# CGGMP Specification

Dfns

## 1 Introduction

We provide a specification for our implementation of the CGGMP threshold signing protocol [2].

## 2 Notation and Preliminaries

$\mathbb{E}$ denotes an elliptic-curve group of prime order[1] $q$ with generator $G$. If $P \in \mathbb{E}$ is a point on the curve, then $P|_x$ denotes the $x$-coordinate of $P$. We let $\mathbb{Z}_n = [n] = \{0, \ldots, n-1\}$, and let $\mathbb{Z}_n^*$ be the subset of elements of $\mathbb{Z}_n$ co-prime to $n$, i.e., $\mathbb{Z}_n^* = \{i \mid i \in \mathbb{Z}_n \wedge \gcd(i, n) = 1\}$. For integers $a, b, \ell$, we set $[a; b) = \{a, \ldots, b-1\}$, $\pm\ell = \{-\ell/2, \ldots, 0, \ldots, \ell/2\}$ if $\ell$ is even, and $\pm\ell = \{-(\ell-1)/2, \ldots, 0, \ldots, (\ell-1)/2\}$ if $\ell$ is odd. Let $\mathbb{B} = [256] = \{0, 1, \ldots, 255\}$, i.e. $\mathbb{B}$ is all possible values of a byte, so $\mathbb{B}^*$ denotes a space of all bytestrings of any size, and $\mathbb{B}^n$ denotes a space of all bytestrings of length $n$. We write $x \leftarrow X$ to denote sampling a uniform element $x$ from a set $X$.

### 2.1 Safe-Prime Generation

A safe prime $p$ has the form $p = 2p' + 1$ where $p'$ is also prime. Assuming a primality test $\mathsf{IsPrime}$, a trivial way to generate a (random) safe prime is to repeatedly sample $p'$ in the appropriate range, test $p'$ for primality, and then (if $p'$ is prime) test $2p'+1$ for primality. (We ignore here the possibility of error in $\mathsf{IsPrime}$.)

Primality testing is expensive, and the trivial algorithm is wasteful in the sense that it tests $p'$ for primality even when it is clear that $p = 2p' + 1$ will not be prime (e.g., if $p' = 1 \bmod 3$). We can avoid this by using simple sieving, as in Algorithm 1 (following [5]).

**Algorithm 1** Generating a (random) safe prime

Let $B$ be a set containing the first $n$ odd primes // $n$ is a parameter
**while** (1) **do**
    Choose (random) $p'$ in the appropriate range
    If $p' \bmod q \in \{0, (q-1)/2\}$ for some $q \in B$ $\mathsf{continue}$
    If $(!\mathsf{IsPrime}(p'))$ $\mathsf{continue}$
    If $(\mathsf{IsPrime}(2p'+1))$ **return** $p = 2p' + 1$

The number of primes $n$ to use for sieving is a parameter that can be heuristically optimized. Note that as $n$ increases, the marginal benefit of sieving decreases while the cost of sieving increases.

[1] The curve order is denoted by `curve_order` in the code. Note that $q$ is also used for the verifier's challenge space in [2], but we use $Q$ for that instead.

## 2.2 Using the Chinese Remainder Theorem

Arithmetic modulo $N$ can be optimized when the (partial) factorization of $N$ is known. Say $N = N_1 \cdot N_2$ where $N_1, N_2 > 1$ and $\gcd(N_1, N_2) = 1$. (Note that $N_1, N_2$ need not be prime.) Then a computation modulo $N$ can be optimized by (1) separately carrying out the computation modulo $N_1$ and $N_2$, and then (2) combining the results. We illustrate the for the particular case of exponentiation (i.e., computing $s^x \bmod N$), but the same idea can be applied for multiplication, multiexponentiation, etc.

**Exponentiation modulo $N_1, N_2$.** Computing $s^x \bmod N_1$ and $s^x \bmod N_2$ will, in general, be faster than computing $s^x \bmod N$ directly because (1) $N_1, N_2$ are shorter than $N$, and (2) assuming the factorizations of $N_1, N_2$ are known, we can reduce the exponent $x$ modulo $\phi(N_1)$ (resp., $\phi(N_2)$) before performing the respective exponentiations. Namely, we can use the fact that, e.g.,

$$s^x \bmod N_1 = s^{x \bmod \phi(N_1)} \bmod N_1$$

(assuming $\gcd(s, N_1) = 1$.)

**Combining the results.** Let $\beta = N_1^{-1} \bmod N_2$. If we have computed $r_1 = s^x \bmod N_1$ and $r_2 = s^x \bmod N_2$, we can compute $s^x \bmod N$ as

$$r_1 + ((r_2 - r_1) \cdot \beta \bmod N_2) \cdot N_1.$$

(Note that no reduction modulo $N$ is needed, and the result will already be in the correct range.) To see that this gives the correct answer, note that

$$r_1 + ((r_2 - r_1) \cdot \beta \bmod N_2) \cdot N_1 = r_1 \bmod N_1$$
$$r_1 + ((r_2 - r_1) \cdot \beta \bmod N_2) \cdot N_1 = r_1 + \left((r_2 - r_1) \cdot N_1^{-1}\right) \cdot N_1 = r_2 \bmod N_2.$$

In our context, the case of interest is when the modulus is $N^2 = p^2 q^2$ and the factors $p, q$ are known. Algorithm 2 shows a complete algorithm for exponentiation modulo $N^2$ in that case.

**Algorithm 2** Computing $s^x \bmod N^2$, where $N^2 = p^2 q^2$ with $p, q$ distinct primes and $\gcd(s, N^2) = 1$

$\beta := p^{-2} \bmod q^2$ // this can be computed in a preprocessing step
$\phi_1 = p \cdot (p-1),\ \phi_2 := q \cdot (q-1)$
$x_1 := x \bmod \phi_1,\ x_2 := x \bmod \phi_2$
$s_1 := s \bmod p^2,\ s_2 := s \bmod q^2$
$r_1 := s_1^{x_1} \bmod p^2,\ r_2 := s_2^{x_2} \bmod q^2$
res $:= r_1 + ((r_2 - r_1) \cdot \beta \bmod q^2) \cdot p^2$
**return** res

## 2.3 Paillier Encryption Scheme

We include a description of the algorithms that constitute the Paillier encryption scheme.

1. keygen generates a private key sk consisting of two safe primes, with the public key being their product $N$.

2. $\mathtt{enc}_N(M;r)$ encrypts a message $M \in \pm N$ using randomness $r \in \mathbb{Z}_N^*$ and Paillier public key $N$. This produces a ciphertext $C \in \mathbb{Z}_{N^2}^*$. Encryption checks that $\gcd(r, N) = 1$ (and raises an error if not), and then computes

$$\mathtt{enc}_N(M;r) := (1 + M \cdot N) \cdot r^N \bmod N^2$$

   We also provide a function $\mathtt{enc}_N(M)$ that samples uniform $r \in \mathbb{Z}_N^*$ and returns $\mathtt{enc}_N(M;r)$.

   Functions $\mathtt{enc}_{\mathsf{sk}}^{\mathsf{crt}}(M;r)$ and $\mathtt{enc}_{\mathsf{sk}}^{\mathsf{crt}}(M)$ are analogous but achieve better performance by using the technique from Section 2.2 when the factorization of $N$ is known.

3. $\mathtt{dec}_{\mathsf{sk}}(C)$ decrypts a ciphertext $C \in \mathbb{Z}_{N^2}^*$ to a plaintext $M \in \pm N$.

4. $C_1 \oplus C_2$ denotes homomorphic addition of ciphertexts $C_1, C_2 \in \mathbb{Z}_{N^2}^*$ encrypted under the same Paillier public key $N$. It is computed as $C_1 \oplus C_2 = C_1 \cdot C_2 \bmod N^2$. Note that $\mathtt{dec}(C_1 \oplus C_2) = [\mathtt{dec}(C_1) + \mathtt{dec}(C_2) \bmod N]$.

5. $k \odot C$ denotes homomorphic multiplication of a ciphertext $C \in \mathbb{Z}_{N^2}^*$ by $k \in \mathbb{Z}$. It is computed as $k \odot C = C^k \bmod N^2$. Note that $\mathtt{dec}(k \odot C) = [k \cdot \mathtt{dec}(C) \bmod N]$.

### 2.4 Speeding up Fixed-Based Multiexponentiation Using Preprocessing

Execution of the protocol involves many computations of the form $s^x t^y \bmod N$, where $s, t, N$ are fixed (and known in advance) but the exponents $x, y$ vary. For the purposes of this section we view this as a multiexponentiation with respect to the bases $s, t$ in a generic group, and so ignore $N$. Efficiency of these multiexponentiations can be improved by using *one-time preprocessing* to generate a small amount of state that is used to speed up subsequent computations.

Say $-\ell_x < x < \ell_x$ and $-\ell_y < y < \ell_y$, where typically $\ell_x, \ell_y$ are powers of 2. In describing the algorithm, we assume $x, y$ are nonnegative; we can handle negative exponents by also precomputing $s^{-\ell_x}$ and $t^{-\ell_y}$ and then, e.g., when $x$ is negative write $s^x t^y = \left(s^{-\ell_x}\right) \cdot \left(s^{x+\ell_x} t^y\right)$ with $x + \ell_x > 0$. The algorithm is parameterized by a value $B$ which is also typically a power of 2 (in practice, taking $B \in \{2^4, 2^8\}$ is a good choice); it stores $T_B \approx (\log \ell_x + \log \ell_y)/\lg B$ group elements and requires $\approx B + T_B$ group operations to compute a multiexponentiation. See Algorithm 3.

**Algorithm 3** Computing $s^x t^y$; parameterized by $B \geq 2$; let $k'_x = \lceil |x|/\lg B \rceil$, $k'_y = \lceil |y|/\lg B \rceil$

```
in preprocessing step, compute s_i := s^{B^i} for i ∈ {0, ..., k'_x − 1}
in a preprocessing step, compute t_i := t^{B^i} for i ∈ {0, ..., k'_y − 1}
let x_{k'_x−1} ··· x_0 and y_{k'_y−1} ··· y_0 be the base-B representations of x and y, respectively
res := 1, tmp := 1
for b = B − 1, ..., 1 do
    for all i such that y_i = b do
        tmp := tmp · t_i
    for all i such that x_i = b do
        tmp := tmp · s_i
    res := res · tmp
return res
```

We refer to $(\{s_i\}_{i=0}^{k'_x}, \{t_i\}_{i=0}^{k'_y})$ as a *table* $T_i$. Precomputation of a table is only done once, so the efficiency of doing so is not critical; nevertheless, for completeness, we describe an algorithm for computing the $\{s_i\}_{i\in\{0,\ldots,k'_x-1\}}$. (The same algorithm can be used for computing the $\{t_i\}$ as well.)

**Algorithm 4** Computing $\{s_i\}_{i=0}^{k}$, where $s_i = s^{B^i}$

1: $s_0 := s$
2: **for** $i = 1, \ldots, k$ **do**
3: $\quad s_i := s_{i-1}^B$

Assuming $B$ is a power of 2, the exponentiation in line 3 requires $\lg B$ squarings; the algorithm thus uses only $k \lg B$ squarings overall.

### 2.5 Security Parameters

The protocol relies on several user-defined parameters that determine its security. Note these do not include the curve order $q$, which is fixed by the underlying signature scheme rather than by the threshold protocol itself. We let $\lambda$ denote the bit length of the curve order (so $2^\lambda \le q < 2^{\lambda+1}$) and assume $\lambda \ge 256$ (which is the case for the signature schemes we support).

The security parameters of the protocol are denoted collectively by $L = (\kappa, \varepsilon, \ell, \ell', m, Q)$. These parameters are used in the following ways:

- $\kappa$ is set to $\log_2 q$, bit-length of curve prime order. E.g. for secp256k1, $\kappa = 256$.
- $n_{\mathtt{rsa}}$ is minimal length of the Paillier public key, private key of which has $\kappa_{\mathtt{rsa}}$ bits. It's chosen to match target security level. E.g. for 128-bits security, NIST recommends [1] 3072 bits public key.
- $\kappa_{\mathtt{rsa}}$ is a bit-length of the primes used for Paillier private keys. Product of two such primes must be at least $n_{\mathtt{rsa}}$ bits long.
- $\ell, \ell'$ correspond to bounds on the ranges of certain plaintexts that are encrypted, while $\varepsilon$ is a slack parameter. (Honest parties choose plaintexts in a range determined by $\ell$ or $\ell'$; the zero-knowledge proofs, however, only prove that a party chose plaintexts in a range determined by $\ell + \varepsilon$ or $\ell' + \varepsilon$.)
- $m$ denotes the number of iterations of some underlying zero-knowledge protocol to run; the soundness error will be $2^{-m}$.

For correctness, we require $\ell \ge \lambda$, $\epsilon \ge 8 + \log Q$, and $\ell' \le 8\kappa$. We also recommend $Q = 2^m$ since it can only hurt efficiency (while not improving security) otherwise.

#### 2.5.1 Security Guidelines

Let $s \le 256$ be a statistical security parameter, so the goal is to achieve roughly $2^{-s}$ "privacy loss" in one execution of the protocol. Parameters can be set using the following guidelines:

- $\kappa$ should be set based on current estimates regarding hardness of factoring. Setting $\kappa = 384$ (so moduli are 3072-bits long) matches NIST recommendations for achieving 128-bit computational security, which is consistent with the security obtained by using $\lambda = 256$.

- $\ell$ needs to be set such that $2^{\ell+1} \geq q$; setting $\ell \geq \lambda$ ensures this. Some of the zero-knowledge proofs have privacy loss and soundness error at least $2^{-\ell}$, but since $\ell = \lambda \geq s$ that is fine.
- $Q$ and $m$ determine the soundness error of several of the zero-knowledge proofs, with some of the proofs having soundness error at least $1/Q$ and others having soundness error at least $2^{-m}$. It thus makes sense to set $Q = 2^m$ (as recommended above). Setting $Q = 2^{128}$ (and $m = 128$) suffices for 128-bit security. **Note:** while it is possible to increase $Q$ without any significant direct impact on efficiency, increasing $Q$ requires increasing $\epsilon, \ell'$, which does impact efficiency.
- $\epsilon$ affects both the completeness error and the privacy loss of several of the zero-knowledge proofs. Since some proofs have privacy loss at least $4Q/2^{\epsilon}$, this requires $\epsilon \geq 2 + s + \log Q$.
- $\ell'$ needs to be set large enough so that adding noise from $\pm 2^{\ell'}$ statistically hides a $(2\lambda+\epsilon)$-bit value. This requires $\ell' \geq 2\lambda + \epsilon + s$.

If the above guidelines are used, the interaction between one honest party and one malicious party during an execution of the signing protocol has privacy loss upper-bounded by $8 \cdot 2^{-s}$ (this accounts for all the zero-knowledge proofs as well as the noise used for statistically hiding different values). If we assume $t-1$ malicious parties and $n-t+1$ honest parties, the overall privacy loss in an execution of the protocol is at most $8 \cdot (n-t+1) \cdot (t-1) \cdot 2^{-s}$.

### 2.6 Unambiguous encoding

We require unambiguous encode function

$$\texttt{Encode}_{\texttt{tag}} : X_1 \times X_2 \times \cdots \times X_n \to \mathbb{B}^*$$

which takes arguments $x_1 \in X_1, \ldots, x_n \in X_n$ and produces their unambiguous bytes encoding. $X_i$ can be any domain that has bytes representation. E.g. if we encode an integer, then $X_i$ is set of all integers, or if we encode a UTF-8 string, then $X_i$ is set of all valid UTF-8 strings.

Encoding function should satisfy these properties:

- **Injectivity.** $\texttt{Encode}_{\texttt{tag}}(x_1, \ldots, x_n)$ has to be a one-to-one function, i.e. for each set of arguments, `Encode` returns their byte-repesentation which is unique to those arguments.

$$\begin{aligned}&\forall x_1, x_1' \in X_1^2, \ldots, x_n, x_n' \in X_n^2,\\ &(x_1 \neq x_1' \vee \cdots \vee x_n \neq x_n') \implies \texttt{Encode}_{\texttt{tag}}(x_1, \ldots, x_n) \neq \texttt{Encode}_{\texttt{tag}}(x_1', \ldots, x_n')\end{aligned}$$

- **Tag-dependence.** Tag should be unambiguously encoded into the output, so $\texttt{Encode}_{\texttt{tag}}(\ldots) \neq \texttt{Encode}_{\texttt{tag}'}(\ldots)$ for any $\text{tag} \neq \text{tag}'$.
- **Platform-independence.** Encoding should be the same regardless of platform or machine on which it's executed.

## 3 Zero-Knowledge Proofs

In this section we describe the various zero-knowledge proofs that are used as sub-routines in the protocol. In each case we first describe an interactive version of the proof; we then describe how we implement a non-interactive version of the proof using the Fiat-Shamir transform.

### 3.1 Implementation guidelines

ZK proofs are defined as set of functions, for each function we explicitly define:

- Input arguments and their domains

  Implementation MUST validate input arguments. For instance, if there's an argument $K \in \mathbb{Z}_N^*$, implementation must verify that $0 \leq K < N \wedge \gcd(K, N) = 1$.

  To avoid reduntant validation of input arguments, we typically relax their domain. For instance, the proof might require that $R = (N, s, t) \in (\mathbb{Z}, \mathbb{Z}_N^*, \mathbb{Z}_N^*)$, however, $R$ is in fact validated during the auxiliary information generation protocol (via $\Pi^{\mathtt{prm}}$), so we do not require other proofs to validate $R$, and thus we only require that $R = (N, s, t) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$.

- Output and its domain

  Domain of function output bears informational purpose. Sometimes it's not accurate enough, e.g. we write that output is in $\mathbb{Z}$ where in fact we could narrow it to $\mathbb{Z}_N^*$. The reason is simply that it takes time to determine output domain, and it doesn't have much of benefit to have it specified.

- Body

  Function body is as close to actual implementation code as possible. Input arguments validation must be done before any of body instructions is executed.

### 3.2 Non-Interactive Challenge Derivation

In order to implement Fiat-Shamir transform correctly, we need to deterministically and securely derive a challenge from the inputs. To do so, we define a function:

$$\mathtt{ChallengeNI}_{\mathtt{tag}} : X_1 \times \cdots \times X_n \to Y_1 \times \cdots \times Y_k$$

which takes proof inputs and outputs uniformelly distributed challenges (number of challenges to be sampled depends on the proof).

To sample the challenges, we first construct the cryptographic pseudo-random generator $G$:

$$\begin{aligned} G = & H(\mathtt{Encode}_{\mathtt{tag}}(0, x_1, \ldots, x_n)) \\ & \| \; H(\mathtt{Encode}_{\mathtt{tag}}(1, x_1, \ldots, x_n)) \\ & \| \; H(\mathtt{Encode}_{\mathtt{tag}}(2, x_1, \ldots, x_n)) \\ & \| \; \ldots \end{aligned}$$

And then we use pseudo-random stream $G$ to deterministically sample $y_1 \leftarrow Y_1, \ldots, y_k \leftarrow Y_k$.
**Note:** sometimes we write $\mathtt{ChallengeNI}^L_{\mathtt{tag}}(\ldots)$ which is equivalent to $\mathtt{ChallengeNI}_{\mathtt{tag}}(L, \ldots)$.

### 3.3 $\Pi^{\mathtt{enc}}$: Paillier Encryption in Range

We assume the prover and verifier agree on shared state $\mathsf{state}$, auxiliary data $R_j = (N_j, s_j, t_j)$ with $s_j, t_j \in \mathbb{Z}_{N_j}^*$, and a security level $L$. The prover and verifier have common input $(N_i, K)$, and the prover additionally has secret input $(k, \rho)$ such that $k \in \pm 2^\ell$ and $K = \mathtt{enc}_{N_i}(k; \rho)$. In all the

cases where this proof is used in the protocol, the prover knows the factorization of $N_i$ (and thus knows $\mathsf{sk}_i$) and the verifier knows the factorization of $N_j$ (and thus knows $\mathsf{sk}_j$).

### 3.3.1 Interactive Version of the Proof

$\mathtt{Commit}^{L}_{\mathtt{enc}}(R_j, N_i; k, [, \mathsf{sk}_i]) \to ((S, A, C); (\alpha, \mu, r, \gamma))$

**Inputs:**

- security level $L = (\ell, \varepsilon, \dots)$
- auxilary data $R_j = (N_j, s_j, t_j) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$
- public encryption key $N_i \in \mathbb{Z}$ and, if known, corresponding secret key $\mathsf{sk}_i$
- secret plaintext $k \in \mathbb{Z}$

**Outputs:**

- public commitment $(S, A, C) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$,
- secret nonce $(\alpha, \mu, r, \gamma) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$

1: $\alpha \leftarrow \pm 2^{\ell+\varepsilon}$
2: $\mu \leftarrow \pm(2^{\ell} \cdot N_j)$
3: $r \leftarrow \mathbb{Z}^*_{N_i}$
4: $\gamma \leftarrow \pm(2^{\ell+\varepsilon} \cdot N_j)$
5: $S = s_j^k t_j^\mu \bmod N_j$ ▷ use precomputed fixed-base multiexponentiation table
6: $A = \mathsf{enc}_{N_i}(\alpha; r)$ (this is computed as $\mathsf{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(\alpha; r)$ if $\mathsf{sk}_i$ is known)
7: $C = s_j^\alpha t_j^\gamma \bmod N_j$ ▷ use precomputed fixed-base multiexponentiation table
8: **return** $((S, A, C); (\alpha, \mu, r, \gamma))$

$\mathtt{Challenge}_{\mathtt{enc}}() \to e$

**Inputs:** none

**Outputs:** $e \in \pm q$

1: Sample $e \leftarrow \pm q$
2: **return** $e$

$\mathtt{Prove}_{\mathtt{enc}}((N_i, K), e; \rho, (\alpha, \mu, r, \gamma)) \to (z_1, z_2, z_3)$

**Inputs:**

- public data $(N_i, K) \in (\mathbb{Z}, \mathbb{Z})$
- challenge $e \in \mathbb{Z}$
- secret nonce $\rho \in \mathbb{Z}$
- local secret commitment none $(\alpha, \mu, r, \gamma) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$

**Outputs:** $(z_1, z_2, z_3) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$

1: $z_1 = \alpha + ek$
2: $z_2 = r \cdot \rho^e \bmod N_i$
3: $z_3 = \gamma + e\mu$
4: **return** $(z_1, z_2, z_3)$

$\mathtt{Verify}^{L}_{\mathtt{enc}}(R_j, (N_i, K), (S, A, C), e, (z_1, z_2, z_3))$

**Inputs:**

– security level $L = (\varepsilon, \ell, \dots)$
– auxilary data $R_j = (N_j, s_j, t_j) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$,
– public data $(N_i, K) \in (\mathbb{Z}, \mathbb{Z}^*_{N_i})$,
– commitment $(S, A, C) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$,
– challenge $e \in \mathbb{Z}$,
– proof $(z_1, z_2, z_3) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$

**Outputs:** aborts if proof is invalid

1: $K \stackrel{?}{\in} \mathbb{Z}^*_{N_i^2}$
2: $S \stackrel{?}{\in} \mathbb{Z}^*_{N_j}$
3: $A \stackrel{?}{\in} \mathbb{Z}^*_{N_i^2}$
4: $C \stackrel{?}{\in} \mathbb{Z}^*_{N_j}$
5: $z_1 \stackrel{?}{\in} \pm 2^{\ell+\varepsilon}$.
6: $z_3 \stackrel{?}{\in} \pm(N_j \cdot 2^{\ell+\varepsilon+1})$
7: $A \oplus (e \odot K) \stackrel{?}{=} \mathtt{enc}_{N_i}(z_1; z_2) \bmod N_i^2$
8: $s_j^{z_1} t_j^{z_3} \stackrel{?}{=} C \cdot S^e \bmod N_j$ ▷ use precomputed multiexp table to compute left part of equation

### 3.3.2 Non-Interactive Version of the Proof

$\mathtt{ProveNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K); k, \rho[, \mathsf{sk}_i]) \to ((S, A, C); (z_1, z_2, z_3))$

**Inputs:**

– security level $L = (Q, \dots)$,
– shared state $\mathsf{state} \in \mathbb{B}^*$,
– auxilary data $R_j$,
– public encryption key $N_i \in \mathbb{Z}$ and, if known, corresponding secret key $\mathsf{sk}_i$,
– public ciphertext $K \in \mathbb{Z}$,
– secret plaintext $k \in \mathbb{Z}$ and secret nonce $\rho \in \mathbb{Z}$

**Outputs:**

– public commitment $(S, A, C) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$,
– proof $(z_1, z_2, z_3) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$

1: $((S, A, C); (\alpha, \mu, r, \gamma)) \leftarrow \mathtt{Commit}^L_{\mathsf{enc}}(R_j, N_i; k[, \mathsf{sk}_i])$ ▷ generate commitment
2: $e \in \pm q = \mathtt{ChallengeNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), (S, A, C))$ ▷ deterministically derive challenge
3: $(z_1, z_2, z_3) = \mathtt{Prove}_{\mathsf{enc}}((N_i, K), e; \rho, (\alpha, \mu, r, \gamma))$
4: **return** $((S, A, C), (z_1, z_2, z_3))$

$\mathtt{VerifyNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), ((S, A, C), (z_1, z_2, z_3)))$

**Inputs:**

– security level $L = (Q, \dots)$,
– shared state $\mathsf{state} \in \mathbb{B}^*$,

- auxilary data $R_j$,
- public data:
  - encryption key $N_i \in \mathbb{Z}$,
  - ciphertext $K \in \mathbb{Z}$,
- non-interactive proof $((S, A, C), (z_1, z_2, z_2)) \in ((\mathbb{Z}, \mathbb{Z}, \mathbb{Z}), (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}))$

**Outputs:** aborts if proof is invalid

1: $e \in \pm q = \mathtt{ChallengeNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), (S, A, C))$ ▷ deterministically derive challenge
2: Assert $\mathtt{Verify}^L_{\mathsf{enc}}((N_i, K), (S, A, C), e, (z_1, z_2, z_3))$

### 3.4 $\Pi^{\mathtt{aff\text{-}g}}$: Paillier Affine Operation with Group Commitment in Range

We assume the prover and verifier agree on shared state $\mathsf{state}$, auxiliary data $R_j = (\hat{N}_j, s_j, t_j)$ with $s_j, t_j \in \mathbb{Z}^*_{N_j}$, an elliptic curve $\mathbb{E}$ of prime order $q$ with generator $G$, and a security level $L$. For this proof, the prover and verifier have common input $(N_j, N_i, C, D, Y, X)$ where $C, D \in \mathbb{Z}^*_{N_j^2}$, $Y \in \mathbb{Z}^*_{N_i^2}$, and $X \in \mathbb{E}$, and the prover additionally has secret input $(x, y, \rho, \rho_y)$ such that $x \in \pm 2^{\ell}$, $y \in \pm 2^{\ell'}$, $\rho \in \mathbb{Z}^*_{N_j}$, $\rho_y \in \mathbb{Z}^*_{N_i}$, $D = (x \odot C) \oplus \mathtt{enc}_{N_j}(y; \rho)$, $Y = \mathtt{enc}_{N_i}(y; \rho_y)$, and $X = x \cdot G$. In all the cases where this proof is used in the protocol, the prover knows the factorization of $N_i$ (and hence knows $\mathsf{sk}_i$) and the verifier knows the factorization of $N_j$ (and hence knows $\mathsf{sk}_j$).

Proof guarantees: $x \in \pm 2^{\ell+\varepsilon}$ and $y \in \pm 2^{\ell'+\varepsilon}$

#### 3.4.1 Interactive Version of the Proof

$\mathtt{Commit}^L_{\mathtt{aff\text{-}g}}(R_j, (N_i, C); (x, y)) \to ((A, B_x, B_y, E, S, F, T); (\alpha, \beta, r, r_y, \gamma, \delta, m, \mu))$

**Inputs:**

- security level $L = (\ell, \varepsilon, \dots)$,
- auxilary data $R_j = (N_j, s_j, t_j) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$,
- public data $N_i, C \in \mathbb{Z}, \mathbb{Z}$
- secret data $x, y \in \mathbb{Z}, \mathbb{Z}$

**Outputs:**

- public commitment $(A, B_x, B_y, E, S, F, T) \in (\mathbb{Z}, \mathbb{E}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$,
- proof $(\alpha, \beta, r, r_y, \gamma, \delta, m, \mu) \in \mathbb{Z}^8$

1: Sample:
$$\alpha \leftarrow \pm 2^{\ell+\varepsilon}, \quad r \leftarrow \mathbb{Z}^*_{N_j}, \quad \gamma, \delta \leftarrow \pm(2^{\ell+\varepsilon} \cdot \hat{N}_j)^2$$
$$\beta \leftarrow \pm 2^{\ell'+\varepsilon}, \quad r_y \leftarrow \mathbb{Z}^*_{N_i}, \quad m, \mu \leftarrow \pm(2^{\ell} \cdot \hat{N}_j)^2.$$
2: $A = (\alpha \odot C) \oplus \mathtt{enc}_{N_j}(\beta; r)$
3: $B_x = \alpha \cdot G$
4: $B_y = \mathtt{enc}_{N_i}(\beta; r_y)$ ▷ this is computed as $\mathtt{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(\beta; r_y)$ if $\mathsf{sk}_i$ is known
5: $E = s_j^{\alpha} t_j^{\gamma} \bmod \hat{N}_j$ ▷ this and below can be computed via pregenerated table for fixed-base multiexp
6: $S = s_j^{x} t_j^{m} \bmod \hat{N}_j$
7: $F = s_j^{\beta} t_j^{\delta} \bmod \hat{N}_j$

8: $T = s_j^y t_j^\mu \bmod \hat{N}_j$
9: **return** $((A, B_x, B_y, E, S, F, T); (\alpha, \beta, r, r_y, \gamma, \delta, m, \mu))$

$\texttt{Challenge}_{\texttt{aff-g}}() \to e$

**Inputs:** none

**Outputs:** challenge $e \in \pm q$

1: Sample $e \leftarrow \pm q$ ▷ $q = |\mathbb{E}|$
2: **return** $e$

$\texttt{Prove}_{\texttt{aff-g}}((N_j, N_i, C, D, Y, X), e; (x, y, \rho, \rho_y), (\alpha, \beta, r, r_y, \gamma, \delta, m, \mu)) \to (z_1, z_2, z_3, z_4, w, w_y)$

**Inputs:**

- public data $(N_j, N_i, C, D, Y, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{E})$
- challenge $e \in \mathbb{Z}$
- secret data $(x, y, \rho, \rho_y) \in \mathbb{Z}^4$
- secret commitment nonce $(\alpha, \beta, r, r_y, \gamma, \delta, m, \mu) \in \mathbb{Z}^8$

**Outputs:** proof $(z_1, z_2, z_3, z_4, w, w_y) \in \mathbb{Z}^6$

1: $z_1 = \alpha + ex$
2: $z_2 = \beta + ey$
3: $z_3 = \gamma + em$
4: $z_4 = \delta + e\mu$
5: $w = r \cdot \rho^e \bmod N_j$
6: $w_y = r_y \cdot \rho_y^e \bmod N_i$
7: **return** $(z_1, z_2, z_3, z_4, w, w_y)$

$\texttt{Verify}^L_{\texttt{aff-g}}(R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T), e, (z_1, z_2, z_3, z_4, w, w_y); [\mathsf{sk}_j])$

**Inputs:**

- security level $L = (\ell, \ell', \varepsilon, \dots)$
- auxilary data $R_j = (s, t, \dots) \in (\mathbb{Z}, \mathbb{Z}, \dots)$
- public data $(N_j, N_i, C, D, Y, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{E})$
- commiment $(A, B_x, B_y, E, S, F, T) \in (\mathbb{Z}, \mathbb{E}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$
- challenge $e \in \mathbb{Z}$
- proof $(z_1, z_2, z_3, z_4, w, w_y) \in \mathbb{Z}^6$
- if known, private key $\mathsf{sk}_j$, corresponding to $N_j$

**Outputs:** aborts if proof is invalid

Verify public data:
1: $C \overset{?}{\in} \mathbb{Z}^*_{N_j^2}$
2: $D \overset{?}{\in} \mathbb{Z}^*_{N_j^2}$
3: $Y \overset{?}{\in} \mathbb{Z}^*_{N_i^2}$
Verify commitment:
4: $A \overset{?}{\in} \mathbb{Z}^*_{N_j^2}$
5: $B_y \overset{?}{\in} \mathbb{Z}^*_{N_i^2}$

6: $E \overset{?}{\in} \mathbb{Z}^*_{\hat{N}_j}$

7: $S \overset{?}{\in} \mathbb{Z}^*_{\hat{N}_j}$

8: $F \overset{?}{\in} \mathbb{Z}^*_{\hat{N}_j}$

9: $T \overset{?}{\in} \mathbb{Z}^*_{\hat{N}_j}$

Verify statement:

10: $A \oplus (e \odot D) \overset{?}{=} (z_1 \odot C) \oplus \mathsf{enc}^{\mathsf{crt}}_{\mathsf{sk}_j}(z_2; w) \bmod N_j^2$

11: $z_1 \cdot G \overset{?}{=} B_x + e \cdot X$

12: $B_y \oplus (e \odot Y) \overset{?}{=} \mathsf{enc}_{N_i}(z_2; w_y) \bmod N_i^2$

13: $s_j^{z_1} t_j^{z_3} \overset{?}{=} E \cdot S^e \bmod \hat{N}_j$ ▷ left part is computed via pregenerated fixed-base mulitexp table

14: $s_j^{z_2} t_j^{z_4} \overset{?}{=} F \cdot T^e \bmod \hat{N}_j$ ▷ right part is computed via CRT exponentiation

15: $z_1 \overset{?}{\in} \pm 2^{\ell+\varepsilon}$

16: $z_2 \overset{?}{\in} \pm 2^{\ell'+\varepsilon}$

### 3.4.2 Non-Interactive Version of the Proof

$\mathtt{ProveNI}^L_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X); (x, y, \rho, \rho_y)) \to ((A, B_x, B_y, E, S, F, T), (z_1, z_2, z_3, z_4, w, w_y))$

**Inputs:**

- security level $L$,
- shared state $\mathsf{state} \in \mathbb{B}^*$,
- auxiliary data $R_j$,
- public data $(N_j, N_i, C, D, Y, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{E})$,
- secret data $(x, y, \rho, \rho_y) \in \mathbb{Z}^4$

**Outputs:**

- public commitment $(A, B_x, B_y, E, S, F, T) \in (\mathbb{Z}, \mathbb{E}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$
- proof $(z_1, z_2, z_3, z_4, w, w_y) \in \mathbb{Z}^6$

1: $(A, B_x, B_y, E, S, F, T); (\alpha, \beta, r, r_y, \gamma, \delta, m, \mu) \leftarrow \mathtt{Commit}^L_{\mathtt{aff\text{-}g}}(R_j, (N_i, C); (x, y))$

2: $e \in \pm q = \mathtt{ChallengeNI}^L_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T))$ ▷ $q = |\mathbb{E}|$

3: $(z_1, z_2, z_3, z_4, w, w_y) = \mathtt{Prove}_{\mathtt{aff\text{-}g}}((N_j, N_i, C, D, Y, X), e; (x, y, \rho, \rho_y), (\alpha, \beta, r, r_y, \gamma, \delta, m, \mu))$

4: **return** $((A, B_x, B_y, E, S, F, T), (z_1, z_2, z_3, z_4, w, w_y))$

$\mathtt{VerifyNI}^L_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), ((A, B_x, B_y, E, S, F, T), (z_1, z_2, z_3, z_4, w, w_y)); [\mathsf{sk}_j])$

**Inputs:**

- security level $L$,
- auxiliary data $R_j$,
- public data $(N_j, N_i, C, D, Y, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{E})$,
- non-interactive proof consisting of:
  - commitment $(A, B_x, B_y, E, S, F, T) \in (\mathbb{Z}, \mathbb{E}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$

– proof $(z_1, z_2, z_3, z_4, w, w_y) \in \mathbb{Z}^6$

– if known, secret key $\mathsf{sk}_j$, corresponding to $N_j$

**Outputs:** aborts if proof is invalid

1: $e \in \pm q = \texttt{ChallengeNI}^L_{\texttt{aff-g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T))$ ▷ $q = |\mathbb{E}|$

2: Assert $\texttt{Verify}^L_{\texttt{aff-g}}(R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T), e, (z_1, z_2, z_3, z_4, w, w_y); [\mathsf{sk}_j])$

## 3.5 $\Pi^{\texttt{mod}}$: Paillier-Blum Modulus

The prover and verifier agree on shared state $\mathsf{state}$ and a security level $L$ (which determines $m$). For this proof, the prover and verifier have common input $N$, and the prover additionally has as secret input primes $p, q = 3 \bmod 4$ such that $N = pq$.

### 3.5.1 Utility function

Below we describe additional functions that are used within the proof.

$\texttt{blum\_sqrt}(x, p, q, N) \to y$

**Inputs:**

– $x$ is a quadratic residue in $\mathbb{Z}_N$,

– primes $p, q$ that are congruent to 3 modulo 4,

– $N = p \cdot q$

**Outputs:** $y$ such that $y^2 = x \bmod N$

**Note:** caller must guarantee that all inputs are correct, so the implementation of this function is not required to validate the arguments. If inputs are not valid, the output is undefined.

1: $e = ((p-1) \cdot (q-1) + 4)/8$ ▷ Described in [4], p. 75, Fact 2.160

2: $y = x^e \bmod N$

3: **return** $y$

$\texttt{find\_residue}(y, w, p, q, N) \to (a, b, y')$ or $\perp$

Finds $a, b \in \{0,1\}^2$ such that $y' = (-1)^a w^b y$ is a quadratic residue in $\mathbb{Z}_N$.

**Inputs:**

– $y \in \mathbb{Z}$,

– primes $p, q \in \mathbb{Z}, \mathbb{Z}$ that are congruent to 3 modulo 4,

– $N = p \cdot q$,

– $w$ with Jacobi symbol $\left(\frac{w}{N}\right) = -1$

**Outputs:** $(a, b, y')$ such that $y' = (-1)^a w^b y$ is a quadratic residue in $\mathbb{Z}_N$

**Note:** caller must guarantee that all inputs are correct, so the implementation of this function is not required to validate the arguments. If inputs are not valid, the output is undefined.

1: Calculate $\left(\frac{y}{p}\right), \left(\frac{y}{q}\right)$

2: **if** $\left(\frac{y}{p}\right) = 1 \wedge \left(\frac{y}{q}\right) = 1$ **then return** $(a = 0, b = 0, y' = y)$

3: **if** $\left(\frac{y}{p}\right) = -1 \wedge \left(\frac{y}{q}\right) = -1$ **then return** $(a = 1, b = 0, y' = n - y)$

4: Calculate $\left(\frac{wy}{p}\right), \left(\frac{wy}{q}\right)$
5: **if** $\left(\frac{wy}{p}\right) = 1 \wedge \left(\frac{wy}{q}\right) = 1$ **then return** $(a = 0, b = 1, y' = wy)$
6: **if** $\left(\frac{wy}{p}\right) = -1 \wedge \left(\frac{wy}{q}\right) = -1$ **then return** $(a = 1, b = 1, y' = n - wy)$
7: **return** $\perp$

### 3.5.2 Interactive Version of the Proof

$\texttt{Commit}_{\texttt{mod}}(N) \to w$

**Inputs:** public key $N \in \mathbb{Z}$

**Outputs:** $w \in \mathbb{Z}_N^*$

1: Sample $w \leftarrow \mathbb{Z}_N^*$ with Jacobi symbol $\left(\frac{w}{N}\right) = -1$ ▷ can be done with rejection sampling: sample $w \leftarrow \mathbb{Z}_N^*$, output $w$ if $\left(\frac{w}{N}\right) = -1$, repeat otherwise
2: **return** $w$

$\texttt{Challenge}_{\texttt{mod}}^{L}(N) \to \vec{y}$

**Inputs:**

- security level $L = (m, \dots)$,
- public key $N \in \mathbb{Z}$

**Outputs:** $\vec{y} \in (\mathbb{Z}_N^*)^m$

1: Sample $y_i \leftarrow \mathbb{Z}_N^*$ for $i \in [m]$
2: **return** $\vec{y} = \{y_i\}_{i \in [m]}$

$\texttt{Prove}_{\texttt{mod}}^{L}(N, \vec{y}, w; (p, q)) \to \{(x_i, a_i, b_i, z_i)\}_{i \in [m]}$

**Inputs:**

- security level $L = (m, \dots)$,
- public key $N \in \mathbb{Z}$,
- challenge $\vec{y} = \{y_i\}_{i \in [m]} \in \mathbb{Z}^m$,
- commitment $w \in \mathbb{Z}$,
- secret data: primes $p, q \in \mathbb{Z}, \mathbb{Z}$ that are congruent 3 modulo 4 such that $N = p \cdot q$
  **Note:** $p, q$ are assumed to be valid, so the implementation does not need to verify their correctness. In the protocol, it's the prover who generates the primes using appropriate procedure. If invalid $p, q$ are given, then resulting proof is invalid.

**Outputs:** $\{(x_i, a_i, b_i, z_i)\}_{i \in [m]} \in \{(\mathbb{Z}, \{0,1\}, \{0,1\}, \mathbb{Z})\}^m$

1: $N' = N^{-1} \bmod \phi(N)$
2: **for** $i \in [m]$ **do**
3: $\quad (a_i, b_i, y_i') = \texttt{find\_residue}(y_i, w, p, q, N)$
4: $\quad x_i = \texttt{blum\_sqrt}(\texttt{blum\_sqrt}(y_i', p, q, N), p, q, N)$ ▷ $x_i$ is principal 4th root of $y_i'$ modulo $N$
5: $\quad z_i = y_i^{N'} \bmod N$
6: **return** $\{(x_i, a_i, b_i, z_i)\}_{i \in [m]}$

$\texttt{Verify}^{L}_{\texttt{mod}}(N, w, \{y_i\}_{i\in[m]}, \{(x_i, a_i, b_i, z_i)\}_{i\in[m]})$

**Inputs:**

- security level $L = (m, \dots)$,
- public data $N \in \mathbb{Z}$,
- commitment $w \in \mathbb{Z}_N^*$,
- challenge $\{y_i\}_{i\in[m]} \in (\mathbb{Z}_N^*)^m$,
- proof $\{(x_i, a_i, b_i, z_i)\}_{i\in[m]} \in \{(\mathbb{Z}, \{0,1\}, \{0,1\}, \mathbb{Z})\}^m$

**Outputs:** aborts if proof is invalid

1: $N \overset{?}{=} 1 \bmod 2$
2: $\mathsf{IsPrime}(N) \overset{?}{=} \texttt{false}$
3: $w \overset{?}{\in} \mathbb{Z}_N^*$

4: **for** $i \in [m]$ **do**
5: $\quad x_i \overset{?}{\in} \mathbb{Z}_N^*$
6: $\quad z_i \overset{?}{\in} \mathbb{Z}_N^*$
7: $\quad z_i^N \overset{?}{=} y_i \bmod N$
8: $\quad (a_i, b_i) \overset{?}{\in} \{0,1\}^2$
9: $\quad x_i^4 \overset{?}{=} (-1)^{a_i} w^{b_i} y_i \bmod N$

### 3.5.3 Non-Interactive Version of the Proof

$\texttt{ProveNI}^{L}_{\texttt{mod}}(\mathsf{state}, N; (p, q)) \to (w, \{(x_i, a_i, b_i, z_i)\}_{i\in[m]})$

**Inputs:**

- security level $L = (m, \dots)$,
- shared $\mathsf{state} \in \mathbb{B}^*$,
- public data $N \in \mathbb{Z}$,
- secret data: primes $p, q \in \mathbb{Z}, \mathbb{Z}$ that are congruent 3 modulo 4 such that $N = p \cdot q$
  **Note:** $p, q$ are assumed to be valid, so the implementation does not need to verify their correctness. In the protocol, it's the prover who generates the primes using appropriate procedure. If invalid $p, q$ are given, then resulting proof is invalid.

**Outputs:**

- challenge $w \in \mathbb{Z}_N^*$,
- proof $\{(x_i, a_i, b_i, z_i)\}_{i\in[m]} \in \{(\mathbb{Z}, \{0,1\}, \{0,1\}, \mathbb{Z})\}^m$

1: $w \leftarrow \texttt{Commit}_{\texttt{mod}}(N)$
2: $\{y_i\}_{i\in[m]} = \texttt{ChallengeNI}^{L}_{\texttt{mod}}(\mathsf{state}, N, w)$
3: $\{(x_i, a_i, b_i, z_i)\}_{i\in[m]} = \texttt{Prove}^{L}_{\texttt{mod}}(N, \{y_i\}_{i\in[m]}, w; (p, q))$
4: **return** $(w, \{(x_i, a_i, b_i, z_i)\}_{i\in[m]})$

$\texttt{VerifyNI}^{L}_{\texttt{mod}}(\mathsf{state}, N, (w, \{(x_i, a_i, b_i, z_i)\}_{i\in[m]}))$

**Inputs:**

- security level $L = (m, \dots)$,

- shared $\mathsf{state} \in \mathbb{B}^*$,
- public data: $N \in \mathbb{Z}$,
- non-interactive proof:
  - challenge $w \in \mathbb{Z}_N^*$,
  - proof $\{(x_i, a_i, b_i, z_i)\}_{i\in[m]} \in \{(\mathbb{Z}, \{0,1\}, \{0,1\}, \mathbb{Z})\}^m$

**Outputs:** aborts if proof is invalid

1: $\{y_i\}_{i\in[m]} = \mathtt{ChallengeNI}^L_{\mathtt{mod}}(\mathsf{state}, N, w)$
2: Assert $\mathtt{Verify}^L_{\mathtt{mod}}(N, w, \{y_i\}_{i\in[m]}, \{(x_i, a_i, b_i, z_i)\}_{i\in[m]})$

## 3.6 $\Pi^{\mathtt{prm}}$: Ring-Pedersen Parameters

The prover and verifier agree on shared state $\mathsf{state}$ and security level $L$ (which determines $m$). For this proof, the prover and verifier have common input $(N, s, t)$ with $s, t \in \mathbb{Z}_N^*$, and the prover additionally has secret input $\lambda$ such that $s = t^\lambda \bmod N$, along with the factorization of $N$.

### 3.6.1 Interactive Version of the Proof

$\mathtt{Commit}^L_{\mathtt{prm}}(N; \phi(N)) \to (\vec{A}; \vec{a})$

**Inputs:**

- security level $L = (m, \dots)$
- public data $N \in \mathbb{Z}$
- secret data $\phi(N) \in \mathbb{Z}$

**Outputs:** $(\vec{A}; \vec{a}) \in (\mathbb{Z}_N^m; \mathbb{Z}_N^m)$ where $\vec{A}$ is sent to the verifier, and $\vec{a}$ is kept private

1: **for** i = 0, ..., m-1 **do**
2: Sample $a_i \leftarrow \mathbb{Z}_{\phi(N)}$
3: Compute $A_i = t^{a_i} \bmod N$
4: Set $\vec{A} = \{A_i\}_{i\in[m]}, \vec{a} = \{a_i\}_{i\in[m]}$
5: **return** $(\vec{A}; \vec{a})$

$\mathtt{Prove}^L_{\mathtt{prm}}(N, s, t; \phi(N), \lambda, \vec{a}) \to \vec{z}$

**Inputs:**

- security level $L = (m, \dots)$,
- public data $N, s, t \in \mathbb{Z}, \mathbb{Z}, \mathbb{Z}$,
- secret data $\phi(N), \lambda \in \mathbb{Z}, \mathbb{Z}$,
- secret commitment nonce $\vec{a} = \{a_i\}_{i\in[m]} \in \mathbb{Z}_N^m$

**Outputs:** proof $\vec{z} \in \mathbb{Z}^m$

1: Compute $z_i = a_i + e_i \cdot \lambda \bmod \phi(N)$ for all $i \in [m]$
2: **return** $\vec{z} = \{z_i\}_{i\in[m]}$

$\mathtt{Verify}^L_{\mathtt{prm}}(N, s, t, \vec{A}, \vec{e}, \vec{z})$

**Inputs:**

- security level $L = (m, \dots)$,
- public data $N, s, t \in \mathbb{Z}, \mathbb{Z}, \mathbb{Z}$

- commitment $\vec{A} = \{A_i\}_{i\in[m]} \in \mathbb{Z}^m$
- challenge $\vec{e} = \{e_i\}_{i\in[m]} \in \{0,1\}^m$
- proof $\vec{z} = \{z_i\}_{i\in[m]} \in \mathbb{Z}^m$

**Outputs:** Aborts if proof is invalid

Verify that inputs are in expected domains:

1: $s \overset{?}{\in} \mathbb{Z}_N^*$
2: $t \overset{?}{\in} \mathbb{Z}_N^*$
3: $A_i \overset{?}{\in} \mathbb{Z}_N^* \ \forall i \in [m]$
4: $z_i \overset{?}{\in} \mathbb{Z}_N \ \forall i \in [m]$

Verify statement:

5: Assert $t^{z_i} \overset{?}{=} A_i s^{e_i} \bmod N$ for all $i \in [m]$

### 3.6.2 Non-Interactive Version of the Proof

$\texttt{ProveNI}^L_{\texttt{prm}}(\mathsf{state}, N, s, t; \phi(N), \lambda) \to (\vec{A}, \vec{z})$

**Inputs:**

- security level $L = (m, \dots)$,
- shared state $\mathsf{state} \in \mathbb{B}^*$
- public data $N, s, t \in \mathbb{Z}, \mathbb{Z}, \mathbb{Z}$,
- secret data $\phi(N), \lambda \in \mathbb{Z}, \mathbb{Z}$,

**Outputs:** $\vec{A}, \vec{z} \in \mathbb{Z}_N^m, \mathbb{Z}_N^m$

1: Set $(\vec{A}; \vec{a}) \leftarrow \texttt{Commit}^L_{\texttt{prm}}(N, \phi(N))$
2: Deterministically derive a challenge $\vec{e} \in \{0,1\}^m = \texttt{ChallengeNI}^L_{\texttt{prm}}(\mathsf{state}, N, s, t, \vec{A})$
3: Compute proof $\vec{z} = \texttt{Prove}^L_{\texttt{prm}}(N, s, t, \phi(N), \lambda, \vec{a})$
4: **return** $(\vec{A}, \vec{z})$

$\texttt{VerifyNI}^L_{\texttt{prm}}(\mathsf{state}, N, s, t, (\vec{A}, \vec{z}))$

**Inputs:**

- security level $L = (m, \dots)$,
- shared state $\mathsf{state} \in \mathbb{B}^*$
- public data $N, s, t \in \mathbb{Z}, \mathbb{Z}, \mathbb{Z}$,
- commitment $\vec{A} \in \mathbb{Z}^m$, proof $\vec{z} \in \mathbb{Z}^m$

**Outputs:** Aborts if proof is invalid

1: Deterministically derive a challenge $\vec{e} \in \{0,1\}^m = \texttt{ChallengeNI}^L_{\texttt{prm}}(\mathsf{state}, N, s, t, \vec{A})$
2: Invoke $\texttt{Verify}^L_{\texttt{prm}}(N, s, t, \vec{A}, \vec{e}, \vec{z})$

## 3.7 $\Pi^{\texttt{fac}}$: No Small Factor Proof

The prover and verifier agree on shared state $\mathsf{state}$, auxiliary data $R_j = (N_j, s_j, t_j)$ with $s_j, t_j \in \mathbb{Z}^*_{N_j}$, and a security level $L$. For this proof, the prover and verifier have common input $N_i > 2^{4\ell}$, and the prover additionally has primes $p, q < 2^\ell \sqrt{N_i}$ with $N_i = pq$.

Proof guarantees that each $p, q > 2^\ell$ (assuming $2^{2\ell+\varepsilon} \approx \sqrt{N_i}$).

### 3.7.1 Interactive Version of the Proof

$\texttt{Commit}^{L}_{\texttt{fac}}(R_j, N_i) \to ((P, Q, A, B, T), (\alpha, \beta, \mu, \nu, r, x, y))$

**Inputs:**

- security level $L = (\ell, \varepsilon, \dots)$,
- auxiliary data $R_j = (N_j, s_j, t_j) \in \mathbb{Z}^3$,
- public data $N_i \in \mathbb{Z}$

**Outputs:**

- public commitment $(P, Q, A, B, T) \in \mathbb{Z}^5$,
- secret nonce $(\alpha, \beta, \mu, \nu, r, x, y) \in \mathbb{Z}^7$

1: $\alpha, \beta \leftarrow (\pm 2^{\ell+\varepsilon} \cdot \sqrt{N_i})^2$
2: $\mu, \nu \leftarrow (\pm 2^{\ell} \cdot N_j)^2$
3: $r \leftarrow \pm 2^{\ell+\varepsilon} \cdot N_i N_j$
4: $x, y \leftarrow (\pm 2^{\ell+\varepsilon} \cdot N_j)^2$

5: $P = s_j^p t_j^\mu \bmod N_j$
6: $Q = s_j^q t_j^\nu \bmod N_j$
7: $A = s_j^\alpha t_j^x \bmod N_j$
8: $B = s_j^\beta t_j^y \bmod N_j$ ▷ $P, Q, A, B$ are computed using fixed-base multiexp
9: $T = Q^\alpha t_j^r \bmod N_j$

10: **return** $((P, Q, A, B, T), (\alpha, \beta, \mu, \nu, r, x, y))$

$\texttt{Challenge}^{L}_{\texttt{fac}}() \to e$

**Inputs:** security level $L \in (\ell, \dots)$

**Outputs:** challenge $e \in \pm 2^{\ell}$

1: $e \leftarrow \pm 2^{\ell}$
2: **return** $e$

$\texttt{Prove}_{\texttt{fac}}(e; (p, q), (\alpha, \beta, \mu, \nu, r, x, y)) \to (z_1, z_2, w_1, w_2, v)$

**Inputs:**

- challenge $e \in \mathbb{Z}$,
- secret data: primes $p, q \in \mathbb{Z}^2$,
- secret commitment nonce: $(\alpha, \beta, \mu, \nu, r, x, y) \in \mathbb{Z}^7$

**Outputs:** proof $(z_1, z_2, w_1, w_2, v) \in \mathbb{Z}^5$

1: $z_1 = \alpha + ep$
2: $z_2 = \beta + eq$
3: $w_1 = x + e\mu$
4: $w_2 = y + e\nu$
5: $v = r - e\nu p$
6: **return** $(z_1, z_2, w_1, w_2, v)$

$\texttt{Verify}^{L}_{\texttt{fac}}(R_j, N_i, (P, Q, A, B, T), e, (z_1, z_2, w_1, w_2, v))$

**Inputs:**

– security level $L = (\ell, \varepsilon, \dots)$,
– auxiliary data $R_j = (N_j, s_j, t_j) \in \mathbb{Z}^3$,
– public data $N_i \in \mathbb{Z}$,
– commitment $(P, Q, A, B, T) \in \mathbb{Z}^5$,
– challenge $e \in \mathbb{Z}$,
– proof $(z_1, z_2, w_1, w_2, v) \in \mathbb{Z}^5$

**Outputs:** aborts if proof is invalid

Verify that inputs are in expected domains:

1: $P \overset{?}{\in} \mathbb{Z}^*_{N_j}$
2: $Q \overset{?}{\in} \mathbb{Z}^*_{N_j}$
3: $A \overset{?}{\in} \mathbb{Z}^*_{N_j}$
4: $B \overset{?}{\in} \mathbb{Z}^*_{N_j}$
5: $T \overset{?}{\in} \mathbb{Z}^*_{N_j}$

Verify statement:

6: $N_i \overset{?}{>} 2^{4\ell}$
7: $s_j^{z_1} t_j^{w_1} \overset{?}{=} A \cdot P^e \bmod N_j \overset{?}{\in} \mathbb{Z}^*_{N_j}$
8: $s_j^{z_2} t_j^{w_2} \overset{?}{=} B \cdot Q^e \bmod N_j \overset{?}{\in} \mathbb{Z}^*_{N_j}$ ▷ This and previous expressions involve fixed-base multiexp
9: $Q^{z_1} t_j^{v} \overset{?}{=} T \cdot s_j^{N_i \cdot e} \bmod N_j \overset{?}{\in} \mathbb{Z}^*_{N_j}$
10: $z_1 \overset{?}{\in} \pm 2^{\ell+\varepsilon} \cdot \sqrt{N_i}$
11: $z_2 \overset{?}{\in} \pm 2^{\ell+\varepsilon} \cdot \sqrt{N_i}$

### 3.7.2 Non-Interactive Version of the Proof

$\texttt{ProveNI}^L_{\texttt{fac}}(\mathsf{state}, R_j, N_i; (p, q)) \to ((P, Q, A, B, T); (z_1, z_2, w_1, w_2, v))$

**Inputs:**

– security level $L = (\ell, \dots)$,
– shared $\mathsf{state} \in \mathbb{B}^*$,
– auxiliary data $R_j$,
– public data $N_i \in \mathbb{Z}$,
– secret data: primes $p, q \in \mathbb{Z}^2$

**Outputs:**

– commitment $(P, Q, A, B, T) \in \mathbb{Z}^5$,
– proof $(z_1, z_2, w_1, w_2, v) \in \mathbb{Z}^5$

1: $((P, Q, A, B, T), (\alpha, \beta, \mu, \nu, r, x, y)) \leftarrow \texttt{Commit}^L_{\texttt{fac}}(R_j, N_i)$
2: $e \in \pm 2^\ell = \texttt{ChallengeNI}^L_{\texttt{fac}}(\mathsf{state}, R_j, N_i, (P, Q, A, B, T))$
3: $(z_1, z_2, w_1, w_2, v) = \texttt{Prove}_{\texttt{fac}}(e; (p, q), (\alpha, \beta, \mu, \nu, r, x, y))$
4: **return** $((P, Q, A, B, T), (z_1, z_2, w_1, w_2, v))$

$\texttt{VerifyNI}^{L}_{\texttt{fac}}(\mathsf{state}, R_j, N_i, ((P, Q, A, B, T), (z_1, z_2, w_1, w_2, v)))$

**Inputs:**

- security level $L = (\ell, \dots)$,
- shared $\mathsf{state} \in \mathbb{B}^*$,
- auxiliary data $R_j$,
- public data $N_i \in \mathbb{Z}$,
- non-interactive proof, consisting of:
  - commitment $(P, Q, A, B, T) \in \mathbb{Z}^5$,
  - proof $(z_1, z_2, w_1, w_2, v) \in \mathbb{Z}^5$

**Outputs:** aborts if proof is invalid

1: $e \in \pm 2^{\ell} = \texttt{ChallengeNI}^{L}_{\texttt{fac}}(\mathsf{state}, R_j, N_i, (P, Q, A, B, T))$
2: Assert $\texttt{Verify}^{L}_{\texttt{fac}}(R_j, N_i, (P, Q, A, B, T), e, (z_1, z_2, w_1, w_2, v))$

### 3.8 $\Pi^{\texttt{sch}}$: Schnorr Proof of Knowledge

We describe the standard Schnorr proof of knowledge, and also set up notation that we will use in what follows.

$\texttt{Commit}_{\texttt{sch}}() \to (A; \alpha)$

**Inputs:** —

**Outputs:** public commitment $A \in \mathbb{E}$ and secret nonce $\alpha \in \mathbb{Z}_q$

1: $\alpha \leftarrow \mathbb{Z}_q$
2: $A = \alpha \cdot G$
3: **return** $(A; \alpha)$

$\texttt{Challenge}_{\texttt{sch}}() \to e$

**Inputs:** —

**Outputs:** challenge $e \in \mathbb{Z}_q$

1: $e \leftarrow \mathbb{Z}_q$
2: **return** $e$

$\texttt{Prove}_{\texttt{sch}}(e; \alpha, x) \to z$

**Inputs:**

- challenge $e \in \mathbb{Z}_q$,
- secret commitment nonce $\alpha \in \mathbb{Z}_q$,
- secret data: $x \in \mathbb{Z}_q$

**Outputs:** proof $z \in \mathbb{Z}_q$

1: $z = \alpha + ex \bmod q$
2: **return** $z$

$\texttt{Verify}_{\texttt{sch}}(X, A, e, z)$

**Inputs:**

- public data $X \in \mathbb{E}$,
- public commitment $A \in \mathbb{E}$,
- challenge $e \in \mathbb{Z}_q$,
- proof $z \in \mathbb{Z}_q$

**Outputs:** aborts if proof is invalid

1: $z \cdot G \stackrel{?}{=} A + e \cdot X$

### 3.9 $\Pi^{\texttt{elog}}$: Dlog with El-Gamal Commitment

Common (public) inputs: elliptic points $L, M, X, Y, H \in \mathbb{E}^5$
Prover has secret input $(y, \lambda) \in \mathbb{Z}_q^2$ such that $L = \lambda G, M = yG + \lambda X$, and $Y = yH$

#### 3.9.1 Interactive Version of the Proof

$\texttt{Commit}_{\texttt{elog}}(X, H) \to ((A, N, B); (\alpha, m))$

**Inputs:**

- (subset of) public data $(X, H) \in \mathbb{E}^2$,

**Outputs:**

- pubilc commitment $(A, N, B) \in \mathbb{E}^3$,
- secret nonce $(\alpha, m) \in \mathbb{Z}_q^2$,

1: $\alpha, m \leftarrow \mathbb{Z}_q^2$
2: $A = \alpha \cdot G$
3: $N = m \cdot G + \alpha \cdot X$
4: $B = m \cdot H$
5: **return** $((A, N, B); (\alpha, m))$

$\texttt{Challenge}_{\texttt{elog}}() \to e$

**Outputs:** challenge $q \in \mathbb{Z}_q$

1: $e \leftarrow \mathbb{Z}_q$ ▷ Note: paper suggests sampling from $\pm q$, but in this particular proof it doesn't matter, as challenge is only used in elliptic point operations. In other proofs you may see that we sample challenge from $\pm q$, but notice that challenge is also used in other arithmetic operations, e.g. $\rho^e \bmod N$ where sign of $e$ matters.
2: **return** $e$

$\texttt{Prove}_{\texttt{elog}}(e; (y, \lambda), (\alpha, m)) \to (z, u)$

**Inputs:**

- challenge $e \in \mathbb{Z}_q$
- secret data $(y, \lambda) \in \mathbb{Z}_q^2$,
- secret commitment nonce $(\alpha, m) \in \mathbb{Z}_q^2$

**Outputs:** proof $(z, u) \in \mathbb{Z}_q^2$

1: $z = \alpha + e\lambda \bmod q$
2: $u = m + ey \bmod q$

3: **return** $(z, u)$

$\texttt{Verify}_{\texttt{elog}}((L, M, X, Y, H), (A, N, B), e, (z, u))$

**Inputs:**

- public data $(L, M, X, Y, H) \in \mathbb{E}^5$,
- public commitment $(A, N, B) \in \mathbb{E}^3$,
- challenge $e \in \mathbb{Z}_q$,
- proof $(z, u) \in \mathbb{Z}_q^2$

**Outputs:** aborts if proof is invalid

1: $z \cdot G \stackrel{?}{=} A + e \cdot L$
2: $u \cdot G + z \cdot X \stackrel{?}{=} N + e \cdot M$
3: $u \cdot H \stackrel{?}{=} B + e \cdot Y$

### 3.9.2 Non-Interactive Version of the Proof

$\texttt{ProveNI}_{\texttt{elog}}(\mathsf{state}, (L, M, X, Y, H); (y, \lambda)) \to ((A, N, B), (z, u))$

**Inputs:**

- shared $\mathsf{state} \in \mathbb{B}^*$,
- public data $(L, M, X, Y, H) \in \mathbb{E}^5$,
- secret data $(y, \lambda) \in \mathbb{Z}_q^2$

**Outputs:**

- commitment $(A, N, B) \in \mathbb{E}^3$,
- proof $(z, u) \in \mathbb{Z}_q^2$

1: $((A, N, B); (\alpha, m)) \leftarrow \texttt{Commit}_{\texttt{elog}}(X, H)$
2: $e \in \mathbb{Z}_q = \texttt{ChallengeNI}_{\texttt{elog}}(\mathsf{state}, (L, M, X, Y, H), (A, N, B))$
3: $(z, u) = \texttt{Prove}_{\texttt{elog}}(e; (y, \lambda), (\alpha, m))$
4: **return** $((A, N, B), (z, u))$

$\texttt{VerifyNI}_{\texttt{elog}}(\mathsf{state}, (L, M, X, Y, H), ((A, N, B), (z, u)))$

**Inputs:**

- shared $\mathsf{state} \in \mathbb{B}^*$,
- public data $(L, M, X, Y, H)$,
- non-interactive proof, consisting of:
  - commitment $(A, N, B) \in \mathbb{E}^3$,
  - proof $(z, u) \in \mathbb{Z}_q^2$

**Outputs:** aborts if proof is invalid

1: $e \in \mathbb{Z}_q = \texttt{ChallengeNI}_{\texttt{elog}}(\mathsf{state}, (L, M, X, Y, H), (A, N, B))$
2: Assert $\texttt{Verify}_{\texttt{elog}}((L, M, X, Y, H), (A, N, B), e, (z, u))$

### 3.10 $\Pi^{\texttt{enc-elg}}$: Range Proof with El-Gamal Commitment

Common (public) input: security level $L = (\ell, \varepsilon, \dots)$, verifier's auxilary data $R_j = (N_j, s_j, t_j) \in (\mathbb{Z}, \mathbb{Z}^*_{N_j}, \mathbb{Z}^*_{N_j})$, curve $\mathbb{E}$ with generator $G$ of prime order $q$, Paillier public key $N_0 \in \mathbb{Z}$, Paillier ciphertext $C \in \mathbb{Z}^*_{N_0^2}$, and commitments $(A, B, X) \in \mathbb{E}^3$.

Prover secret input: $(x, \rho, a, b) \in (\pm 2^\ell, \mathbb{Z}^*_{N_0}, \mathbb{Z}_q, \mathbb{Z}_q)$ such that $x \in \pm 2^\ell, C = \texttt{enc}_{N_0}(x, \rho), A = a \cdot G, B = b \cdot G, X = (ab + x) \cdot G$.

Proof guarantees that $x \in \pm 2^{\ell+\varepsilon}$

#### 3.10.1 Interactive Version of the Proof

$\texttt{Commit}^L_{\texttt{enc-elg}}(R_j, (N_0, A); x) \to ((S, T, D, Y, Z); (\alpha, \mu, r, \beta, \gamma))$

**Inputs:**

- security level $L = (\ell, \varepsilon, \dots)$,
- verifier's auxiliary data $R_j = (N_j, s_j, t_j) \in \mathbb{Z}^3$,
- (subset of) public data $(N_0, A) \in (\mathbb{Z}, \mathbb{E})$
- (subset of) secret data $x \in \mathbb{Z}$

**Outputs:**

- public commitment $(S, T, D, Y, Z) \in (\mathbb{Z}^*_{N_j}, \mathbb{Z}^*_{N_j}, \mathbb{Z}^*_{N_0^2}, \mathbb{E}, \mathbb{E})$,
- secret nonce $(\alpha, \mu, r, \beta, \gamma) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}_q, \mathbb{Z})$

1: $\alpha \leftarrow \pm 2^{\ell+\varepsilon}$
2: $\mu \leftarrow \pm 2^\ell \cdot N_j$
3: $r \leftarrow \mathbb{Z}^*_{N_0}$
4: $\beta \leftarrow \mathbb{Z}_q$
5: $\gamma \leftarrow \pm 2^{\ell+\varepsilon} \cdot N_j$

6: $S = s_j^x t_j^\mu \bmod N_j$
7: $T = s_j^\alpha t_j^\gamma \bmod N_j$ ▷ $S, T$ are computed via fixed-base multiexp algorithm
8: $D = \texttt{enc}_{N_0}(\alpha, r)$
9: $Y = \beta \cdot A + \alpha G$
10: $Z = \beta \cdot G$

11: **return** $(S, T, D, Y, Z); (\alpha, \mu, r, \beta, \gamma)$

$\texttt{Challenge}_{\texttt{enc-elg}}() \to e$

**Inputs:** none

**Outputs:** challenge $e \in \pm q$

1: $e \leftarrow \pm q$
2: **return** $e$

$\texttt{Prove}_{\texttt{enc-elg}}(N_0, e; (x, \rho, b), (\alpha, \mu, r, \beta, \gamma)) \to (z_1, z_2, z_3, w)$

**Inputs:**

- (subset of) public inputs: $N_0 \in \mathbb{Z}$,

– challenge $e \in \mathbb{Z}_q$,
– (subset of) secret data: $(x, \rho, b) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}_q)$,
– secret commitment nonce: $(\alpha, \mu, r, \beta, \gamma) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}_q, \mathbb{Z})$

**Outputs:** proof $(z_1, z_2, z_3, w) \in (\mathbb{Z}, \mathbb{Z}_{N_0}, \mathbb{Z}, \mathbb{Z}_q)$

1: $z_1 = \alpha + ex$
2: $z_2 = r \cdot \rho^e \bmod N_0$
3: $z_3 = \gamma + e\mu$
4: $w = \beta + eb \bmod q$
5: **return** $(z_1, z_2, z_3, w)$

$\texttt{Verify}^L_{\texttt{enc-elg}}(R_j, (N_0, C, A, B, X), (S, T, D, Y, Z), e, (z_1, z_2, z_3, w))$

**Inputs:**

– security level $L = (\ell, \varepsilon, \dots)$,
– prover's auxiliary data $R_j = (N_j, s_j, t_j) \in \mathbb{Z}^3$,
– public data $(N_0, C, A, B, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{E}, \mathbb{E}, \mathbb{E})$,
– commitment $(S, T, D, Y, Z) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{E}, \mathbb{E})$,
– proof $(z_1, z_2, z_3, w) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}_q)$

**Outputs:** aborts if proof is invalid

Verify that inputs are in expected domains:
1: $C \overset{?}{\in} \mathbb{Z}^*_{N_0^2}$
2: $S \overset{?}{\in} \mathbb{Z}^*_{N_j}$
3: $T \overset{?}{\in} \mathbb{Z}^*_{N_j}$
4: $D \overset{?}{\in} \mathbb{Z}^*_{N_0^2}$
Verify statement:
5: $\texttt{enc}_{N_0}(z_1, z_2) \overset{?}{=} D \oplus e \odot C$
6: $w \cdot A + z_1 \cdot G \overset{?}{=} Y + e \cdot X$
7: $w \cdot G \overset{?}{=} Z + e \cdot B$
8: $s^{z_1} t^{z_3} \overset{?}{=} T \cdot S^e \bmod N_j$ ▷ left part is computed via fixed-base multiexp
9: $z_1 \overset{?}{\in} \pm 2^{\ell+\varepsilon}$

### 3.10.2 Non-Interactive Version of the Proof

$\texttt{ProveNI}^L_{\texttt{enc-elg}}(\mathsf{state}, R_j, (N_0, C, A, B, X); (x, \rho, b)) \to ((S, T, D, Y, Z), (z_1, z_2, z_3, w))$

**Inputs:**

– security level $L$,
– shared $\mathsf{state} \in \mathbb{B}^*$,
– verifier's auxiliary data $R_j$,
– public data $(N_0, C, A, B, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{E}, \mathbb{E}, \mathbb{E})$,
– (subset of) secret data $(x, \rho, b) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}_q)$,

**Outputs:**

- commitment $(S, T, D, Y, Z) \in (\mathbb{Z}^*_{N_j}, \mathbb{Z}^*_{N_j}, \mathbb{Z}^*_{N_0^2}, \mathbb{E}, \mathbb{E})$,
- proof $(z_1, z_2, z_3, w) \in (\mathbb{Z}, \mathbb{Z}_{N_0}, \mathbb{Z}, \mathbb{Z}_q)$

1: $((S, T, D, Y, Z); (\alpha, \mu, r, \beta, \gamma)) \leftarrow \mathtt{Commit}^L_{\mathtt{enc\text{-}elg}}(R_j, (N_0, A); x)$
2: $e \in \pm q = \mathtt{ChallengeNI}^L_{\mathtt{enc\text{-}elg}}(\mathsf{state}, (N_0, C, A, B, X), (S, T, D, Y, Z))$
3: $(z_1, z_2, z_3, w) = \mathtt{Prove}_{\mathtt{enc\text{-}elg}}(N_0, e; (x, \rho, b), (\alpha, \mu, r, \beta, \gamma))$
4: **return** $((S, T, D, Y, Z), (z_1, z_2, z_3, w))$

$\mathtt{VerifyNI}^L_{\mathtt{enc\text{-}elg}}(\mathsf{state}, R_j, (N_0, C, A, B, X), ((S, T, D, Y, Z), (z_1, z_2, z_3, w)))$

**Inputs:**

- security level $L$,
- shared $\mathsf{state} \in \mathbb{B}^*$,
- verifier's auxiliary data $R_j$,
- public data $(N_0, C, A, B, X) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{E}, \mathbb{E}, \mathbb{E})$,
- non-interactive proof consisting of:
  - commitment $(S, T, D, Y, Z) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{E}, \mathbb{E})$,
  - proof $(z_1, z_2, z_3, w) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}_q)$

**Outputs:** aborts if proof is invalid

1: $e \in \pm q = \mathtt{ChallengeNI}^L_{\mathtt{enc\text{-}elg}}(R_j, (N_0, C, A, B, X), (S, T, D, Y, Z))$
2: Assert $\mathtt{Verify}_{\mathtt{enc\text{-}elg}}(R_j, (N_0, C, A, B, X), (S, T, D, Y, Z), e, (z_1, z_2, z_3, w))$

# 4 Threshold Protocols

In this section we describe the various threshold protocols we have implemented. Overall, we have the following protocols:

1. When a "signing cluster" is initialized, the signers in that cluster run a provisioning protocol in which they each generate auxiliary information. That protocol is described in Section 4.1.

2. When key generation is requested in an initialized cluster, the signers in that cluster run a distributed key-generation protocol. We have implemented protocols for both $n$-out-of-$n$ key generation (cf. Section 4.2.1), as well as $t$-out-of-$n$ key generation (cf. Section 4.2.2).

3. A threshold of signers who have already generated a key can compute *presignatures* with respect to that key. Protocols for doing that, in both the "non-threshold" (i.e., $n$-out-of-$n$) and "threshold" (i.e., $t$-out-of-$n$) cases, are described in Section 4.3.

4. If a presignature has already been generated by some threshold of signers with respect to some key, those signers can non-interactively compute a signature on a given message $m$ using the presignature they have computed. See Section 4.4.

## 4.1 Provisioning Protocol

This protocol is run to generate auxiliary information for each signer in a cluster.

### 4.1.1 Utility functions

$\texttt{generate\_paillier\_key}^L() \to (N; p, q)$

**Inputs:** security level $L = (\kappa_{\texttt{rsa}}, \dots)$

**Outputs:**

- Paillier public key $N \in \mathbb{Z}$ of $n_{\texttt{rsa}}$ bits or more
- safe primes $p, q \in \mathbb{Z}^2$, each of $\kappa_{\texttt{rsa}}$ bits or more

1: Generate $\kappa_{\texttt{rsa}}$-bit safe primes $p, q$ using the algorithm from Section 2.1.
2: Compute Pailier public key $N = pq$
3: **return** $(N; p, q)$

$\texttt{generate\_pedersen\_params}^L() \to (N, s, t; \phi, \lambda)$

**Inputs:** security level $L = (\kappa_{\texttt{rsa}}, \dots)$

**Outputs:** Pedersen public parameters $N, s, t \in \mathbb{Z}^3$, and secret parameters $\phi = \phi(N)$ and $\lambda \in \mathbb{Z}$

1: $(N; p, q) \leftarrow \texttt{generate\_paillier\_key}^L()$
2: Set $\phi = (p-1)(q-1)$
3: Sample $r \leftarrow \mathbb{Z}_N^*$, and $\lambda \leftarrow [\phi/4]$
4: Set $t = r^2 \bmod N$, and $s = t^\lambda \bmod N$ ▷ $s$ is computed via CRT exponentiation as described in 2.2
5: **return** $(N, s, t; \phi, \lambda)$

### 4.1.2 Protocol

**Input.**

- party index $i$, $0 \le i < n$,
- amount of parties in the protocol $n \ge 2$,
- context separation string $\texttt{sid} \in \mathbb{B}^*$,
- security level $L = (\kappa, \kappa_{\texttt{rsa}}, n_{\texttt{rsa}}, \dots)$ (see Section 2.5),
- flag $\texttt{echo\_enabled} \in \{0,1\}$ indicating whether reliability check is enabled

**Round 1.**

1: Generate Paillier key $(N_i; p_i, q_i) \leftarrow \texttt{generate\_paillier\_key}^L()$
2: Generate Pedersen parameters $(\hat{N}_i, s_i, t_i; \hat{\phi}_i, \lambda_i) \leftarrow \texttt{generate\_pedersen\_params}^L()$
3: Compute $\hat{\psi}_i = \texttt{ProveNI}^L_{\texttt{prm}}(\texttt{Encode}(\texttt{sid}, i), \hat{N}_i, s_i, t_i; \hat{\phi}_i, \lambda_i)$.
4: Sample $\rho_i, u_i \leftarrow \{0,1\}^\kappa$, and compute $V_i = H(\texttt{Encode}_{\texttt{hash\_com}}(\texttt{sid}, i, N_i, \hat{N}_i, s_i, t_i, \hat{\psi}_i, \rho_i, u_i))$.
5: Send $V_i$ to all parties.

**Round 2.**

1: Receive $V_j \in \mathbb{B}^*$ from all parties
2: **if** $\texttt{echo\_enabled}$ **then** ▷ Reliability check
3: Compute $h_i = H(\texttt{Encode}_{\texttt{echo}}(\texttt{sid}, V_0, \dots, V_{n-1}))$ and send $h_i$ to all parties
4: Upon receiving $h_j \in \{0,1\}^\kappa$ from all parties, abort if $h_i \neq h_j$ for some $j \in [n]$
5: Send $(N_i, \hat{N}_i, s_i, t_i, \hat{\psi}_i, \rho_i, u_i)$ to all parties

**Round 3.**

1: Receive $(N_j, \hat{N}_j, s_j, t_j, \hat{\psi}_j, \rho_j, u_j)$ from all parties
2: For all $j \in [n]$, set $R_j = (\hat{N}_j, s_j, t_j)$; let $\vec{R} = (R_j)_{j\in[n]}$, and $\vec{N} = (N_j)_{j\in[n]}$
3: **for** $j \in [n] \setminus \{i\}$ **do**
4: Assert $(N_j, \hat{N}_j, s_j, t_j, \rho_j, u_j) \stackrel{?}{\in} (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \{0,1\}^\kappa, \{0,1\}^\kappa)$
5: Assert $V_j \stackrel{?}{=} H(\texttt{Encode}_{\texttt{hash\_com}}(\texttt{sid}, j, N_j, \hat{N}_j, s_j, t_j, \hat{\psi}_j, \rho_j, u_j))$
6: Assert that $N_j, \hat{N}_j$ are at least $n_{\texttt{rsa}}$ bits in length
7: Assert $\texttt{VerifyNI}^L_{\texttt{prm}}(\texttt{Encode}(\texttt{sid}, j), \hat{N}_j, s_j, t_j, \hat{\psi}_j)$
8: Compute $\rho = \bigoplus_j \rho_j$
9: Compute $\psi_i = \texttt{ProveNI}^L_{\texttt{mod}}(\texttt{Encode}(\texttt{sid}, i, \rho), N_i; (p_i, q_i))$
10: **for** $j \in [n] \setminus \{i\}$ **do**
11: Compute $\psi'_{i,j} = \texttt{ProveNI}^L_{\texttt{fac}}(\texttt{Encode}(\texttt{sid}, i, \rho), R_j, N_i; (p_i, q_i))$
12: Send $(\psi_i, \psi'_{i,j})$ to $P_j$

**Output.**

1: Receive $(\psi_j, \psi'_{j,i})$ from all parties
2: **for** $j \in [n] \setminus \{i\}$ **do**
3: Assert $\texttt{VerifyNI}^L_{\texttt{mod}}(\texttt{Encode}(\texttt{sid}, j, \rho), N_j, \psi_j)$
4: Assert $\texttt{VerifyNI}^L_{\texttt{fac}}(\texttt{Encode}(\texttt{sid}, j, \rho), R_i, N_j, \psi'_{j,i})$
5: For $j \in [n]$ (including $j = i$), precompute a fixed-based multiexponentiation table $T_j$ as described in Section 2.4. Let $\vec{T} = (T_j)_{j\in[n]}$
6: **return** $(p_i, q_i, \vec{N}, \vec{R}, \vec{T})$

## 4.2 Distributed Key Generation

We implement two versions of distributed key generation. One generates a key along with an $n$-out-of-$n$ additive sharing of that key, and the other generates a key along with a $t$-out-of-$n$ Shamir secret sharing of that key. Note that only the former protocol is described in [2].

### 4.2.1 Non-Threshold (i.e., $n$-out-of-$n$) Key Generation

This protocol is based on [2, Figure 5], but we have added the option to replace the broadcast channel with a reliable broadcast subroutine, and we added optional support of HD-wallets.

**Input.**

- party index $i$, $0 \le i < n$,
- number of signers $n \ge 2$,
- context separation string $\texttt{sid} \in \mathbb{B}^*$,
- security level $L$ (see Section 2.5),
- flag $\texttt{echo\_enabled} \in \{0,1\}$ indicating whether reliability check is enabled,
- flag $\texttt{hd\_enabled} \in \{0,1\}$ indicating whether HD-wallets support is enabled,
- curve $\mathbb{E}$ with generator $G$ of prime order $q$

**Round 1.**

1: Sample $x_i \leftarrow \mathbb{Z}_q$, and set $X_i = x_i \cdot G$
2: Sample $\mathsf{rid}_i \leftarrow \{0,1\}^\kappa$
3: Compute $(A_i; \tau_i) = \mathtt{Commit}_{\mathtt{sch}}()$
4: **if** `hd_wallets` **then**
5: Sample local chain code contribution $c_i \leftarrow \mathbb{B}^{32}$
6: **else**
7: Set $c_i = \bot$
8: Sample $u_i \leftarrow \{0,1\}^\kappa$ and set $V_i = H(\mathtt{Encode}_{\mathtt{hash\_com}}(\mathtt{sid}, i, \mathsf{rid}_i, X_i, A_i, u_i, c_i))$
9: Send $V_i$ to all parties

**Round 2.**

1: Receive $V_j \in \{0,1\}^\kappa$ from all parties
2: **if** `echo_enabled` **then** ▷ Reliability check
3: Compute $h_i = H(\mathtt{Encode}_{\mathtt{echo}}(\mathtt{sid}, V_0, \ldots, V_{n-1}))$ and send $h_i$ to all parties
4: Upon receiving $h_j \in \{0,1\}^\kappa$ from all other parties: abort if $h_i \neq h_j$ for some $j \in [n]$
5: Send $(\mathsf{rid}_i, X_i, A_i, u_i, c_i)$ to all parties

**Round 3.**

1: Receive $(\mathsf{rid}_j, X_j, A_j, u_j, c_j)$ from all other parties
2: Abort if $(\mathsf{rid}_j, X_j, A_j, u_j) \notin (\{0,1\}^\kappa, \mathbb{E}, \mathbb{E}, \{0,1\}^\kappa)$ for some $j \in [n]$
3: Abort if $V_j \neq H(\mathtt{Encode}_{\mathtt{hash\_com}}(\mathtt{sid}, j, \mathsf{rid}_j, X_j, A_j, u_j, c_j))$ for some $j \in [n]$
4: Set $\mathsf{rid} = \bigoplus_j \mathsf{rid}_j$
5: **if** `hd_wallets` **then**
6: Check that $c_j \stackrel{?}{\in} \mathbb{B}^{32}$ for all $j \in [n]$
7: Set chain code $c = \bigoplus_j c_j$
8: Set $e_i \in \mathbb{Z}_q = \mathtt{ChallengeNI}_{\mathtt{sch}}(\mathtt{sid}, i, \mathsf{rid}, X_i, A_i)$ and compute $\psi_i = \mathtt{Prove}_{\mathtt{sch}}(e_i; \tau_i, x_i)$
9: Send $\psi_i$ to all parties

**Output.**

1: Upon receiving $\psi_j \in \mathbb{Z}_q$ from all other parties:
2: **for** $j \in [n] \setminus \{i\}$ **do**
3: Set $e_j \in \mathbb{Z}_q = \mathtt{ChallengeNI}_{\mathtt{sch}}(\mathtt{sid}, j, \mathsf{rid}, X_j, A_j)$
4: Assert $\mathtt{Verify}_{\mathtt{sch}}(X_j, A_j, e_j, \psi_j)$
5: Set $X = \sum_j X_j$, $\vec{X} = (X_j)_{j \in [n]}$
6: **return** $(X, x_i, \vec{X}, c)$

#### 4.2.2 Threshold (i.e., $t$-out-of-$n$) Key Generation

**Input.**

- party index $i$, $0 \leq i < n$,
- threshold parameter $t$, $2 \leq t \leq n$,
- number of signers $n \geq 2$,
- context separation string $\mathtt{sid} \in \mathbb{B}^*$,
- security level $L$ (see Section 2.5),
- flag `echo_enabled` $\in \{0,1\}$ indicating whether reliability check is enabled,

– flag $\texttt{hd\_enabled} \in \{0,1\}$ indicating whether HD-wallets support is enabled,
– curve $\mathbb{E}$ with generator $G$ of prime order $q$

**Round 1.**

Sample $s_{i,0}, \ldots, s_{i,t-1} \leftarrow \mathbb{Z}_q^t$.
Set $\vec{S}_i = (s_{i,k} \cdot G)_{k \in [t]}$
Let $f_i(x) = \sum_{k \in [t]} s_{i,k} \cdot x^k$ and $F_i(x) = f_i(x) \cdot G$
Compute $\sigma_{i,j} = f_i(j+1)$ for all $j \in [n]$
Sample $\mathsf{rid}_i \leftarrow \{0,1\}^\kappa$
Compute $(A_i; \tau_i) \leftarrow \texttt{Commit}_{\texttt{sch}}()$
**if** $\texttt{hd\_wallets}$ **then**
    Sample local chain code contribution $c_i \leftarrow \mathbb{B}^{32}$
**else**
    Set $c_i = \bot$
Sample $u_i \leftarrow \{0,1\}^\kappa$ and compute $V_i = H(\texttt{Encode}_{\texttt{hash\_com}}(\texttt{sid}, i, \mathsf{rid}_i, \vec{S}_i, A_i, u_i, c_i))$
Send $V_i$ to all parties

**Round 2.**

Receive $V_j \in \{0,1\}^\kappa$ from all parties
**if** $\texttt{echo\_enabled}$ **then** ▷ Reliability check
    Compute $h_i = H(\texttt{Encode}_{\texttt{echo}}(\texttt{sid}, V_0, \ldots, V_{n-1}))$, and send $h_i$ to all parties.
    Upon receiving $h_j \in \{0,1\}^\kappa$ from all parties: abort if $h_i \neq h_j$ for some $j \in [n]$.
Send $(\mathsf{rid}_i, \vec{S}_i, A_i, u_i, c_i)$ to all parties.
For all $j \neq i$, send $\sigma_{i,j}$ to $P_j$ via private channel.

**Round 3.**

Receive $(\mathsf{rid}_j, \vec{S}_j, A_j, u_j, c_j)$ and $\sigma_{j,i}$ from all parties:
**for** $j \in [n] \setminus \{i\}$ **do**
    Check that $(\mathsf{rid}_j, \vec{S}_j, A_j, u_j) \stackrel{?}{\in} (\{0,1\}^\kappa, \mathbb{E}^t, \mathbb{E}, \{0,1\}^\kappa)$ ▷ make sure that $\vec{S}_j$ has length $t$
    Assert $V_j \stackrel{?}{=} H(\texttt{Encode}_{\texttt{hash\_com}}(\texttt{sid}, j, \mathsf{rid}_j, \vec{S}_j, A_j, u_j, c_j))$
    Define $F_j(x) = \sum_{k \in [t]} x^k \cdot S_{j,k}$
    Assert $\sigma_{j,i} \cdot G \stackrel{?}{=} F_j(i+1)$
Compute $\mathsf{rid} = \bigoplus_{j \in [n]} \mathsf{rid}_j$
**if** $\texttt{hd\_wallet}$ **then**
    Check that $c_j \stackrel{?}{\in} \mathbb{B}^{32}$ for all $j \in [n]$
    Set chain code $c = \bigoplus_{j \in [n]} c_j$
Let $F(x) = \sum_{k \in [t]} x^k \cdot \left( \sum_{j \in [n]} S_{j,k} \right) = \sum_{j \in [n]} F_j(x)$
For $j \in [n]$, compute $X_j = F(j+1)$. Let $\vec{X} = (X_j)_{j \in [n]}$
Compute $x_i = \sum_{j \in [n]} \sigma_{j,i}$
Compute challenge $e_i \in \mathbb{Z}_q = \texttt{ChallengeNI}_{\texttt{sch}}(\texttt{sid}, i, \mathsf{rid}, X_i, A_i)$
Compute Schnorr proof $\psi_i = \texttt{Prove}_{\texttt{sch}}(e_i; \tau_i, \sigma_i)$
Send $\psi_i$ to all parties

**Output.**

1: Receive $\psi_j \in \mathbb{F}_q$ from all parties
2: **for** $j \in [n] \setminus \{i\}$ **do**
3: Set $e_j \in \mathbb{Z}_q = \texttt{ChallengeNI}_{\texttt{sch}}(\texttt{sid}, j, \mathsf{rid}, X_j, A_j)$
4: Assert $\texttt{Verify}_{\texttt{sch}}(X_j, A_j, e_j, \psi_j)$
5: Compute $Y = \sum_{j\in[n]} S_{j,0}$
6: Create identity mapping $I : [n] \to \mathbb{Z}_q \setminus \{0\}$, $I(i) = i + 1$
7: **return** $(Y, x_i, \vec{X}, I, c)$

## 4.3 Presigning

We implemented both non-threshold and threshold versions of presigning. The non-threshold version assumes the $n$ parties running the protocol have additive shares of the private key. The threshold version of the protocol, which is not described in [2], first maps the key shares (which are a $t$-out-of-$n$ Shamir sharing of the private key) to a $t$-out-of-$t$ sharing of the key using Lagrange interpolation and then runs the non-threshold protocol among the $t$ parties.

### 4.3.1 Non-Threshold ($n$-out-of-$n$) Presigning

The following protocol is based on [2, Figure 7], although we have corrected some typos and eliminated some extraneous parts. (In particular, we do not have identifiable abort.)

**Input.**

- party index $i$, $0 \le i < n$,
- number of signers $n \ge 2$,
- (additive) secret share $x_i \in \mathbb{Z}_q$,
- list of signers' public key shares $\vec{X} = \{X_j\}_{j\in[n]} \in \mathbb{E}^n$,
- Paillier private key $\mathsf{sk}_i$,
- list of signers' Paillier public keys $\vec{N} = (N_j)_{j\in[n]} \in \mathbb{Z}^n$
- list of signers' pedersen parameters $\vec{R} = \{(s_j, t_j, \hat{N}_j)\}_{j\in[n]} \in \{\mathbb{Z}^3\}^n$,
- context separation string $\texttt{sid} \in \mathbb{B}^*$,
- security level $L$ (see Section 2.5),
- flag $\texttt{echo\_enabled} \in \{0,1\}$ indicating whether reliability check is enabled,
- curve $\mathbb{E}$ with generator $G$ of prime order $q$

**Outputs.**

- secret presignature $(\Gamma, \tilde{k}_i, \tilde{\chi}_i) \in (\mathbb{E}, \mathbb{Z}_q, \mathbb{Z}_q)$,
- public presignature commitments $(\Gamma, (\tilde{\Delta}_j, \tilde{S}_j)_{j\in[n]}) \in (\mathbb{E}, (\mathbb{E}, \mathbb{E})^n)$

**Round 1.**

1: Sample $k_i, \gamma_i \leftarrow \mathbb{Z}_q^2$, $\rho_i, v_i \leftarrow (\mathbb{Z}_{N_i}^*)^2$
2: Set $G_i = \mathsf{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(\gamma_i; v_i)$, $K_i = \mathsf{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(k_i; \rho_i)$.
3: Sample $Y_i \leftarrow \mathbb{E}$, and $a_i, b_i \leftarrow \mathbb{Z}_q^2$
4: Set $A_{i,1} = a_i G$, $A_{i,2} = a_i Y_i + k_i G$
5: Set $B_{i,1} = b_i G$, $B_{i,2} = b_i Y_i + \gamma_i G$

6: **for** $j \in [n] \setminus \{i\}$ **do**
7: $\quad \psi^0_{j,i} = \mathtt{ProveNI}^L_{\mathtt{enc\text{-}elg}}(\mathtt{Encode}(\mathtt{sid}, i), R_j, (N_i, K_i, Y_i, A_{i,1}, A_{i,2}); (k_i, \rho_i, a_i))$
8: $\quad \psi^1_{j,i} = \mathtt{ProveNI}^L_{\mathtt{enc\text{-}elg}}(\mathtt{Encode}(\mathtt{sid}, i), R_j, (N_i, G_i, Y_i, B_{i,1}, B_{i,2}); (\gamma_i, v_i, b_i))$
9: Send $(K_i, G_i, Y_i, A_{i,1}, A_{i,2}, B_{i,1}, B_{i,2})$ to all parties, and for $j \neq i$ send $(\psi^0_{j,i}, \psi^1_{j,i})$ to $P_j$

**Round 2.**

1: Receive $(K_j, G_j, Y_j, A_{j,1}, A_{j,2}, B_{j,1}, B_{j,2}, \psi^0_{i,j}, \psi^1_{i,j}) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{E}, \mathbb{E}, \mathbb{E}, \mathbb{E}, \mathbb{E}, *, *)$ from all parties
▷ domain of $\psi^0_{i,j}$ and $\psi^1_{i,j}$ is specified within ZK proof verify function
2: **if** `echo_enabled` **then** ▷ Reliability check
3: Compute

$$h_i = H(\mathtt{Encode}_{\mathtt{echo}}(\mathtt{sid}, K_0, G_0, Y_0, A_{0,1}, A_{0,2}, B_{0,1}, B_{0,2}, \ldots, K_{n-1}, G_{n-1}, Y_{n-1}, A_{n-1,1}, A_{n-1,2}, B_{n-1,1}, B_{n-1,2}))$$

and send $h_i$ to all parties
4: Upon receiving $h_j$ from all parties, abort if $h_i \neq h_j$ for some $j \in [n]$
5: **for** $j \in [n] \setminus \{i\}$ **do**
6: $\quad \mathtt{VerifyNI}^L_{\mathtt{enc\text{-}elg}}(\mathtt{Encode}(\mathtt{sid}, j), R_i, (N_j, K_j, Y_j, A_{j,1}, A_{j,2}), \psi^0_{i,j})$
7: $\quad \mathtt{VerifyNI}^L_{\mathtt{enc\text{-}elg}}(\mathtt{Encode}(\mathtt{sid}, j), R_i, (N_j, G_j, Y_j, B_{j,1}, B_{j,2}), \psi^1_{i,j})$
8: Compute $\Gamma_i = \gamma_i \cdot G$
9: Set $\tilde{\psi}_i = \mathtt{ProveNI}_{\mathtt{elog}}(\mathtt{Encode}(\mathtt{sid}, i), (B_{i,1}, B_{i,2}, Y_i, \Gamma_i, G); (\gamma_i, b_i))$
10: **for** $j \in [n] \setminus \{i\}$ **do**
11: $\quad$ Sample $r_{i,j}, \hat{r}_{i,j} \leftarrow \mathbb{Z}^2_{N_i}$, $s_{i,j}, \hat{s}_{i,j} \leftarrow \mathbb{Z}^2_{N_j}$, and $\beta_{i,j}, \hat{\beta}_{i,j} \leftarrow (\pm 2^{\ell'})^2$
12: $\quad$ Compute $D_{j,i} = (\gamma_i \odot K_j) \oplus \mathtt{enc}_{N_j}(-\beta_{i,j}; s_{i,j})$ and $F_{j,i} = \mathtt{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(-\beta_{i,j}; r_{i,j})$
13: $\quad$ Compute $\hat{D}_{j,i} = (x_i \odot K_j) \oplus \mathtt{enc}_{N_j}(-\hat{\beta}_{i,j}; \hat{s}_{i,j})$ and $\hat{F}_{j,i} = \mathtt{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(-\hat{\beta}_{i,j}; \hat{r}_{i,j})$
14: $\quad$ Compute $\psi_{j,i} = \mathtt{ProveNI}^L_{\mathtt{aff\text{-}g}}(\mathtt{Encode}(\mathtt{sid}, i), R_j, (N_j, N_i, K_j, D_{j,i}, F_{j,i}, \Gamma_i); (\gamma_i, -\beta_{i,j}, s_{i,j}, r_{i,j}))$
15: $\quad$ Compute $\hat{\psi}_{j,i} = \mathtt{ProveNI}^L_{\mathtt{aff\text{-}g}}(\mathtt{Encode}(\mathtt{sid}, i), R_j, (N_j, N_i, K_j, \hat{D}_{j,i}, \hat{F}_{j,i}, X_i); (x_i, -\hat{\beta}_{i,j}, \hat{s}_{i,j}, \hat{r}_{i,j}))$
16: $\quad$ Send $(\Gamma_i, \tilde{\psi}_i, D_{j,i}, F_{j,i}, \hat{D}_{j,i}, \hat{F}_{j,i}, \psi_{j,i}, \hat{\psi}_{j,i})$ to $P_j$

**Round 3.**

1: Receive $(\Gamma_j, \tilde{\psi}_j, D_{i,j}, F_{i,j}, \hat{D}_{i,j}, \hat{F}_{i,j}, \psi_{i,j}, \hat{\psi}_{i,j}) \in (\mathbb{E}, *, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, *, *)$ from all parties
▷ domains of the proofs are specified within ZK proofs verify functions
2: **for** $j \in [n] \setminus \{i\}$ **do**
3: $\quad$ Assert $\mathtt{VerifyNI}_{\mathtt{elog}}(\mathtt{Encode}(\mathtt{sid}, j), (B_{j,1}, B_{j,2}, Y_j, \Gamma_j, G), \tilde{\psi}_j)$
4: $\quad$ Assert $\mathtt{VerifyNI}^L_{\mathtt{aff\text{-}g}}(\mathtt{Encode}(\mathtt{sid}, j), R_i, (N_i, N_j, K_i, D_{i,j}, F_{i,j}, \Gamma_j), \psi_{i,j}; \mathsf{sk}_i)$
5: $\quad$ Assert $\mathtt{VerifyNI}^L_{\mathtt{aff\text{-}g}}(\mathtt{Encode}(\mathtt{sid}, j), R_i, (N_i, N_j, K_i, \hat{D}_{i,j}, \hat{F}_{i,j}, X_j), \hat{\psi}_{i,j}; \mathsf{sk}_i)$
6: Compute $\Gamma = \sum_{j \in [n]} \Gamma_j$ and $\Delta_i = k_i \cdot \Gamma$

7: **for** $j \in [n] \setminus \{i\}$ **do**
8: Compute $\alpha_{i,j} = \mathtt{dec}_{\mathtt{sk}_i}(D_{i,j})$ and $\hat{\alpha}_{i,j} = \mathtt{dec}_{\mathtt{sk}_i}(\hat{D}_{i,j})$
9: Compute $\psi'_i = \mathtt{ProveNI}_{\mathtt{elog}}(\mathtt{Encode}(\mathtt{sid}, i), (A_{i,1}, A_{i,2}, Y_i, \Delta_i, \Gamma); (k_i, a_i))$
10: Compute $\delta_i = \gamma_i k_i + \sum_{j \neq i}(\alpha_{i,j} + \beta_{i,j}) \bmod q$
11: Compute $\chi_i = x_i k_i + \sum_{j \neq i}(\hat{\alpha}_{i,j} + \hat{\beta}_{i,j}) \bmod q$
12: Compute $S_i = \chi_i \cdot \Gamma$
13: Send $(\delta_i, S_i, \Delta_i, \psi'_i)$ to all

**Output**

1: Receive $(\delta_j, S_j, \Delta_j, \psi'_j) \in (\mathbb{Z}_q, \mathbb{E}, \mathbb{E}, *)$ from all parties ▷ domain of $\psi'_j$ is specified within ZK proof verify function
2: Assert $\mathtt{VerifyNI}_{\mathtt{elog}}(\mathtt{Encode}(\mathtt{sid}, j), (A_{j,1}, A_{j,2}, Y_j, \Delta_j, \Gamma), \psi'_j)$ for all $j \in [n] \setminus \{i\}$
3: Compute $\delta = \sum_j \delta_j$
4: Assert $\delta \cdot G \stackrel{?}{=} \sum_j \Delta_j$
5: Assert $\delta \cdot X \stackrel{?}{=} \sum_j S_j$ ▷ in case of failure, we are supposed to run sub-protocol to identify aborting parties, but it's not yet implemented
6: Set $\tilde{k}_i = k_i/\delta$, and $\tilde{\chi}_i = \chi_i/\delta$
7: Set $\tilde{\Delta}_j = \delta^{-1}\Delta_j$ and $\tilde{S}_j = \delta^{-1}S_j$ for all $j \in [n]$
8: **return** $((\Gamma, \tilde{k}_i, \tilde{\chi}_i), (\Gamma, (\tilde{\Delta}_j, \tilde{S}_j)_{j \in [n]}))$

#### 4.3.2 Threshold ($t$-out-of-$n$) Presinging

**Input.**

- size of signing set $t \geq 2$,
- party index $i \in [t]$,
- injective index map $S : [t] \to [n]$ that maps signers indicies at signing to their indicies at key generation, i.e. $S(i)$ is the index that $P_i$ had at key-generation time,
- key share $K_{S(i)}$,
- context separation string $\mathtt{sid} \in \mathbb{B}^*$,
- additive shift $\mathtt{shift} \in \mathbb{Z}_q$ for HD child derivation. $\mathtt{shift} = 0$ disables HD derivation. The procedure for deriving the shift is out of scope for this algorithm; for specific standards for shift derivation see, for instance, [6] or [3]
- security level $L$ (see Section 2.5),
- flag $\mathtt{echo\_enabled} \in \{0,1\}$ indicating whether reliability check is enabled,
- curve $\mathbb{E}$ with generator $G$ of prime order $q$

**Map threshold signing protocol to non-threshold by transforming the shares**

1: Extract data from key share $K_{S(i)}$, which contains:
   - threshold value $\mathtt{min\_signers}$,
   - number of key holders $n$,
   - secret share $x'_{S(i)} \in \mathbb{Z}_q$,

- parties' public shares $\vec{X}' = (X'_j)_{j\in[n]} \in \mathbb{E}^n$,
- an injective map $I : [n] \to \mathbb{F}_q \setminus \{0\}$,
- Paillier secret key $\mathsf{sk}'_{S(i)}$,
- parties' Paillier keys $\vec{N}' = (N'_j)_{j\in[n]} \in \mathbb{Z}^n$,
- parties' auxiliary information $\vec{R}' = (s_j, t_j, \hat{N})_{j\in[n]} \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})^n$

2: Set $\mathsf{sk}_i = \mathsf{sk}'_{S(i)}$ and $\vec{R} = (R'_{S(j)})_{j\in[t]}$

3: **if** shares are additive[2] shares of the private key **then**

4: Set $x_i = x'_{S(i)}$, $\vec{X} = (X'_{S(j)})_{j\in[t]}$

5: **if** shares are Shamir secret shares of the private key **then**

6: For $j \in [t]$, compute Lagrange coefficient $\lambda_j = \prod_{m\in[t]\setminus\{j\}} \frac{I(S(m))}{I(S(m))-I(S(j))} \bmod q$.

7: Compute $x_i = \lambda_i \cdot x'_{S(i)}$.

8: For $j \in [t]$, compute $X_j = \lambda_j \cdot X'_{S(j)}$; then set $\vec{X} = \{X_j\}_{j\in[t]}$.

9: **if** `shift` $\neq 0$ **then** ▷ i.e. if HD wallets are enabled

10: Set $X_0 := X_0 + \text{shift} \cdot G$

11: If $i = 0$, set $x_0 := x_0 + \text{shift}$ ▷ output signature will be valid for public key $Y + \text{shift} \cdot G$

12: Call non-threshold signing protocol with inputs:

- signer index $i$,
- number of signers $n = t$,
- (additive) secret share $x_i$,
- list of signers' public key shares $\vec{X}$,
- Paillier private key $\mathsf{sk}_i$,
- list of signers' auxiliary data $\vec{R}$,
- context separation string `sid`,
- security level $L$,
- flag `echo_enabled`,
- curve $\mathbb{E}$ with generator $G$ of prime order $q$

## 4.4 Signing

The signing protocol has two parts: one that takes the output from the presignature protocol and a hashed message and produces a partial signature, and another that takes partial signatures and combines them to produce a signature.

`issue_partial_signature`$((\Gamma, \tilde{k}_i, \tilde{\chi}_i), m, \texttt{shift}) \to (r, \sigma_i)$

**Inputs:**

- presignature $(\Gamma, \tilde{k}_i, \tilde{\chi}_i) \in (\mathbb{E}, \mathbb{Z}_q, \mathbb{Z}_q)$,
- a hashed message $m \in \mathbb{Z}_q$,
- if HD wallets support is enabled, additive `shift` $\in \mathbb{Z}_q$ (`shift` $= 0$ disables HD derivation)

**Outputs:** partial signature $(r, \sigma_i) \in (\mathbb{Z}_q, \mathbb{Z}_q)$

[2] In this case we have $t = n$.

1: **if** HD-wallets support enabled **then**
2: Set $\tilde{\chi}_i := \tilde{\chi}_i + \tilde{k}_i \cdot \texttt{shift} \bmod q$
3: Set $r = \Gamma|_x$ and $\sigma_i = \tilde{k}_i m + r\tilde{\chi}_i \bmod q$
4: **return** $(r, \sigma_i)$

$\texttt{combine\_partial\_signatures}((\Gamma, (\tilde{\Delta}_j, \tilde{S}_j)_{j=0}^{n-1}), (\sigma_j)_{j=0}^{n-1}, m) \to (r, \sigma)$

**Inputs:**

- presignature commitments $(\Gamma, (\tilde{\Delta}_j, \tilde{S}_j)_{j=0}^{n-1}) \in (\mathbb{E}, (\mathbb{E}, \mathbb{E})^n)$
- partial signatures $(\sigma_j)_{j=0}^{n-1} \in \mathbb{Z}_q^n$,
- a hashed message $m \in \mathbb{Z}_q$

**Outputs:** signature $(r, \sigma) \in (\mathbb{Z}_q, \mathbb{Z}_q)$

1: Set $r := \Gamma|_x$
2: Verify that $\sigma_j \Gamma \stackrel{?}{=} m\tilde{\Delta}_j + r\tilde{S}_j$ for all $j \in [n]$
3: Let $\sigma = \sum_j \sigma_j \bmod q$
4: **return** $(r, \sigma)$

# References


[1] Elaine Barker. Recommendation for key management: Part 1 - general, nist special publication 800-57 part 1 revision 5. Available at `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf`.

[2] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. Cryptology ePrint Archive, Paper 2021/060, 2021. Available at `https://eprint.iacr.org/2021/060`.

[3] Jochen Hoenicke and Pavol Rusnak. Universal private key derivation from master private key. Available at `https://github.com/satoshilabs/slips/blob/master/slip-0010.md`.

[4] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of applied cryptography. Available at `https://cacr.uwaterloo.ca/hac/`.

[5] M.J. Wiener. Safe prime generation with a combined sieve. Available at `https://eprint.iacr.org/2003/186.pdf`.

[6] Pieter Wuille. Hierarchical deterministic wallets. Available at `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`.