Threshold ECDSA

Threshold signature schemes enable sharing signing power amongst n parties such that any subset of t + 1 can jointly sign, but any smaller subset cannot.

I Model, Definitions and Tools

1.1 Model

Communication Model

We assume that our computation model is composed of a set of n players $P_1, \cdots, P_n$ connected by a complete network of point-to-point channels and a broadcast channel.

The Adversary

We assume that an adversary, A, can corrupt up to t of the n players in the network. A learns all the information stored at the corrupted nodes, and hears all the broadcasted messages. We consider two type of adver saries:

1.2 Definitions

Signature Scheme

A signature scheme S is a triple of efficient randomized algorithms $(Key - Gen, Sig, Ver)$.

- **Key-Gen** is the key generator algorithm.

  on input the security parameter 1^λ , it outputs a pair (y, x), such that y is the public key and x is the secret key of the signature scheme.

- **Sig** is the signing algorithm:

  on input a message m and the secret key x, it outputs sig, a signature of the message m.

  Since Sig can be a randomized algorithm there might be several valid signatures sig of a message m under the key x; with $Sig(m, x)$ we will denote the set of such signatures

- **Ver** is the verification algorithm.

On input a message m, the public key y, and a string sig, it checks whether sig is a proper signature of m, i.e. if $sig \in Sig(m, x)$.

Threshold secret sharing

Given a secret value x we say that the values $(x_1, \cdots, x_n)$ constitute a (t, n)-threshold secret sharing of x if t (or less) of these values reveal no information about x, and if there is an efficient algorithm that outputs x having $t + 1$ of the values x_i as inputs.

Threshold signature schemes.

Let $S = (Key - Gen, Sig, Ver)$ be a signature scheme. A (t, n)-threshold signature scheme TS for S is a pair of protocols $(Thresh - Key - Gen, Thresh - Sig)$ for the set of players $P_1, \cdots, P_n$.

- **Thresh-Key-Gen** is a distributed key generation protocol used by the players to jointly generate a pair (y, x) of public/private keys on input a security parameter 1^λ .
- **Thresh-Sig** is the distributed signature protocol. The private input of P_i is the value x_i . The public inputs consist of a message m and the public key y. The output of the protocol is a value $sig \in Sig(m, x)$.

1.3 Tools

Additively Homomorphic Encryption

We assume the existence of an encryption scheme E which is additively homo- morphic modulo a large integer N, One instantiation of a scheme with these properties is **Paillier's encryption scheme**.

ref: [Paillier's encryption scheme]()

With $\bigoplus_{i=1}^{t+1}$ a*i, we denote the summation over the addition operation $+*E$ of the encryption scheme:

$$\bigoplus_{i=1}^{t+1} a_i = a_1 +_E a_2 +_E \cdots +_E a_{t+1}$$

Threshold Cryptosystems

In a (t, n)-threshold cryptosystem, there is a public key pk with a matching secret key sk which is shared among n players with a (t, n)-secret sharing.

When a message m is encrypted under pk, $t + 1$ players can decrypt it via a communication protocol that does not expose the secret key.

More formally, a public key cryptosystem ϵ is defined by three efficient algorithms:

- key generation **Enc-Key-Gen** that takes as input a security parameter λ, and outputs a public key pk

and a secret key sk.

- An encryption algorithm **Enc** that takes as input the public key pk and a message m, and outputs a ciphertext c. Since **Enc** is a randomized algorithm, there will be several valid encryptions of a message m under the key pk; with $Enc(m, pk)$ we will denote the set of such ciphertexts.

- And a decryption algorithm Dec which is run on input c, sk and outputs m, such that $c \in Enc(m, pk)$.

A (t, n) threshold cryptosystem $T\epsilon$, consists of the following protocols for n players $P_1, \cdots, P_n$.

- A key generation protocol **TEnc-Key-Gen** that takes as input a security parameter λ, and the parameter t, n, and it outputs a public key pk and a vector of secret keys $(sk_1, \cdots, sk_n)$ where sk_i is private to player P_i . This protocol could be obtained by having a trusted party run Enc-Key-Gen and sharing sk among the players.

- A threshold decryption protocol **TDec**, which is run on public input a ciphertext c and private input the share sk_i . The output is m, such that $c \in Enc(m, pk)$.

Threshold variations of Paillier's scheme have been presented in the literature:

- O. Baudron, P.-A. Fouque, D. Pointcheval, G. Poupard and J. Stern. Practical Multi-Candidate Election System. PODC'01

- I. Damg˚ard and M. Jurik. A Generalisation, a Simplification and Some Appli- cations of Paillier's Probabilistic Public-Key System. PKC'01, LNCS Vol.1992, pp.119–136

- I. Damg˚ard, M. Koprowski: Practical Threshold RSA Signatures without a Trusted Dealer. EUROCRYPT 2001: LNCS Vol.2045, pp. 152-165

- C. Hazay, G.L. Mikkelsen, T. Rabin, T. Toft. and A.A. Nicolosi: Efficient RSA key generation and threshold Paillier in the two-party setting.

Independent Trapdoor Commitments

A trapdoor commitment scheme allows a sender to commit to a message with information-theoretic privacy.

A (non-interactive) **trapdoor commitment scheme** consists of four algorithms $KG, Com, Ver, Equiv$ with the following properties:

- KG is the key generation algorithm, on input the security parameter it outputs a pair pk, tk where pk is the public key associated with the commitment scheme, and tk is called the **trapdoor**.

- Com is the commitment algorithm. On input pk and a message M it outputs $[C(M), D(M)] = Com(pk, M, R)$ where r are the coin tosses. $C(M)$ is the commitment string, while $D(M)$ is the decommitment string which is kept secret until opening time.

- Ver is the verification algorithm. On input C, D and pk it either outputs a message M or $\perp$.

- $Equiv$ is the algorithm that opens a commitment in any possible way given the trapdoor information. It takes as input pk, strings M, R with $[C(M), D(M)] = Com(pk, M, R)$, a message $M' \neq M$ and a string T. If $T = tk$ then $Equiv$ outputs D' such that $Ver(pk, C(M), D') = M'$.

We note that if the sender refuses to open a commitment we can set $D = \perp$ and $Ver(pk, C, \perp) = \perp$. Trapdoor commitments must satisfy the following properties:

- **Correctness:**

  If $[C(M), D(M)] = Com(pk, M, R)$ then $Ver(pk, C(M), D(M)) = M$.

- **Information Theoretic Security:**

  For every message pair M, M' the distributions $C(M)$ and $C(M')$ are statistically close.

- **Secure Binding:**

  We say that an adversary A wins if it outputs C, D, D' such that $Ver(pk, C, D) = M, Ver(pk, C, D') = M'$ and $M \neq M'$. We require that for all efficient algorithms A, the probability that A wins is negligible in the security parameter.

- **Independence:**

  if the honest parties open their commitments in different ways using the trapdoor, the adversary cannot change the way he opens his commitments C_j based on the honest parties' opening.

II Scheme of GG16

Initialization phase

In this phase, a common reference string containing the public information pk for an independent trapdoor commitment KG, Com, Ver, $Equiv$ is selected and published. This could be accomplished by a trusted third party, who can be assumed to erase any secret information (i.e. the trapdoor of the commitment) after selection.

The common parameters G, g, q for the DSA scheme are assumed to be known.

Key generation protocol

```
from klefki.types.algebra.concrete import EllipticCurveCyclicSubgroupSecp256k1 as
ECC
from klefki.types.algebra.concrete import EllipticCurveGroupSecp256k1 as Cruve
from klefki.types.algebra.concrete import FiniteFieldCyclicSecp256k1 as CF
from klefki.types.algebra.concrete import FiniteFieldSecp256k1 as F
from klefki.zkp.pedersen import PedersonCommitment
from klefki.types.algebra.utils import randfield
from klefki.bitcoin.address import gen_address
```

Here we describe how the players can jointly generate a DSA key pair $(x, y = g_x)$ with y public and x shared among the players.

The idea is to generate a public key E for an additively $(\mod N)$ homomorphic encryption scheme E, together with the secret key D in shared form among the players.

The value N is is chosen to be larger than q^8

```
from klefki.crypto.paillier import Paillier
from klefki.zkp.pedersen import PedersonCommitment, com as commit
from functools import partial
from klefki.types.algebra.concrete import FiniteFieldCyclicSecp256k1 as CF
from klefki.numbers.primes import generate_prime
from functools import reduce
from operator import mul, add
```

```
q = CF(CF.P)
```

```
P = generate_prime(1024)
Q = generate_prime(1024)
Pai = Paillier(P, Q)
E, D = Pai.E, Pai.D
```

Then a value x is generated, and encrypted with E, with the value $\alpha = E(x)$ made public.

Note that this is an implicit (t, n) secret sharing of x, since the decryption key of E is shared among the players.

- Each player P_i selects a random value $x_i \in Z_q$, computes $y_i = g^{x_i} \in G$ and $[C_i, D_i] = Com(y_i)$;

```
G = ECC.G
n = 3
xs = [randfield(CF) for _ in range(n)]
ys = [G ** x for x in xs]

trap = randfield(CF)
H = G ** trap
com = partial(PedersonCommitment, H=H, G=G)

coms = [com(x=y.value[0], r=y.value[1]) for y in ys]
```

```
x = reduce(add, xs)
```

- Each player P_i broadcast C_i
  - D_i which allows everybody to compute $y_i = Ver(C_i, D_i)$
  - $\alpha_i = E(x_i)$;
  - a ZK argument Π_i that states
    - $\exists \mu = y_i$
    - $D(a_i) = \mu$

  If any of the ZK arguments fails, the protocol terminates.

```
all([c.C == commit(H=H, G=G, *c.D) for c in coms])
```

```
True
```

```
from operator import mul
from klefki.types.algebra.meta import field
```

```
from operator import add
from functools import reduce

alpha = reduce(mul, [E(x.value) for x in xs])
y = reduce(add, ys)
FN = alpha.functor
```

- proof

```
assert CF(D(alpha)) == reduce(add, xs) == x
assert G ** CF(D(alpha)) == y
```

- The players compute $\alpha = \bigoplus_{i=1}^{t+1} a_i$ and $y = \sum_{i=1}^{t+1} y_i$

Signature Generation

The signature generation protocol is run on input m (the hash of the message M being signed)

```
from klefki.utils import to_sha256int

m = CF(to_sha256int("Hello Threshold ECDSA"))
```

Round 1

Each player P_i

1. choose $\rho_i \leftarrow Z_q$
2. compute $u_i = E(\rho_i)$ and $v_i = p_i \times_E \alpha = E(\rho_i x)$
3. compute $[C_{1,i}, D_{i,i}] = Com([u_i, v_i])$ and broadcast $C_{1,i}$

```
ps = [randfield(CF) for _ in range(n)]
us = [E(p) for p in ps]
vs = [alpha ** p for p in ps]

coms1 = [com(x=t[0], r=t[1]) for t in zip(us, vs)]
```

Round 2

Each player P_i broadcasts:

- $D_{1,i}$. This allow everybody to compute $[u_i, v_i] = Ver(C_{1,i}, D_{1,i})$
- a zero-knowledge argument $\Pi_{1,i}$ which states
  - $\exists \eta \in [-q^3, q^3]$ such that

    $D(u_i) = \eta$

    $D(v_i) = \eta D(E(x)$

Players compute $u = \bigoplus_{i=1}^{t+1} u_i = E(\rho)$ and $v = \bigoplus_{i=1}^{t+1} v_i = E(\rho x)$, where $\rho = \sum_{i=1}^{1+1} \rho_i$

```
all([c.C == commit(H=H, G=G, *c.D) for c in coms1])
```

```
True
```

```
u = reduce(mul, us)
v = reduce(mul, vs)
```

- proof

```
assert CF(D(u)) == reduce(add, (ps))
```

```
assert CF(D(alpha ** ps[1])) == x * ps[1]
```

```
assert CF(D(alpha ** ps[0] * alpha ** ps[1] * alpha ** ps[2])) == x * (ps[0] + ps[
1] + ps[2]) == CF(D(v))
```

```
assert CF(D(v)) == CF(D(E(reduce(add, ps) * x)))
```

Round 3

Each player P_i

- choose $k_i \in Z_q$ and $c_i \in R[-q^6, q^6]$
- computes $r_i = g^{k_i}$ and $w_i = (k_i \times_E u) +_E E(c_i q) = E(k_i \rho + c_i q))$
- computes $[C_{2,i}, D_{2_i} = Com(r_i, w_i)$ and broadcast C_{2i}

```
ks = [randfield(CF) for _ in range(n)]
rs = [G**k for k in ks]
```

```
cs = [randfield(CF) for _ in range(n)]
ws = [(u ** k) * E(c*q) for c, k in zip(cs, ks)]
```

```
coms2 = [com(x=c, r=w) for c, w in zip(cs, ws)]
```

```
dcoms2 = [c.D for c in coms]
```

- Proof

```
CF(D(u ** ks[0])) == reduce(add, (ps)) * ks[0]
```

```
True
```

```
CF(D(u ** ks[0] * E(cs[0] * q))) == CF(D(u ** ks[0]) + D(E(cs[0] * q))) \
                                 == reduce(add, (ps)) * ks[0] + cs[0] * q
```

```
True
```

Round 4

Each player P_i broadcasts

1. $D_{2,i}$, which allows everybody to compute $[r_i, w_i]$ = $Ver(C_{2,i}, D_{2,i})$

2. a zero-knowledge argument $\Pi_{(2,i)}$

Players compute $w = \oplus_{i=1}^{t+1} w_i = E(k\rho + cq)$ where $k = \sum_{i=1}^{t+1} k_i$ and $c = \sum_{i=1}^{t+1} c_i$. Players also compute $R = \Pi_i^{t+1} r_i = g^k$ and $r = H'$® $\in Z_q$

```
all([c.C == commit(H=H, G=G, *c.D) for c in coms2])
```

```
True
```

```
w = reduce(mul, ws)
R = reduce(mul, rs)
r = CF(R.x)
```

- Proof

```
R == G ** reduce(add, ks)
```

```
True
```

```
CF(D(w)) == reduce(add, (ps)) * reduce(add, ks) + reduce(add, cs) * q
```

```
True
```

Round 5

players jointly decrypt w using **TDec** to learn the value $\tau \in [-q^7, q^7]$ such that $\tau = k\rho \mod q$ and $\psi = \eta^{-1} \mod q$

Each player computes:

\begin{align} \sigma &= \psi \times*E [m\times*E u]+*E(r\times*E v)]\ &= \psi \times*E [E(m\rho) +*E E(r \rho x)]\ &= (k^{-1\rho^{-1}}\times_E [E(\rho (m+xr)) \ &=E(k^{-1}(m+xr))\ &=E(s) \end{align}

```
psi = CF(D(w))
```

```
pai = ~psi
```

```
sigma = ((u**m) * (v**r)) ** pai
```

- Proof

```
x = reduce(add, xs)
p = reduce(add, ps)
k = reduce(add, ks)
```

```
assert CF(D(u**m)) == m * p
assert CF(D(v**r)) == r * p * x
assert CF(D(u**m)) + CF(D(v**r)) == p * (m + x * r)
assert CF(D((u**m) * (v**r))) == p * (m + x * r)
assert CF(D(((u**m) * (v**r)) ** pai)) == (m + x * r)/k
```

```
assert CF(D(sigma)) == (m + x * r)/k
assert (G ** k).x == r
```

Round 6

The players invoke distributed decryption protocol TDec over the ciphertext σ. Let $s = D(\sigma) mod q$. The players output (r, s) as the signature for m.

```
r, s = r, CF(D(sigma))
```

Verify

```
from klefki.crypto.ecdsa.secp256k1 import verify
```

```
verify(pub=y, sig=(r, s), msg="Hello Threshold ECDSA")
```

```
True
```

III Scheme of GG18

A share conversion protocol

Assume that we have two parties Alice and Bob holding two secrets $a, b \in Z_q$ respectively which we can think of as multiplicative shares of a secret $x = ab \mod q$. Alice and Bob would like to compute secret additive shares α, β of x, that is random values such that $\alpha + \beta = x = ab \mod q$ with Alice holding a and Bob holding b.

Here we show a protocol based on an additively homomorphic scheme. We assume that Alice is associated with a public key E_A for an additively homomorphic scheme E over an integer N. Let $K > q$ also be a bound which will be specified later.

The players run on input G, g the cyclic group used by the DSA signature scheme. We assume that each player P_i is associated with a public key E_i for an additively homomorphic encryption scheme E. In our protocol we also assume that $B = g^b$ might be public.

MtA (for Multiplicative to Additive)

MtAwc (as MtA “with check”).

```
from klefki.crypto.paillier import Paillier
from klefki.numbers.primes import generate_prime
from klefki.types.algebra.meta import field
from klefki.types.algebra.utils import randfield
from klefki.types.algebra.concrete import EllipticCurveCyclicSubgroupSecp256k1 as
ECC
from klefki.types.algebra.concrete import EllipticCurveGroupSecp256k1 as Cruve
from klefki.types.algebra.concrete import FiniteFieldCyclicSecp256k1 as CF
from klefki.zkp.pedersen import PedersonCommitment
```

```
Pai_A = Paillier(generate_prime(32), generate_prime(32))
```

```
F_q = field(generate_prime(32), "q")
F_n = field(Pai_A.N)
```

```
a, b = randfield(F_q), randfield(F_q)
ab = a * b
```

Step 1

Alice initiates the protocol by

- sending $c_A = E_A(a)$ to Bob.
- proving in ZK that $a < K$ via a range proof.

```
c_a = Pai_A.E(a)
```

Step 2

Bob computes the ciphertext $c_B = b \times_E c_A +_E E_A(\beta') = E_A(ab + \beta')$ where β' is chosen uniformly at random in Z_N. Bob sets his share to $\beta = -\beta' \mod q$. He responds to Alice by

- sending c_B
- proving in ZK that $b < K$
- only if $B = g^b$ is public proving in ZK that he knoows b, β' s.t. $B = g^b$ and $c_B = b \times_E c_A +_E E_A(\beta')$

```
beta_ = randfield(F_q)
```

```
assert F_q(Pai_A.D(Pai_A.E(beta_) * c_a ** b)) == F_q(Pai_A.D(Pai_A.E(a*b + beta_)
))
c_b = Pai_A.E(beta_) * c_a ** b
```

```
beta = F_q(-beta_)
```

Step 3

Alice decrypts c_B to obtain α

```
alpha = F_q(Pai_A.D(c_b))
```

```
assert alpha + beta == a*b
```

Implementation

```
from functools import partial
from klefki.numbers import length

def MtA(a, b=None, p=None, q=None):
    if not (p and q):
        p, q = generate_prime(128), generate_prime(128)
    if not b:
        return partial(MtA, a=a)
    assert length(a.P) < length(p*q)
    Pai_A = Paillier(p, q)
    F_n = field(Pai_A.N)
    F_q = a.functor
    c_a = Pai_A.E(a)
    beta_ = randfield(F_q)
    beta =  F_q(-beta_)
    c_b = Pai_A.E(beta_) * c_a ** b
    alpha = F_q(Pai_A.D(c_b))
    assert a * b == alpha + beta
    return alpha, beta
```

```
RF = field(generate_prime(8))
a, b = randfield(RF), randfield(RF)
MtA(a)(b=b)
```

```
(FiniteField::100, FiniteField::143)
```

Key generation Protocol

- **Phase 1**. Each Player P_i select $u_i \in_R Z_q$; computes $[KGC_i, KGD_i] = Com(g^{u_i})$ and broadcast KGC_i. Each player P_i broadcast E_i the public key for Paillier' cryptosytem

```
n = 9
t = 6
G = ECC.G

p = generate_prime(128)
q = generate_prime(128)
pai = Paillier(p, q)
```

```
pai_pks = [Paillier(p, q) for _ in range(n)]
```

```
us = [randfield(CF) for _ in range(n)]
```

```
ys = [G ** u for u in us]

trap = randfield(CF)
H = G ** trap
com = partial(PedersonCommitment, H=H, G=G)
coms = [com(x=y.value[0], r=y.value[1]) for y in ys]
```

- **Phase 2**. Each Player P_i broadcast KCD_i, let y_i be the value decommitted by P_i. The player P_i performs a (t, n) Feldman-VSS of the value u_i, with y_i as the "free term in the exponent". The pubkey is set to $y = \prod_i y_i$. Each player adds the private shares received during the n VSS. The resoulting values x_i are a (t, n) SSSS of secret key $x = \sum_i u_i$. Note that the value $X_i = g_i^x$ are public.

```
from klefki.crypto.vss import VSS
from klefki.types.algebra.concrete import FiniteFieldCyclicSecp256k1 as CF
from klefki.types.algebra.concrete import EllipticCurveCyclicSubgroupSecp256k1 as
ECC
from functools import reduce
from operator import mul, add
```

```
vss = [VSS(CF, ECC.G).setup(u, t, n) for u in us]
assert len(vss) == n
```

```
ids = [randfield(CF) for i in range(n)]
vss_shares = [(CF(i), reduce(add, [v.f(i) for v in vss])) for i in ids]
```

```
xs = [i[1] for i in vss_shares]
ids = [i[0] for i in vss_shares]
```

```
x = reduce(add, us)
y = reduce(mul, ys)
assert G ** x == y
```

```
assert x == VSS.decrypt(vss_shares)
```

- **Phase 3**, let $N_i = p_i q_i$ be the RSA mod associated with E*i, Each palyer $P*i $provesinZKthatheknows$x_i$

Signature Generation

- **Prepare**

Using the appropriate Lagrangian coefficients $\lambda_{i,S}$ each player in S can locally map its own (t, n) share of x_i of x into a (t, t) share of x.

$$x = \sum_{j=0}^{k-1} f(x_i) \prod_{j=0;i \neq j}^{k-1} \frac{x_m}{x_j - x_i}$$

$$m_i = f(x_i) \prod_{i=0;j \neq j}^{k-1} \frac{x_m}{x_j - x_i}$$

```
ws = [xs[j] * reduce(mul, [ids[m] / (ids[m]-ids[j]) for m in range(t) if m != j])
for j in range(t)]
```

```
assert x == reduce(add, ws)
```

We will to sign msg m:

```
from klefki.utils import to_sha256int

m = CF(to_sha256int("Hello Threshold ECDSA"))
```

- **Phase1**.

Each Player P_i selects $k_i, \gamma_i \in_R Z_q$; computes $[C_i, D_i] = Com(G^{\gamma_i})$ and broad cast C_i. Define $k = \sum_{i \in s} k_i, \gamma = \sum_{i \in s} \gamma_i$. Note that

$$k\gamma = \sum_{i,j \in S} k_i \gamma_j \mod q \; kx = \sum_{i,j \in S} k_i w_i \mod q$$

```
ks = [randfield(CF) for _ in range(t)]
rs = [randfield(CF) for _ in range(t)]
```

```
k = reduce(add, ks)
r = reduce(add, rs)
```

- **Phase2**.

Every pair of player P_i, P_j engages in two $multiplicative - to - additive$ share conversation subprotocols

- 2.1 P_i, P_j run MtA with shares k_i, γ_j respectively. Let α_{ij} [resp. β_{ij}] be the share received by player P_i [resp. P_j] at the end of this protocol, i.e.

$$k_i \gamma_j = \alpha_{ij} + \beta_{ij}$$

Player P_i set $\delta_i = k_i \gamma_i + \sum_{j \neq i} \alpha_{ij} + \sum_{j \neq i} \beta_{ij}$. Note that the δ_i are a (t, t) additive sharing of $k\gamma = \sum_{i \in s} \delta_i$

```
p = generate_prime(512)
q = generate_prime(512)
```

```
PMtA = partial(MtA, p=p, q=q)
shares_kg = [
    [PMtA(a=ks[i], b=rs[j]) for j in range(t) if j != i]
for i in range(t)]
```

```
ds = [ks[i]*rs[i] + reduce(add, [s[0] + s[1] for s in shares_kg[i]]) for i in rang
e(t)]
```

```
assert k * r == reduce(add, ds)
```

- 2.2 P_i, P_j run MtAwc with shares k_i, w_i respectively. Let μ_{ij} [resp. ν_{ij}] be the share received by player P_i [resp. P_j] at the end of this protocol, i.e.

$$k_i w_i = \mu_{ij} + \nu_{ij}$$

Player P_i sets $\sigma_i = k_i w_i + \sum_{j \neq i} \mu_{ij} + \sum_{j \neq i} \nu_{ij}$

```
p = generate_prime(512)
q = generate_prime(512)
PMtA = partial(MtA, p=p, q=q)
shares_kw = [
    [PMtA(a=ks[i], b=ws[j]) for j in range(t) if j != i]
for i in range(t)]
```

```
sigmas = [ks[i] * ws[i] + reduce(add, [s[0] + s[1] for s in shares_kw[i]]) for i i
n range(t)]
```

```
assert k * x == reduce(add, sigmas)
```

- **Phase 3**

Every player P_i broadcasts δ_i and the players reconstruct $\delta = \sum_{i \in s} \delta_i = k\gamma$. The players compute $\delta^{-1} \mod q$.

```
d = reduce(add, ds)
```

```
d_ = ~d
```

- **Phase 4**

Each Player P_i broadcasts D_i, let Γ_i be the value decommited by P_i who proves in ZK that he knows γ_i s.t. $\Gamma_i = g^{\gamma_i}$ using Schnoor's protocol.

The players compute

$$R = \left[\prod_{i \in S} \Gamma_i\right]^{-1} = g^{(\sum_{i \in S} \gamma_i) k^{-1} \gamma^{-1}} = g^{\gamma k^{-1} \gamma^{-1}} = g^{k^{-1}}$$

and $r = R.x$

```
assert k * r == reduce(add, ds)
```

```
reduce(add, ds) * (~r) == k
```

```
True
```

```
com_rs = [G**r for r in rs]
```

```
R = reduce(mul, com_rs) ** d_
```

```
assert R == G ** (~k)
```

```
r = CF(R.x)
```

- **Phase 5**

Each Player P_i sets $s_i = mk_i + r\sigma_i$. Note that

$$\sum_{i \in S} s_i = m \sum_{i \in s} k_i + r \sum_{i \in S} \sigma_i = mk + rkx = k(m + xr) = s$$

```
assert G ** x == y
assert k * x == reduce(add, sigmas)
assert k == reduce(add, ks)
```

```
assert m * reduce(add, ks) + reduce(add, sigmas) * r == k * (m + x * r)
```

```
ss = [m * ks[i] + sigmas[i] * r for i in range(t)]
```

```
s = reduce(add, ss)
```

```
s == k * (m + x * r)
```

```
True
```

Verify

```
from klefki.crypto.ecdsa.secp256k1 import verify
```

```
verify(pub=y, sig=(r, s), msg="Hello Threshold ECDSA")
```

```
True
```

Ref:

- Antonio Salazar Cardozo, Threshold ECDSA — Safer, more private multi-signatures, https://blog.keep.network/threshold-ecdsa-safer-more-private-multi-signatures-51153f3e9ed2
- Gennaro, Rosario, Steven Goldfeder, and Arvind Narayanan. “Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security.” In Applied Cryptography and Network Security, edited by Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, 9696:156–74. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-39555-5_9.
- Rosario Gennaro and Steven Goldfeder. 2018. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18). Association for Computing Machinery, New York, NY, USA, 1179–1194. DOI:https://doi.org/10.1145/3243734.3243859