Fireblocks Multi-Party (EC)DSA, ver 1.2

Contents

1 Notation and Assumptions

We denote $\mathbb{Z}_n$ for the (additive) group of integers modulo n. $\mathbb{G} = \langle g \rangle$ is a (multiplicative) group generated by element g of prime order $q = |\mathbb{G}| = \mathrm{ord}(g)$ (hence $k \mapsto g^k$ is an isomorphism from $\mathbb{Z}_q$ to $\mathbb{G}$). We assume the discrete log problem is hard in $\mathbb{G}$. Concretely we use Secp256k1 parameters.

Denote a player/party in a protocol by $\mathcal{P}_i$ (i from a relevant index set), and it has a corresponding fixed public unique non-zero $id_i \in \mathbb{Z}_q$. We assume secure peer-to-peer channels between each two parties (enabled via PKI and message encryption and authentication), and also a broadcast channel (enabled via agreement round after each broadcast).

2 Key Shares Distribution

2.1 Background

A set of parties wish to generate and distribute between themselves shares for a secret value (in $\mathbb{Z}_q$), such that no single party knows the secret. Moreover, only specific "authorized" subsets of the parties should have enough information enabling them to reconstruct the secret (authorization is predefined by an access structure). Specifically, we require that the secret value is a linear combination of the authorized subset's secret shares (with public coefficients, which depend on the authorized subset wishing to reconstruct).

Each unauthorized subset of parties shouldn't get any information on the secret, given their shares and also the communication received during the key shares distribution. The secret should be distributed uniformly when at least one the parties is honest, and all honest parties should verify the correctness of the share they received (without knowing the secret).

To enable verification of generated shares, during the distribution protocol the parties also broadcast public group elements in $\mathbb{G}$ related to their shares. Specifically, if $\mathcal{P}_i$ got at the end of the protocol a share $x_i \in \mathbb{Z}_q$, then g^{x_i} should be public. Also, if all the $\{x_i\}_i$ are shares of the secret $x \in \mathbb{Z}_q$, then g^x should also be public.

After the key shares are generated and distributed between the parties, each authorized subset of parties can participate in the multi-party DSA signing protocol of section 3, with respect to the publc DSA key g^x. We remark that this is done without the parties reconstructing the secret x itself, and the signing process keeps the promise that no party gets more information about the secret at the end of the signing protocol (and even multiple invocation of it).

2.2 t/n Threshold

In this access structure, the parties are $\mathcal{P}_i$ for $i = 1, \ldots, n$. A threshold of t parties should be able to reconstruct the secret x (which will be defined by the protocol execution), t, n define the access structure. Each $\mathcal{P}_i$ starts with a secret seed $\mathit{seed}_i \in \mathbb{Z}_q$ (it can be randomly sampled or generated by some deterministic pseudo-random process like HD key generation). $\mathcal{P}_i$ splits seed_i to $u_{i,1}, \ldots, u_{i,n}$ using Shamir secret sharing scheme in $\mathbb{Z}_q$ (with threshold t, using the players id), and sends $u_{i,j}$ for $\mathcal{P}_j$. After $\mathcal{P}_i$ has received all shares $u_{j,i}$ for $j = 1, \ldots, n$, he computes $x_i := \sum_{j=1}^{n} u_{j,i}$ as his DSA private key share. The generated $\{x_i\}_{i=1,\ldots,n}$ form a t/n Shamir secret sharing of $x = \sum_{j=1}^{n} \mathit{seed}_j$ (since each $\{u_{i,j}\}_{j=1,\ldots,n}$ is t/n Shamir secret sharing of seed_i, and by the linearity, see also [3]).

To validate the shares were distributed as expected, the parties also send all their $g^{u_{i,j}}$, and also $g^{a_{i,j}}$ where $a_{i,j}$ are the Shamir secret sharing polynomial coefficients which generated $u_{i,j}$. With this, each party can verify their share consistency by computing the Shamir polynomial in the exponent (see details in [3], [1] and in the specification section). Also, this public values enable computing the share's public value g^{x_i}, and also the secret's public value g^x.

Notice that for an authorized group of players $S \subseteq \{1, \ldots, n\}$ (namely, $|S| \geq t$) it holds $x = \sum_{j \in S} \lambda_{j,S} \cdot x_j$, where $\lambda_{j,S}$ are the Lagrange interpolation coefficients.

2.3 Disjunction of Groups

For two integers m, n, this access structure can be described by

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} \mathcal{P}_{i,j} = (\mathcal{P}_{1,1} \wedge \ldots \wedge \mathcal{P}_{1,n}) \vee \ldots \vee (\mathcal{P}_{m,1} \wedge \ldots \wedge \mathcal{P}_{m,n}) .$$

Namely, there are m groups, each with n parties (usually a single party is a user's device, and the rest $n-1$ parties in the group are security enhancing servers). The secret can be reconstructed only by all players in some group (for any group).

Initially, we set $m = 1$, and the secret is generated and distributed by the n/n threshold protocol for the n parties in the single group. Later, more groups can be added, and fresh shares to the existing secret can be distributed (to the new parties), as follows.

2.3.1 Adding a Group

Assuming there are currently m groups in the access structure, in order to add group $m+1$ (with players $\mathcal{P}_{m+1,1}, \ldots, \mathcal{P}_{m+1,n}$), group 1 (or any other authorized group) does the following. Each $\mathcal{P}_{1,j}$ for $j = 1, \ldots, n$ splits his share $x_{1,j}$ to $u_{j,1}, \ldots, u_{j,n}$ by n/n Shamir secret sharing, and sends $\lambda_j \cdot u_{j,k}$ to new player $\mathcal{P}_{m+1,k}$, where λ_j is the Lagrange interpolation coefficient for reconstructing the secret x from $x_{1,j}$ (namely, from group 1). Each $\mathcal{P}_{m+1,k}$ then computes $x_{m+1,k} = \sum_{i=1}^{n} \lambda_k \cdot u_{j,k}$ as its secret share.

By the initial Lagrange interpolation for $x_{1,j}$ and the above protocol, it also holds $\sum_{k=1}^{m} \lambda_k \cdot x_{m+1,k} = x$, hence the new shares are generated as required. Similarly to before, to allow verification, each party sends it's public values corresponding to both his shares and his randomness (in Shamir polynomials).

3 Gennaro & Goldfeder [1] Signing Protocol

In the following we assume all players follow the protocol honestly, so the only security goal is privacy. Namely, after participating in the protocol, a player (or an eavesdropper) doesn't gain any new information. This is called seciroty against "semi-honest" adversary. To enable full (provable) security against maliciously acting adversaries – commitments, zero knowledge proofs and verification of secret sharing are added between communication rounds (see full specification of the protocol at the end).

3.1 Signature Generation

Parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ (specifically, an authorized group in the disjunction of groups access structure) want to sign a fixed known message $m \in \mathbb{Z}_q$ (which in practice, m is the hash of the real message). After the key generation phase, each $\mathcal{P}_i$ ($i \in S$, where $S = \{1, \ldots, n\}$) has a share x_i of a secret x, such that the linear combination holds $x = \sum_{i=1}^{n} \lambda_i \cdot x_i$, with public λ_i (which are defined by the signing players id's). We assume g^x is public, as are all the public share value g^{x_i}.

Define $w_i := \lambda_{i,S} \cdot x_i$ (which player $\mathcal{P}_i$ computes). The goal is to distributively compute a signature (r, s),

$$R = g^{k^{-1}}, r = H_x(R), s = k \cdot (m + r \cdot x),$$

where $k \in \mathbb{Z}_q$ is uniformly sampled and H_x is a known hashing algorithm defined by the signature protocol (e.g., x-projection in ECDSA). The idea is for each party $i \in S$ to uniformly sample a share k_i, and define $k := \sum_{i \in S} k_i$ (which nobody can know, since this break DSA security, knowing k enables knowing x after seeing a valid signature).

First, the parties want to distributively compute $g^{k^{-1}}$, by sharing g^γ for uniform $\gamma = \sum_{i \in S} \gamma_i$ (where $\mathcal{P}_i$ sampled γ_i), and sharing $\delta = k \cdot \gamma$ (which hides k, since γ is uniform). Thus enabling computing $r = (g^\gamma)^{\delta^{-1}} = g^{k^{-1}}$. However, sharing $\delta = k \cdot \gamma$ is not trivial, since no party should know either k or γ (since together with δ, gives k). So to share $\delta = k \cdot \gamma$, notice

$$\delta = \left(\sum_{i \in S} k_i\right) \cdot \left(\sum_{i \in S} \gamma_i\right) = \sum_{i,j \in S,S} k_i \cdot \gamma_j.$$

$\mathcal{P}_i$ has k_i and $\mathcal{P}_j$ has γ_j, so the two parties will share $k_i \cdot \gamma_j$, by using a **M**ultiplication-**t**o-**A**ddition protocol, which uses Paillier Encryption. With MtA, two players interact, each with its own input (say a, b inputs), and after two messages they each get uniform α, β (and no other information) such that $a \cdot b = \alpha + \beta$.

3.1.1 Paillier Encryption

A public key encryption scheme. A party generates (ek, dk), enc/dec key pair, publishes ek to the world. Everyone can encrypt a message m by $c := \mathsf{Enc}_{ek}(m)$, but only the player can decrypt $m = \mathsf{Dec}_{dk}(c)$. Paillier have efficient additivity and (public) scalar multiplication (by everyone who can encrypt):

$$\mathsf{Enc}_{ek}(m_1) \oplus \mathsf{Enc}_{ek}(m_2) = \mathsf{Enc}_{ek}(m_1 + m_2), \quad \alpha \odot \mathsf{Enc}_{ek}(m) = \mathsf{Enc}_{ek}(\alpha \cdot m).$$

3.1.2 Multiplication-to-Addition

Alice and Bod have inputs a, b, respectively. At the end of the MtA protocol, they (securely) get α, β, respectively, satisfying $\alpha + \beta = b \cdot a$ (and nothing else).

Now we can describe in more detail the distributed creation of $g^{k^{-1}}$.

3.1.3 MPC for $r = g^{k^{-1}}$

Player $\mathcal{P}_i$ does the following

1. Sample k_i, γ_i. Broadcast g^{γ_i}, each player computes $g^\gamma = \prod_{j \in S} g^{\gamma_j}$.
2. With each other player $j \in S \setminus \{i\}$ engage MtA(k_i, γ_j) and MtA(k_j, γ_i). Get shares $\alpha_{i,j}, \beta_{j,i}$ respectively ($\mathcal{P}_i$ gets only one input in each MtA. $\mathcal{P}_j$ of course gets the other related shares $\beta_{i,j}, \alpha_{j,i}$ such that $\alpha_{i,j} + \beta_{i,j} = k_i \cdot \gamma_j$, and similarly for the symmetric indices).
3. Compute and broadcast $\delta_i := k_i \cdot \gamma_i + \sum_{j \in S \setminus \{i\}} (\alpha_{i,j} + \beta_{j,i})$.
4. Compute $\delta := \sum_{j \in S} \delta_j = \sum_{i,j} k_i \cdot \gamma_j = k \cdot \gamma$, and get $(g^\gamma)^{\delta^{-1}} = g^{k^{-1}}$.

3.1.4 MPC for s

After all parties got $r = g^{k^{-1}}$, now they can distributively compute s (recall a party must not know k or x). Recall

$$s = k \cdot (m + r \cdot x) = \left(\sum_{i \in S} k_i\right) \cdot \left(m + r \cdot \left(\sum_{j \in S} w_j\right)\right) = m \cdot \sum_{i \in S} k_i + r \cdot \sum_{i,j \in S,S} k_i \cdot w_j.$$

In order to generate $k_i \cdot w_j$ for players $\mathcal{P}_i, \mathcal{P}_j$, the engage again in MtA in the s generation protocol.

Player $\mathcal{P}_i$ does as follows:

1. With each other player $j \in S \setminus \{i\}$ engage MtA(k_i, w_j) and MtA(k_j, w_i). Get shares $\mu_{i,j}, \nu_{j,i}$ respectively ($\mathcal{P}_i$ gets only one input in each MtA. $\mathcal{P}_j$ of course gets the other related shares $\nu_{i,j}, \mu_{j,i}$ such that $\mu_{i,j} + \nu_{i,j} = k_i \cdot w_j$, and similarly for the symmetric indices).
2. Compute $\sigma_i := k_i \cdot w_i + \sum_{j \in S \setminus \{i\}} (\mu_{i,j} + \nu_{j,i})$. Notice $\sum_{i \in S} \sigma_i = k \cdot x$, but no player can compute it.
3. Compute and broadcast $s_i := k_i \cdot m + \sigma_i \cdot r$.
4. Compute $s := \sum_{j \in S} s_j = k \cdot m + k \cdot w \cdot r = k \cdot (m + x \cdot r)$.

All players finally got (r, s) which is a valid signature of m.

Remarks

- The above protocol is describes in logical interaction steps, in the real protocol some steps are combined.
- Recall everything above is private for semi-honest parties, no information is leaked if all parties follows the protocol.

To make the protocol secure against active adversaries, we need to take the interaction steps much more carefully, essentially interweaving cryptographic commitments of (later to be revealed) values add zero knowledge proofs at critical points, enabling the security proof. See the specification for full details.

4 Technical Specification

4.1 Cryptographic Tools

4.1.1 Commitment Scheme

Using cryptographically secure hash function H. Public security parameter λ (default $\lambda = q$).

- $(C, D) := \text{Com}(\lambda, x)$
  - Sample $r \xleftarrow{\$} \mathbb{Z}_\lambda$
  - Output $C := H(x||r), D := x||r$

Inside a larger protocol, whenever a decommitment D is revealed, it is always checked as valid with respect to a previously received commitment C, namely check $C = H(D)$.

4.1.2 Paillier Encryption

Using security parameter λ (default $\lambda = 4q$).

- $(ek, dk) := \text{PaillierKeygen}(\lambda)$
  - Sample two primes p, q, each of λ bit-length
  - Set $N := p \cdot q$, $\varphi = (p-1) \cdot (q-1)$
  - Output encryption key $ek := N$, decryption key $dk := \varphi$
- $c := \mathsf{Enc}_{ek}(m)$ for $m \in \mathbb{Z}_N^*$

  - Sample $r \xleftarrow{\$} \mathbb{Z}_N^*$
  - Output $c := (1+N)^m \cdot r^N \pmod{N^2}$

- $m := \mathsf{Dec}_{dk}(c)$ for $c \in \mathbb{Z}_{N^2}$
  - Compute $\alpha := \varphi^{-1} \pmod{N}$
  - Compute $\beta := c^{\varphi} \pmod{N^2}$, get integer value $\beta \in \mathbb{Z}_{N^2}$
  - Compute $\gamma := (\beta - 1)/N$ (as integer division, not modulo), get $\gamma \in \mathbb{Z}_N$
  - Output $m := \gamma \cdot \alpha \pmod{N}$

4.1.3 Feldman-Shamir Verifiable Secret Sharing Scheme

Sharing a secret $s \in \mathbb{Z}_q$. Players $\{1, \ldots, n\}$ are identified by $ID = \{id_i\}_{i=1\ldots n}$ for unique non-zero $id_i \in \mathbb{Z}_q$.

- $y := \mathrm{EvalPol}(\overline{\mathit{coeffs}}, x)$
  Evaluate a polynomial P (defined by coefficients), at point x, return $P(x)$
  - Set $\mathit{deg} := \left|\overline{\mathit{coeffs}}\right| - 1$
  - Output $y := \sum_{i=0}^{deg} \mathit{coeffs}_i \cdot x^i$

- $(\overline{\mathit{shares}}, \overline{\mathit{ver}}) := \mathrm{SplitPol}(ID, \overline{\mathit{coeffs}})$
  Evaluate polynomial at $ID \subset \mathbb{Z}_q$ points, and output public verification for coefficients (group elements)
  - Set $n = |ID|$ $(ID = \{id_i\}_{i=1,\ldots,n})$, and $\mathit{thr} := \left|\overline{\mathit{coeffs}}\right|$
  - For $i = 1, \ldots, n$, set $\mathit{shares}_i := \mathrm{EvalPol}(\overline{\mathit{coeffs}}, id_i)$
  - For $i = 0, \ldots, \mathit{thr} - 1$, set $\mathit{ver}_i := g^{\mathit{coeffs}_i}$ (in G)
  - Output $\overline{\mathit{shares}} := (\mathit{shares}_i)_{i=1,\ldots,n}$, $\overline{\mathit{ver}} := (\mathit{ver}_i)_{i=0,\ldots,\mathit{thr}-1}$

- $(\overline{\mathit{shares}}, \overline{\mathit{ver}}) := \mathrm{SplitSecret}(ID, \mathit{thr}, \mathit{secret})$
  Split secret to thr/n Samir shares (defined by ID), and add public shares for verifications (group elements).
  - Set $\mathit{coeffs}_0 := \mathit{secret}$
  - For $j = 1, \ldots, \mathit{thr} - 1$, sample $\mathit{coeffs}_j \xleftarrow{\$} \mathbb{Z}_q$
  - Set $n = |ID|$ $(ID = \{id_i\}_{i=1,\ldots,n})$
  - Verify $\mathit{thr} \leq n$, and all $id_i \in ID$ are unique non-zero
  - Calculate and output $\mathrm{SplitPol}(ID, \overline{\mathit{coeffs}})$

- $h := \mathrm{GroupEvalPol}(\overline{\mathit{elem}}, x)$
  Evaluate polynomial in the exponent at x, where elem define the coefficients in the exponent.
  - Set $\mathit{deg} := \left|\overline{\mathit{elem}}\right| - 1$
  - Output $h := \sum_{i=0}^{deg} (\mathit{elem}_i)^{x^i}$ (in G)

- $\mathit{isValid} := \mathrm{PubShareVerify}(\mathit{id}, \mathit{pubshare}, \overline{\mathit{ver}})$
  Check if $\mathit{pubshare}$ (group element) corresponds to verification polynomial in the exponent (at id)
  - Output `True` iff $\mathit{pubshare} = \mathrm{GroupEvalPol}(\overline{\mathit{ver}}, \mathit{id})$

- $\lambda := \mathrm{LagrangeCoeffAtPoint}(SID, i, x)$ $(SID \subseteq ID)$
  Return Lagrange coefficient at x of secret share of player $i \in SID$.
  Needed when player i wants to use share to reconstruct secret $(x = 0)$.
  Or to create share for new player $(x = \mathit{newid})$.
  - Set $k := |SID|$, $SID = \{id_j\}_{j=1,\ldots,k}$ (usually should be $k \geq \mathit{threshold}$)
  - Check $1 \leq i \leq k$
  - Compute and output

$$\lambda := \prod_{\substack{j=1,\ldots,k \\ j \neq i}} \frac{x - id_j}{id_i - id_j}.$$

4.1.4 MtA Protocol

Alice and Bob has inputs $a, b \in \mathbb{Z}_q$, using MtA protocol, they (securely) receive $\alpha, \beta \in \mathbb{Z}_q$ such that $a \cdot b = \alpha + \beta$. Alice generates a Paillier (ek, dk) key pair ($ek = N$ of bit-length $8q$), where ek is publicly sent (with ZKP). Also, g^b should be public, and will be verified with respect to b (known to Bob).

- Alice:
  - Generate Paillier key $(ek, dk) :=$ PaillierKeygen$(\lambda = 4q)$
  - Compute $c_A := \mathsf{Enc}_{ek}(a)$
  - Generate $zkp1 :=$ ZKPOK-Factorization-Paillier$(dk; ek)$
  - Generate $zkp2 :=$ ZKP-Coprime-Paillier$(dk; ek)$
  - Send $c_A, ek, zkp1, zkp2$
- Bob:
  - Receive $c_A, ek, zkp1, zkp2$ from Alice
  - Verify $zkp1, zkp2$ with respect to ek
  - Sample $\beta' \overset{\$}{\leftarrow} \mathbb{Z}_N$
  - Compute and send $c_B := (b \odot c_A) \oplus \mathsf{Enc}_{ek}(\beta')$
  - Send $B := g^b, B' := g^{\beta'}$
  - Send ZKP-Scnorr$(b; B, g)$
  - Send ZKP-Scnorr$(\beta'; B', g)$
  - Bob sets $\beta := -\beta' \pmod q$
- Alice:
  - Receive c_B, B, B' and two ZKP from Bob
  - Verify validity of ZKP
  - Set $\alpha := \mathsf{Dec}_{dk}(c_B) \pmod q$
  - Verify $B^a \cdot B' = g^\alpha$

Remark: with negligible probability $1/q^6$, the guarantee $a \cdot b = \alpha + \beta$ fails.

4.1.5 ZKP-Schnorr(a ; A, g)

Alice holds a secret $a \in \mathbb{Z}_q$, for which $A = g^a \in G$ is public, and she wishes to prove (to anyone) in zero knowledge that she holds the correct a as above. For this she does the following:

- Sample $b \overset{\$}{\leftarrow} \mathbb{Z}_q$
- Compute $B := g^b \in G$
- Compute $c := Hash(id_{\text{Alice}}||B||A) \in \mathbb{Z}_q$
- Compute $s := b - a \cdot c \in \mathbb{Z}_q$

Alice sends the ZKP: (B, s). Bob (a receiver), who knows the public $A, g \in G$, verifies by:

- Receive (B, s)
- Compute $c = Hash(id_{\text{Alice}}||B||A)$
- Verify $B = A^c \cdot g^s$

By default, whenever a ZKP is broadcast, each recipient checks its validity.

4.1.6 ZKP-2-Schnorr(x, y ; V, g_1, g_2)

Alice holds two secret $x, y \in \mathbb{Z}_q$, for which $V = g_1^x \cdot g_2^y \in G$ is public, and she wishes to prove (to anyone) in zero knowledge that she holds the correct x, y as above. For this she does the following:

- Sample $a, b \xleftarrow{\$} \mathbb{Z}_q$
- Compute $U := g_1^a \cdot g_2^b \in G$
- Compute $c := Hash(id_{\text{Alice}}||U||V) \in \mathbb{Z}_q$
- Compute $\alpha := a - c \cdot x, \beta = b - c \cdot y$

Alice sends the ZKP: (U, α, β). Bob (a receiver), who knows the public $V, g_1, g_2 \in G$, verifies by:

- Receive (U, α, β)
- Compute $c = Hash(id_{\text{Alice}}||U||V)$
- Verify $U = V^c \cdot g_1^\alpha \cdot g_2^\beta$

4.1.7 ZKP-Coprime-Paillier($dk; ek$)

Alice holds a secret $dk = \varphi(N)$ for which $N = ek$ is public, and she wishes generates a non-interactive zero-knowledge proof that $gcd(N, \varphi(N)) = 1$ (namely, $N, \varphi(N)$ are coprime). This is done according to section 3.1 of [2]. Set $K = 16$ and α the product of all primes smaller than 2^{16}.

- Set $seed_1 := id_{\text{session}}||id_{\text{Alice}}||id_{\text{Bob}}||N$
- For $i = 1, \ldots, K$: compute $(x_i, seed_{i+1}) :=$ DeterministicSample$(seed_i, N)$
  if $gcd(x_i, N) > 1$, repeat sampling x_i
- Compute $M := N^{-1} \pmod{\varphi(N)}$
- Compute $y_i := x_i^M \pmod N$

Alice sends ZKP: $(y_1, \ldots, y_K)$. Bob who knows N verifies:

- Set $seed_1 := id_{\text{session}}||id_{\text{Alice}}||id_{\text{Bob}}||N$
- For $i = 1, \ldots, K$: compute $(x_i, seed_{i+1}) :=$ DeterministicSample$(seed_i, N)$
  if $gcd(x_i, N) > 1$, repeat sampling x_i
- Check $gcd(N, \alpha) = 1$
- Check $y_i^N = x_i \pmod N$ for all $i = 1, \ldots, K$

4.1.8 ZKPOK-Factorization-Paillier($dk; ek$)

Alice holds a secret $dk = \varphi(N)$ for which $N = ek$ is public, and she wishes generates a non-interactive zero-knowledge proof of knowledge of p, q such that $N = p \cdot q$. Denote λ as the security parameter of the Paillier encryption, namely $2\lambda = |N|$ (default $\lambda = 4q$). The ZKP is according to [4], with parameters $K = 10$, $A = N/2$ and B of bit–length $|B| = 0.5\lambda$.

- Set $seed_1 := id_{\text{session}}||id_{\text{Alice}}||id_{\text{Bob}}||N$
- For $i = 1, \ldots, K$: compute $(z_i, seed_{i+1}) :=$ DeterministicSample$(seed_i, N)$
- Sample $r \xleftarrow{\$} \mathbb{Z}_A$
- Compute $x := \text{Hash}\left((z_1^r \bmod N) || \ldots || (z_K^r \bmod N)\right)$
- Compute $e := \text{Hash}_B\left(N||x||z_1|| \ldots ||z_K\right) \in \mathbb{Z}_B$
- Compute $y := r + (N - \varphi(N)) \cdot e \in \mathbb{Z}_A$

Alice sends the ZKP: (x, y). Bob (a receiver), who knows the public $N = ek$, verifies by:

- Receive (x, y)
- Set $seed_1 := id_{\text{session}}||id_{\text{Alice}}||id_{\text{Bob}}||N$
- For $i = 1, \dots, K$: compute $(z_i, seed_{i+1}) :=$ DeterministicSample$(seed_i, N)$
- Compute $e = \text{Hash}_B\left(N||x||z_1||\dots||z_K\right) \in \mathbb{Z}_B$
- Verify $x = \text{Hash}\left(\left(z_1^{y-N\cdot e} \bmod N\right)||\dots||\left(z_K^{y-N\cdot e} \bmod N\right)\right)$

4.1.9 $(z, nextseed) :=$ DeterministicSample$(seed, N)$

Deterministically generate a pseudo-random element $z \in \mathbb{Z}_N$ from a given seed. Also output *nextseed* to allow next usage of DeterministicSample. This is to used simulate a common reference string in the Paillier-ZKP. We use a cryptographic hash function H outputting $2k$ pseudo-uniform bits, where $|seed| = k$.

1. Set $T = \lceil \frac{N}{k} \rceil$
2. For $i = 1 \dots T$: compute $(z_i, seed) := Hash(seed)$ (k bits for z_i and k for next $seed$).
3. Set $z = z_1||\dots||z_T$
4. If $z \notin \mathbb{Z}_N$, go to step 1
5. Output $(z, seed)$

4.2 Key Shares Distribution

Player $\mathcal{P}_i$ ($i \in S = \{1, \dots, n\}$ with $SID = \{id_1, \dots, id_n\}$) does the following

1. - Sample random $seed_i \xleftarrow{\$} \mathbb{Z}_q$
   - Compute $(\overline{u}_i, \overline{ver}_i) :=$ SplitSecret$(SID, n, seed_i)$
   - Compute $\overline{pubver}_i := \left(g^{u_{i,j}}\right)_{j=1,\dots,n}$
   - Compute $(C_i, D_i) = \text{Com}\left(\overline{ver}_i||\overline{pubver}_i\right)$
   - Broadcast C_i
2. - Broadcast D_i
   - Send $u_{i,j}$ to $\mathcal{P}_j$ (for each j)
3. For each j do:
   - Receive $u_{j,i}, D_j$
   - Extract $\overline{ver}_j, \overline{pubver}_j$ from D_j received (contains $ver_{j,0} = g^{seed_j}$)
   - Check $g^{u_{j,i}} = pubver_{j,i}$
   - For all $\ell \in S$: Check PubShareVerify $\left(id_\ell, pubver_{j,\ell}, \overline{ver}_j\right)$
4. - Compute $x_i := \sum_{j=1}^{n} u_{j,i}$
   - Compute $y := \prod_{j=1}^{n} g^{seed_j}$ as the public key
   - For each j: compute $y_j := \prod_{\ell=1\dots n} pubver_{j,\ell}$
   - Verify $y_i = g^{x_i}$
   - Broadcast ZKP-Schnorr$(x_i\ ;\ y_i, g)$

4.3 Adding a Group

Player $\mathcal{P}_i$ ($i \in S, S = \{1, \ldots, n\}$) does the following with the new group of users with indices N (and $NID = \{id_r\}_{r \in N}$):

1. - Compute $\lambda_i := \text{LagrangeCoeffAtPoint}(SID, i, 0)$
   - Compute $(\overline{u}_i, \overline{ver}_i) := \text{SplitSecret}(NID, \lambda_i \cdot x_i)$
   - Compute $\overline{pubver}_i := (g^{u_{i,r}})_{r \in N}$
   - Send $u_{i,r}, \overline{ver}_i, \overline{pubver}_i$ to $\mathcal{P}_r$ (for each $r \in N$)

Each new player $\mathcal{P}_r$ ($r \in N$) does the following:

1. For each $i \in S$ do:
   - Receive $u_{i,r}, \overline{ver}_i, \overline{pubver}_i$
   - Extract $g^{x_i} = ver_{i,0}$ from $\overline{ver}_i$
   - Check $g^{u_{i,r}} = pubver_{i,r}$
   - For all $\ell \in N$: Check PubShareVerify $\left(id_\ell, pubver_{i,\ell}, \overline{ver}_i\right)$
2. - Compute $x_r := \sum_{j \in S} u_{j,r}$
   - Compute $y := \prod_{j=1}^{n} g^{x_i}$ as the public key
   - Broadcast y_r, k_r, and ZKP-Schnorr$(x_r \; ; \, y_r, g)$

4.4 Signature Generation

Player $\mathcal{P}_i$ ($i \in S, S = \{1, \ldots, n\}, SID = \{id_1, \ldots, id_n\}$) does the following

1. - For each $j \in S$: compute $\lambda_j := \text{LagrangeCoeffAtPoint}(SID, j, 0)$
   - Set $w_i := \lambda_i \cdot x_i$
   - Sample $k_i, \gamma_i \leftarrow \mathbb{Z}_q$
   - Compute $(C_i, D_i) := \text{Com}(g^{\gamma_i} || g^{w_i})$
   - Broadcast C_i
2. Broadcast D_i
3. Receive D_j, and extract g^{γ_j}, g^{w_j} for all $j \neq i$
4. With each $\mathcal{P}_j$, $j \neq i$:
   A. Engage MtA (k_i, γ_j) as Alice (public g^{γ_j}) to get $\alpha_{i,j}$
   B. Engage MtA (k_j, γ_i) as Bob (public g^{γ_i}) to get $\beta_{j,i}$
5. With each $\mathcal{P}_j$, $j \neq i$:
   A. Engage MtA (k_i, w_j) as Alice (public g^{w_j}) to get $\alpha_{i,j}$
   B. Engage MtA (k_j, w_i) as Bob (public g^{w_i}) to get $\beta_{j,i}$
6. - Compute $\delta_i := k_i \cdot \gamma_i + \sum_{j \in S \setminus \{i\}} (\alpha_{i,j} + \beta_{j,i})$
   - Compute $\sigma_i := k_i \cdot w_i + \sum_{j \in S \setminus \{i\}} (\mu_{i,j} + \nu_{j,i})$
   - Broadcast δ_i (without σ_i)
7. - Get δ_j from all $\mathcal{P}_j$
   - Compute $\delta := \sum_j \delta_j$ and $\delta^{-1} \pmod q$
   - Compute $R := \left(\prod_j g^{\gamma_j}\right)^{\delta^{-1} \pmod q}$
   - Set $r = H_x(R)$ (x projection in ECDSA)
   - Set $s_i := m \cdot k_i + r \cdot \sigma_i$
   - Engage ValidateSigShares(s_i)
8. Broadcast s_i.

4.4.1 Validate Signature Shares

Each player $\mathcal{P}_i$ (of a set S of players), has a share $s_i \in \mathbb{Z}_q$ of the sum $s := \sum_i s_i$. The protocol verifies (s, r) is a valid signature corresponding to public key y, of a (public) message m, where $r = H_x(R)$ for $R = g^{k^{-1}}$ which is also public.

Player $\mathcal{P}_i$ does the following:

1. - Sample $\ell_i, \rho_i \overset{\$}{\leftarrow} \mathbb{Z}_q$
   - Compute $V_i := R^{s_i} \cdot g^{\ell_i}$
   - Compute $A_i := g^{\rho_i}$
   - Compute $(\hat{C}_i, \hat{D}_i) := \text{Com}(V_i || A_i)$
   - Broadcast $\hat{C}_i$
2. - Broadcast $\hat{D}_i$
   - Broadcast ZKP-Schnorr$(\rho_i \ ; \ A_i, g)$ (A_i from $\hat{D}_i$)
   - Broadcast ZKP-2-Schnorr$(s_i, \ell_i \ ; \ V_i, R, g)$ (V_i, R in $\hat{D}_i$)
   - Compute $A := \prod_{j \in S} A_j$
   - Compute $V := g^{-m} \cdot y^{-r} \cdot \prod_{j \in S} V_j$
3. - Compute $U_i := V^{\rho_i}$, $T_i := A^{\ell_i}$
   - Compute $(\tilde{C}_i, \tilde{D}_i) := \text{Com}(U_i || T_i)$
   - Broadcast $\tilde{C}_i$
4. - Broadcast $\tilde{D}_i$
   - Verify $\prod_{j \in S} T_j = \prod_{j \in S} U_j$

Remarks

- Abort (and broadcast failure) if some step/validation doesn't go through as planed.

References

[1] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ecdsa with fast trustless setup. *CCS 18*, 2018.

[2] Rosario Gennaro, Daniele Micciancio, and Tal Rabin. An efficient non-interactive statistical zero-knowledge proof system for quasi-safe prime products. *5th ACM CCS*, 1998.

[3] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. *Avances in Cryptology*, 1995.

[4] Guillaume Poupard and Jacques Stern. Short proofs of knowledge for factoring. In *Public Key Cryptography*, 2000.