing x, why?]

Fix: Use h(m ll r) instead of h(m).

Deriving Schnorr Signatures (con'd)

Proposed signing algorithm: [m = message, x = private key, h = CHF]

- choose b at random from {0,1,2,…,q-1}
- signature := (r,s), where r := g^b and s := (h(m) • x + b) mod q

Proposed verification algorithm: accept ⇔ $g^s = (pk)^{h(m)} \cdot r$

Issue: For any m, pk, and s, can forge a valid signature (r,s) by taking $r = g^{-x \cdot h(m)+s}$. [can compute r without knowing x, why?]

Fix: Use h(m ll r) instead of h(m).

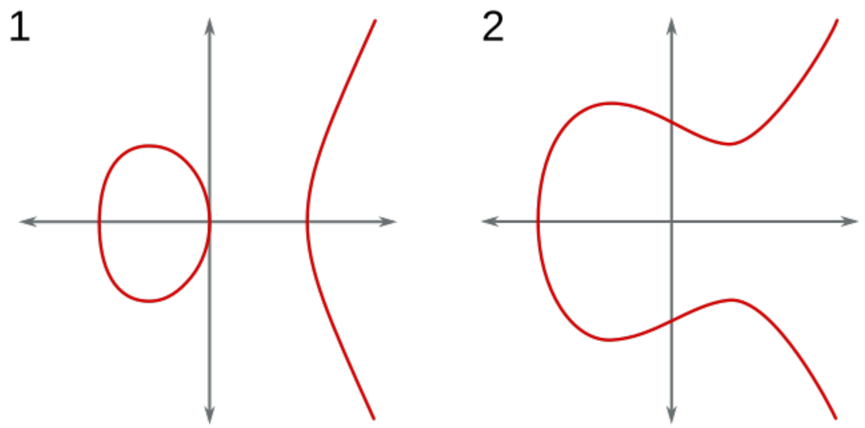
- h CHF (as good as random) ➔ infeasible to find r satisfying $r = g^{-x \cdot h(m \,\|\, r)+s}$

Schnorr Signatures (in One Slide)

- let G = cyclic group with generator g, prime order $q \approx 2^t$
- key generation:
  - sk = random x in $\{0,1,2,\ldots,q-1\}$
  - pk = g^x
- to sign:
  - choose random b in $\{0,1,2,\ldots,q-1\}$
  - set $a := h(m \,||\, g^b)$ [h = cryptographic hash function, acts like random]
  - output as signature $(r:=g^b, s:=(ax+b) \bmod q)$ [$\approx$ 2t bits]
- to verify: accept signature $(r,s) \Leftrightarrow g^s = (pk)^{h(m \,||\, r)} \cdot r$

Elliptic Curves

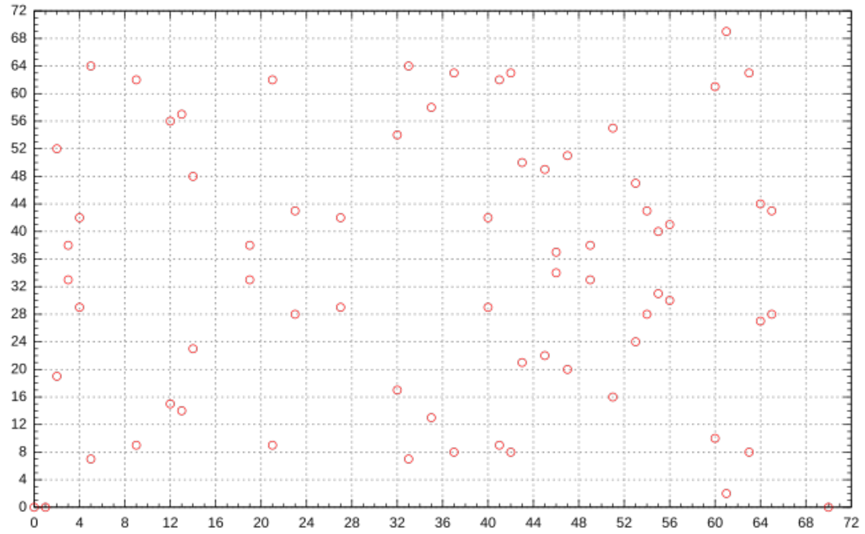
Approximate definition: an *elliptic curve* is the set of solutions (x,y) to an equation of the form $y^2 = x^3+ax+b$ (for some a,b).



$y^2 = x^3 - x$ $\qquad$ $y^2 = x^3 - x + 1$

Elliptic Curves (over Finite Fields)

Approximate definition: an *elliptic curve* is the set of solutions (x,y) to an equation of the form $y^2 = x^3+ax+b \pmod{p}$ (for some a,b).



Set of affine points of elliptic curve $y^2 = x^3 - x$ over finite field $\mathbf{F}_{71}$.

Elliptic Curves (over Finite Fields)

Approximate definition: an *elliptic curve* is the set of solutions (x,y) to an equation of the form $y^2 = x^3+ax+b \pmod p$ (for some a,b).

Non-obvious fact: the points of an elliptic curve form a group under a suitable operation (would take 10-20 minutes to explain).

Elliptic Curves (over Finite Fields)

Approximate definition: an *elliptic curve* is the set of solutions (x,y) to an equation of the form $y^2 = x^3+ax+b \pmod p$ (for some a,b).

Non-obvious fact: the points of an elliptic curve form a group under a suitable operation (would take 10-20 minutes to explain).

Example: secp256k1. [used in Bitcoin and Ethereum]

- defining equation: $y^2 = x^3 + 7 \bmod (2^{256}-2^{32}-977)$
- group of prime order (➔ cyclic), canonical generator

ECDSA Signatures

tl;dr: like Schnorr signatures, with two changes:

1. start from $f(x) = (x + h(m)) \bmod q$ rather than $f(x) = (h(m) \cdot x) \bmod q$
   – same chain of reasoning leads to a different verification equation (from ElGamal)

ECDSA Signatures

tl;dr: like Schnorr signatures, with two changes:

1. start from $f(x) = (x + h(m)) \bmod q$ rather than $f(x) = (h(m) \cdot x) \bmod q$
   - same chain of reasoning leads to a different verification equation (from ElGamal)
2. instead of $r = g^a$, use r = x-coordinate of g^a
   - note: only makes sense if G = elliptic curve (over some Z_p)

ECDSA Signatures

tl;dr: like Schnorr signatures, with two changes:

1. start from $f(x) = (x + h(m)) \bmod q$ rather than $f(x)= (h(m) \cdot x) \bmod q$
   – same chain of reasoning leads to a different verification equation (from ElGamal)
2. instead of $r = g^a$, use r = x-coordinate of g^a
   – note: only makes sense if G = elliptic curve (over some Z_p)

- to sign: [some details omitted]
  – choose random a in $\{0,1,2,\ldots,q-1\}$, $r :=$ x-coordinate of g^a
  – set $s := a^{-1}(r \cdot x + h(m)) \bmod q$

ECDSA Signatures

tl;dr: like Schnorr signatures, with two changes:

1. start from $f(x) = (x + h(m)) \bmod q$ rather than $f(x) = (h(m) \cdot x) \bmod q$
   – same chain of reasoning leads to a different verification equation (from ElGamal)
2. instead of $r = g^a$, use r = x-coordinate of g^a
   – note: only makes sense if G = elliptic curve (over some Z_p)

- to sign: [some details omitted]
  – choose random a in {0,1,2,…,q-1}, r := x-coordinate of g^a
  – set $s := a^{-1}(r \cdot x + h(m)) \bmod q$
- to verify: accept signature (r,s) ⇔ r = x-coordinate of $(g^{h(m)} \cdot (pk)^r)^{s^{-1}}$

tl;dr of BLS Signatures

- famous for signature aggregation properties
  - can combine many signatures (for different pks) into one, verify the aggregate
  - used by Ethereum validators (for quorum certificates, effectively)

tl;dr of BLS Signatures

- famous for signature aggregation properties
  - can combine many signatures (for different pks) into one, verify the aggregate
  - used by Ethereum validators (for quorum certificates, effectively)

Details:

- messages and signatures live in an elliptic curve group G_1 (prime order q)
- public keys live in an elliptic curve group G_2 (prime order q)

tl;dr of BLS Signatures

- famous for signature aggregation properties
  - can combine many signatures (for different pks) into one, verify the aggregate
  - used by Ethereum validators (for quorum certificates, effectively)

Details:

- messages and signatures live in an elliptic curve group G_1 (prime order q)
- public keys live in an elliptic curve group G_2 (prime order q)
- sk := random x in {1,2,...,q-1}, pk := $(g_2)^x$

tl;dr of BLS Signatures

- famous for signature aggregation properties
  - can combine many signatures (for different pks) into one, verify the aggregate
  - used by Ethereum validators (for quorum certificates, effectively)

Details:

- messages and signatures live in an elliptic curve group G_1 (prime order q)
- public keys live in an elliptic curve group G_2 (prime order q)
- sk := random x in {1,2,...,q-1}, pk := $(g_2)^x$
- pairing: bilinear map e(.,.) from G_1 x G_2 to a target group G_T

tl;dr of BLS Signatures

- famous for signature aggregation properties
  - can combine many signatures (for different pks) into one, verify the aggregate
  - used by Ethereum validators (for quorum certificates, effectively)

Details:

- messages and signatures live in an elliptic curve group G_1 (prime order q)
- public keys live in an elliptic curve group G_2 (prime order q)
- sk := random x in {1,2,...,q-1}, pk := $(g_2)^x$
- pairing: bilinear map e(.,.) from G_1 x G_2 to a target group G_T
- signature on m := $h(m)^x$ [h = hash function from msg space to G_1]

tl;dr of BLS Signatures

- famous for signature aggregation properties
  - can combine many signatures (for different pks) into one, verify the aggregate
  - used by Ethereum validators (for quorum certificates, effectively)

Details:

- messages and signatures live in an elliptic curve group G_1 (prime order q)
- public keys live in an elliptic curve group G_2 (prime order q)
- sk := random x in {1,2,...,q-1}, pk := $(g_2)^x$
- pairing: bilinear map e(.,.) from G_1 x G_2 to a target group G_T
- signature on m := $h(m)^x$ [h = hash function from msg space to G_1]
- to verify (m,pk,sig): accept $\Leftrightarrow$ $e(H(m),pk) = e(sig, g_2)$