CGGMP Specification

Dfns

1 Introduction

We provide a specification for our implementation of the CGGMP threshold signing protocol [1].

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, ℓ, we set $[a; b) = \{a, \ldots, b-1\}$ and $\pm\ell = \{-\ell, \ldots, 0, \ldots, \ell\}$. 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 [3]).

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' mod q ∈ {0, (q − 1)/2} for some q ∈ B continue
    If (!IsPrime(p')) continue
    If (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 [1], 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

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

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
$\phi_1 = p \cdot (p-1),\ \phi_2 := q \cdot (q-1)$
$x_1 := x \bmod \phi_1,\ x_2 := x \bmod \phi_2$
$s_1 := s \bmod p^2,\ s_2 := s \bmod q^2$
$r_1 := s_1^{x_1} \bmod p^2,\ r_2 := s_2^{x_2} \bmod q^2$
$\mathsf{res} := r_1 + ((r_2 - r_1) \cdot \beta \bmod q^2) \cdot p^2$
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 $M \in \{-(N-1)/2, \ldots, (N-1)/2\}$ 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 $M \in \{-(N-1)/2, \ldots, (N-1)/2\}$.

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 ℓ_x, ℓ_y are powers of 2. In describing the algorithm, we assume x, y are nonnegative; we can handle negative exponents by also pre-computing $s^{-\ell_x}$ 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 $i \in \{0, \ldots, k'_x - 1\}$
in a preprocessing step, compute $t_i := t^{B^i}$ for $i \in \{0, \ldots, k'_y - 1\}$
let $x_{k'_x - 1} \cdots x_0$ and $y_{k'_y - 1} \cdots y_0$ be the base-B representations of x and y, respectively
$\mathsf{res} := 1$, $\mathsf{tmp} := 1$
for $b = B - 1, \ldots, 1$ **do**
    for all i such that $y_i = b$ **do**
        $\mathsf{tmp} := \mathsf{tmp} \cdot t_i$
    for all i such that $x_i = b$ **do**
        $\mathsf{tmp} := \mathsf{tmp} \cdot s_i$
    $\mathsf{res} := \mathsf{res} \cdot \mathsf{tmp}$
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 λ denote the bit length of the curve order (so $2^\lambda \leq q < 2^{\lambda+1}$) and assume $\lambda \geq 256$ (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:

- κ determines the length of the primes used for Paillier private keys. Specifically, the primes are chosen to be of length 4κ and so the Paillier modulus has length 8κ.
- ℓ, ℓ' correspond to bounds on the ranges of certain plaintexts that are encrypted, while ε is a slack parameter. (Honest parties choose plaintexts in a range determined by ℓ or ℓ'; 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}.
- Q determines the challenge space for some of the zero-knowledge proofs.

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

2.5.1 Security Guidelines

Let $s \leq 256$ 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:

- κ 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 $\lambda = 256$.
- ℓ 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 ϵ, ℓ', which does impact efficiency.

- ϵ 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 $\epsilon \geq 2 + s + \log Q$.

- ℓ' 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 $8 \cdot (n-t+1) \cdot (t-1) \cdot 2^{-s}$.

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 Π^{enc}: Paillier Encryption in Range

We assume the prover and verifier agree on shared state 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, ρ) such that $k \in \pm 2^\ell$ and $K = \mathsf{enc}_{N_0}(k; \rho)$. In all the cases where this proof is used in the protocol, the prover knows the factorization of N_i (and thus knows sk_i) and the verifier knows the factorization of N_j (and thus knows sk_j).

3.1.1 Interactive Version of the Proof

1. In the first round of the protocol, the prover does the following:

   - The prover samples the following values:

   $$\alpha \leftarrow \pm 2^{\ell+\varepsilon}, \quad \mu \leftarrow \pm(2^\ell \cdot N_j), \quad r \leftarrow \mathbb{Z}^*_{N_i}, \quad \gamma \leftarrow \pm(2^{\ell+\varepsilon} \cdot N_j).$$

   - The prover then computes:
     - $S = s_j^k t_j^\mu \bmod N_j$
     - $A = \mathsf{enc}_{N_i}(\alpha; r)$ (this is computed as $\mathsf{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(\alpha; r)$ if sk_i is known)
     - $C = s_j^\alpha t_j^\gamma \bmod N_j$.

     Note that S and C are computed using fixed-based multiexponentiations.
   - The prover sends first message (S, A, C), and maintains local (secret) state (α, μ, r, γ).

2. The verifier chooses $e \leftarrow \pm Q$ and sends e to the prover.

3. On input (N_i, K), challenge e, and local state including (k, ρ), (α, μ, r, γ), the prover computes:

   – $z_1 = \alpha + ek$

   – $z_2 = r \cdot \rho^e \bmod N_i$

   – $z_3 = \gamma + e\mu$.

   It then sends (z_1, z_2, z_3) to the verifier.

4. Given (N_i, K), initial message (S, A, C), challenge e, and response (z_1, z_2, z_3), the verifier accepts if and only if all the following are true:

   – $A \oplus (e \odot K) = \mathtt{enc}_{N_i}(z_1; z_2) \bmod N_i^2$

   – $s_j^{z_1} t_j^{z_3} = C \cdot S^e \bmod N_j$

   – $z_1 \in \pm 2^{\ell+\varepsilon}$.

   Note the second computation involves a fixed-based multiexponentiation.

3.1.2 Non-Interactive Version of the Proof

– We deterministically derive a challenge by applying a hash function to inputs that include state, auxiliary data R_j, the common input (N_i, K), and the initial protocol message (S, A, C). We write the resulting function as $e = \mathtt{ChallengeNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), (S, A, C))$.

– The prover generates a proof as follows: first it computes (S, A, C) as described above; then it computes $e = \mathtt{ChallengeNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), (S, A, C))$; next, it computes (z_1, z_2, z_3) as described above, using the challenge e. Finally, it outputs the proof $((S, A, C), (z_1, z_2, z_3))$. We write the resulting function as $\mathtt{ProveNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), (k, \rho))$.

– A party verifies a proof $\psi = ((S, A, C), (z_1, z_2, z_3))$ by first computing

$$e = \mathtt{ChallengeNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), (S, A, C))$$

  and then verifying as described above, using the challenge e. We write the resulting function as $\mathtt{VerifyNI}^L_{\mathsf{enc}}(\mathsf{state}, R_j, (N_i, K), \psi)$.

3.2 $\Pi^{\mathtt{aff\text{-}g}}$: Paillier Affine Operation with Group Commitment in Range

We assume the prover and verifier agree on shared state state, auxiliary data[2] $R_j = (N_j, s_j, t_j)$ 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, ρ, ρ_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 sk_i) and the verifier knows the factorization of N_j (and hence knows sk_j).

[2] In [1], the auxiliary data is an arbitrary modulus $\hat{N}$. In the protocol, however, it always holds that $\hat{N} = N_j$.

3.2.1 Interactive Version of the Proof

1. In the first round of the protocol, the prover does the following:

   – The prover samples the following values:
   $\alpha \leftarrow \pm 2^{\ell+\varepsilon}, \quad r \leftarrow \mathbb{Z}_{N_j}^*, \quad \gamma, \delta \leftarrow \pm(2^{\ell+\varepsilon} \cdot N_j)$
   $\beta \leftarrow \pm 2^{\ell'+\varepsilon}, \quad r_y \leftarrow \mathbb{Z}_{N_i}^*, \quad m, \mu \leftarrow \pm(2^{\ell} \cdot N_j)$.

   – The prover then computes:
     – $A = (\alpha \odot C) \oplus \mathtt{enc}_{N_j}(\beta; r)$
     – $B_x = \alpha \cdot G$
     – $B_y = \mathtt{enc}_{N_i}(\beta; r_y)$ (this is computed as $\mathtt{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(\beta; r_y)$ if sk_i is known)
     – $E = s_j^{\alpha} t_j^{\gamma} \bmod N_j,\ S = s_j^{x} t_j^{m} \bmod N_j$
     – $F = s_j^{\beta} t_j^{\delta} \bmod N_j,\ T = s_j^{y} t_j^{\mu} \bmod N_j$.

     Note that the final two sets of computations are fixed-based multiexponentiations.

   – The prover sends first message $(A, B_x, B_y, E, S, F, T)$, and maintains local (secret) state $(\alpha, \beta, r, r_y, \gamma, \delta, m, \mu)$.

2. The verifier chooses $e \leftarrow \pm Q$ and sends e to the prover.

3. On input (N_j, N_i, C, D, Y, X), the challenge e, and local state that includes (x, y, ρ, ρ_y), $(\alpha, \beta, r, r_y, \gamma, \delta, m, \mu)$, the prover computes:

   $z_1 = \alpha + ex$

   $z_2 = \beta + ey$

   $z_3 = \gamma + em$

   $z_4 = \delta + e\mu$

   $w = r \cdot \rho^e \bmod N_j$

   $w_y = r_y \cdot \rho_y^e \bmod N_i$,

   and sends $(z_1, z_2, z_3, z_4, w, w_y)$ to the verifier.

4. Given (N_j, N_i, C, D, Y, X), initial message $(A, B_x, B_y, E, S, F, T)$, challenge e, and response $(z_1, z_2, z_3, z_4, w, w_y)$, the verifier accepts if and only if all the following are true:

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

   $z_1 \cdot G = B_x + e \cdot X$

   $B_y \oplus (e \odot Y) = \mathtt{enc}_{N_i}(z_2; w_y) \bmod N_i^2$

   $s_j^{z_1} t_j^{z_3} = E \cdot S^e \bmod N_j$

   $s_j^{z_2} t_j^{z_4} = F \cdot T^e \bmod N_j$

   $z_1 \in \pm 2^{\ell+\varepsilon}$

   $z_2 \in \pm 2^{\ell'+\varepsilon}$.

   Note that two of the above computations involve fixed-base multiexponentiations.

3.2.2 Non-Interactive Version of the Proof

– We deterministically derive a challenge by applying a hash function to inputs that include state, the auxiliary data R_j, the common input (N_j, N_i, C, D, Y, X), and the initial protocol message $(A, B_x, B_y, E, S, F, T)$. We write the resulting function as

$$e = \mathtt{ChallengeNI}^{L}_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T)).$$

– The prover generates a proof as follows: it computes its initial message $(A, B_x, B_y, E, S, F, T)$ as described above; then it computes

$$e = \mathtt{ChallengeNI}^{L}_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T));$$

next, it computes $(z_1, z_2, z_3, z_4, w, w_y)$ as described above, using the challenge e. Finally, it outputs the proof $((A, B_x, B_y, E, S, F, T), (z_1, z_2, z_3, z_4, w, w_y))$. We write the resulting function as $\mathtt{ProveNI}^{\mathbb{E},L}_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, C); (x, y, \rho, \rho_y))$.

– A party verifies a proof $\psi = ((A, B_x, B_y, E, S, F, T), (z_1, z_2, z_3, z_4, w, w_y))$ by first computing

$$e = \mathtt{ChallengeNI}^{L}_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), (A, B_x, B_y, E, S, F, T))$$

and then verifying as described above, using the challenge e. We write the resulting function as $\mathtt{VerifyNI}^{\mathbb{E},L}_{\mathtt{aff\text{-}g}}(\mathsf{state}, R_j, (N_j, N_i, C, D, Y, X), \psi)$.

3.3 Π^{mod}: Paillier-Blum Modulus

The prover and verifier agree on shared state 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 $N = pq$.

3.3.1 Interactive Version of the Proof

1. In the first round of the protocol, the prover samples uniform $w \in \mathbb{Z}_N$ with Jacobi symbol $\left(\frac{w}{N}\right) = -1$. It sends w to the verifier and maintains local state w.

2. The verifier chooses uniform $y_i \in \mathbb{Z}_N$ for $i = 1, \ldots, m$.

3. Given N, the challenge $y_1, \ldots, y_m$, and local state that includes p, q, w, the prover does the following for $i = 1, \ldots, m$:

   (a) Compute $a_i, b_i \in \{0, 1\}$ such that $y'_i = (-1)^{a_i} w^{b_i} y_i \bmod N$ is a quadratic residue modulo N.

   (b) Let x_i be the principal[3] 4th root of y'_i modulo N.

   (c) Compute $N' = N^{-1} \bmod \phi(N)$ and set $z_i = y_i^{N'} \bmod N$.

   Send $\{(x_i, a_i, b_i, z_i)\}_{i=1}^{m}$ to the verifier.

[3]This means that x_i is itself a quadratic residue.

4. Given N, initial message w, challenge $\{y_i\}_{i=1}^m$, and response $\{(x_i, a_i, b_i, z_i)\}_{i=1}^m$, the verifier accepts if and only if all the following are true:
   (a) N is an odd non-prime.
   (b) For $i \in \{1, \ldots, m\}$:
      – $z_i^N = y_i \bmod N$.
      – $x_i^4 = (-1)^{a_i} w^{b_i} y_i \bmod N$.

3.3.2 Non-Interactive Version of the Proof

– We deterministically derive a challenge by applying a hash function to inputs that include state, the common input N, and the initial protocol message w. We write the resulting function as $(y_1, \ldots, y_m) = \mathtt{ChallengeNI}^L_{\mathtt{mod}}(\mathsf{state}, N, w)$.

– The prover generates a proof by first computing its initial message w as described above; then it computes $(y_1, \ldots, y_m) = \mathtt{ChallengeNI}^L_{\mathtt{mod}}(\mathsf{state}, N, w)$; next, it computes $\{(x_i, a_i, b_i, z_i)\}_{i=1}^m$ as described above, using the challenge $\{y_i\}_{i=1}^m$. It outputs the proof $(w, \{(x_i, a_i, b_i, z_i)\}_{i=1}^m)$. We write the resulting function as $\mathtt{ProveNI}^L_{\mathtt{mod}}((\mathsf{state}, N), (p, q))$.

– A party verifies a proof $(w, \{(x_i, a_i, b_i, z_i)\}_{i=1}^m)$ by first computing

$$(y_1, \ldots, y_m) = \mathtt{ChallengeNI}^L_{\mathtt{mod}}(\mathsf{state}, N, w)$$

and then verifying as described above, using the challenge $\{y_i\}_{i=1}^m$.

3.4 $\Pi^{\mathtt{prm}}$: Ring-Pedersen Parameters

The prover and verifier agree on shared state 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 λ such that $s = t^\lambda \bmod N$, along with the factorization of N.

3.4.1 Interactive Version of the Proof

1. In the first round of the protocol, the prover first does the following for $i = 1, \ldots, m$:
   – The prover samples $a_i \leftarrow \mathbb{Z}_{\phi(N)}$.
   – The prover computes $A_i = t^{a_i} \bmod N$.

   The prover then sends first message $\{A_i\}_{i=1}^m$ and maintains local (secret) state $\{a_i\}_{i=1}^m$.

2. For $i = 1, \ldots, m$, the verifier chooses $e_i \leftarrow \{0, 1\}$, and sends $\{e_i\}_{i=1}^m$ to the prover.

3. On input N, s, t, the challenge $\{e_i\}_{i=1}^m$, and local state including $\phi(N), \lambda$, and the $\{a_i\}_{i=1}^m$, for $i = 1, \ldots, m$ the prover computes $z_i = a_i + e_i \cdot \lambda \bmod \phi(N)$. It sends $\{z_i\}_{i=1}^m$ to the verifier.

4. Given N, s, t, initial message $\{A_i\}_{i=1}^m$, challenge $\{e_i\}_{i=1}^m$, and response $\{z_i\}_{i=1}^m$, the verifier accepts if and only if $t^{z_i} = A_i s^{e_i} \bmod N$ for $i = 1, \ldots, m$.

3.4.2 Non-Interactive Version of the Proof

– We deterministically derive a challenge by hashing inputs that include state, the common input (N, s, t), and the initial protocol message $\{A_i\}_{i=1}^m$. We write the resulting function as $\{e_i\}_{i=1}^m = \mathtt{ChallengeNI}_{\mathtt{prm}}^L(\mathsf{state}, N, s, t, \{A_i\}_{i=1}^m)$.

– The prover generates a proof as follows: first, it computes its initial message $\{A_i\}_{i=1}^m$ as described above; then it computes $\{e_i\}_{i=1}^m = \mathtt{ChallengeNI}_{\mathtt{prm}}^L(\mathsf{state}, N, s, t, \{A_i\}_{i=1}^m)$; next, it computes $\{z_i\}_{i=1}^m$ as described above, using the challenge $\{e_i\}_{i=1}^m$. Finally, it outputs the proof $(\{A_i\}_{i=1}^m, \{z_i\}_{i=1}^m)$. We write the resulting function as $\mathtt{ProveNI}_{\mathtt{prm}}^L(\mathsf{state}, (N, s, t), (\phi, \lambda))$.

– A party verifies a proof $\psi = (\{A_i\}_{i=1}^m, \{z_i\}_{i=1}^m)$ for (N, s, t) by setting

$$\{e_i\}_{i=1}^m = \mathtt{ChallengeNI}_{\mathtt{prm}}^L(\mathsf{state}, N, s, t, \{A_i\}_{i=1}^m)$$

and then verifying as described above, using the challenge $\{e_i\}_{i=1}^m$. We write the resulting function as $\mathtt{VerifyNI}_{\mathtt{prm}}^L(\mathsf{state}, (N, s, t), \psi)$.

3.5 $\Pi^{\log*}$: Group Element vs. Paillier Encryption in Range

The prover and verifier agree on shared state state, auxiliary data $R_j = (N_j, s_j, t_j)$ 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_i, C, X, B) with $C \in \mathbb{Z}_{N_i^2}^*$ and $X, B \in \mathbb{E}$, and the prover additionally has secret input (x, ρ) such that $x \in \pm 2^\ell$, $C = \mathtt{enc}_{N_i}(x; \rho)$, and $X = x \cdot B$. In all the cases where this proof is used in the protocol, the prover knows the factorization of N_i (and hence knows sk_i) and the verifier knows the factorization of N_j (and hence knows sk_j).

3.5.1 Interactive Version of the Proof

1. In the first round of the protocol, the prover does the following:
   - The prover samples the following values:
     $\alpha \leftarrow \pm 2^{\ell+\varepsilon}$
     $\mu \leftarrow \pm(2^\ell \cdot N_j)$
     $r \leftarrow \mathbb{Z}_{N_0}^*$
     $\gamma \leftarrow \pm(2^{\ell+\varepsilon} \cdot N_j)$.
   - The prover then computes:
     - $S = s_j^x t_j^\mu \bmod N_j$
     - $A = \mathtt{enc}_{N_i}(\alpha; r)$ (this is computed as $\mathtt{enc}_{\mathsf{sk}_i}^{\mathsf{crt}}(\alpha; r)$ when sk_i is known)
     - $Y = \alpha \cdot B$
     - $D = s_j^\alpha t_j^\gamma \bmod N_j$.

     Note that S and D are computed using fixed-base multiexponentiations.
   - The prover sends first message (S, A, Y, D) and maintains local (secret) state (α, μ, r, γ).
2. The verifier chooses $e \leftarrow \pm Q$ and sends e to the prover.

3. On input (N_i, C, X, B), the challenge e, and local state that includes $(x, \rho), (\alpha, \mu, r, \gamma)$, the prover computes:

   $z_1 = \alpha + ex$

   $z_2 = r \cdot \rho^e \bmod N_i$

   $z_3 = \gamma + e\mu,$

   and sends (z_1, z_2, z_3) to the verifier.

4. Given (N_i, C, X, B), initial message (S, A, Y, D), challenge e, and response (z_1, z_2, z_3), the verifier accepts if and only if the following are true:

   $\mathtt{enc}_{N_i}(z_1; z_2) = A \oplus (e \odot C) \bmod N_i^2$
   $z_1 \cdot B = Y + e \cdot X$
   $s_j^{z_1} t_j^{z_3} = D \cdot S^e \bmod N_j$
   $z_1 \in \pm 2^{\ell+\varepsilon}$.

3.5.2 Non-Interactive Version of the Proof

– We deterministically derive a challenge by hashing inputs that include state, the auxiliary data R_j, the common input (N_i, C, X, B), and the initial protocol message (S, A, Y, D). We write the resulting function as $e = \mathtt{ChallengeNI}^{\mathbb{E},L}_{\mathtt{log*}}(\mathsf{state}, R_j, (N_i, C, X, B), (S, A, Y, D))$.

– The prover generates a proof by first computing its initial message (S, A, Y, D) as described above, then computing $e = \mathtt{ChallengeNI}^{\mathbb{E},L}_{\mathtt{log*}}(\mathsf{state}, R_j, (N_i, C, X, B), (S, A, Y, D))$, and next computing (z_1, z_2, z_3) as described above, using challenge e. It outputs the proof $((S, A, Y, D), (z_1, z_2, z_3))$. We write the resulting function as $\mathtt{ProveNI}^{\mathbb{E},L}_{\mathtt{log*}}(\mathsf{state}, R_j, (N_i, C, X, B); (x, \rho))$.

– A party verifies a proof $\psi - ((S, A, Y, D), (z_1, z_2, z_3))$ by first computing

$$e = \mathtt{ChallengeNI}^{\mathbb{E},L}_{\mathtt{log*}}(\mathsf{state}, R_j, (N_i, C, X, B), (S, A, Y, D))$$

  and then verifying as described above, using the challenge e. We write the resulting function as $\mathtt{VerifyNI}^{\mathbb{E},L}_{\mathtt{log*}}(\mathsf{state}, R_j, (N_i, C, X, B), \psi)$.

3.6 Π^{fac}: No Small Factor Proof

The prover and verifier agree on shared state 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, and the prover additionally has primes $2^\ell < p, q < \pm 2^\ell \cdot \sqrt{N_i}$ with $N_i = pq$. In all the cases where this proof is used in the protocol, the verifier knows the factorization of N_j (and hence knows sk_j).

3.6.1 Interactive Version of the Proof

1. In the first round of the protocol, the prover does the following:

   – The prover samples the following values:
     $\alpha, \beta \leftarrow \pm \left(2^{\ell+\varepsilon}\sqrt{N_i}\right)$
     $\mu, \nu \leftarrow \pm \left(2^\ell N_j\right)$

$\sigma \leftarrow \pm\left(2^{\ell} N_i N_j\right)$
$r \leftarrow \pm\left(2^{\ell+\varepsilon} N_i N_j\right)$
$x, y \leftarrow \pm\left(2^{\ell+\varepsilon} N_j\right)$.

   – The prover then computes:
      – $P = s_j^{p} t_j^{\mu} \bmod N_j$
      – $Q = s_j^{q} t_j^{\nu} \bmod N_j$
      – $A = s_j^{\alpha} t_j^{x} \bmod N_j$
      – $B = s_j^{\beta} t_j^{y} \bmod N_j$
      – $T = Q^{\alpha} t_j^{r} \bmod N_j$.

     Note that P, Q, A, B are computed using fixed-base multiexponentiations.
   – The prover sends the first message (P, Q, A, B, T, σ) and also maintains local (secret) state $(\alpha, \beta, \mu, \nu, r, x, y)$.

2. The verifier chooses $e \leftarrow \pm Q$ and sends e to the prover.

3. On input N_i, the challenge e, and local state that includes (p, q), σ, and $(\alpha, \beta, \mu, \nu, r, x, y)$, the prover computes:
   $z_1 = \alpha + ep$
   $z_2 = \beta + eq$
   $w_1 = x + e\mu$
   $w_2 = y + e\nu$
   $v = r + e \cdot (\sigma - \nu p)$,

   and sends (z_1, z_2, w_1, w_2, v) to the verifier.

4. Given N_i, initial message (P, Q, A, B, T, σ), challenge e, and response (z_1, z_2, w_1, w_2, v), the verifier accepts if and only if the following are true:
   – $s_j^{z_1} t_j^{w_1} = A \cdot P^{e} \bmod N_j$
   – $s_j^{z_2} t_j^{w_2} = B \cdot Q^{e} \bmod N_j$
   – $Q^{z_1} t_j^{v} = T \cdot (s_j^{N_i} t_j^{\sigma})^{e} \bmod N_j$
   – $z_1 \in \pm\left(2^{\ell+\varepsilon}\sqrt{N_i}\right)$
   – $z_2 \in \pm\left(2^{\ell+\varepsilon}\sqrt{N_i}\right)$.

   Note that the 1st and 2nd checks involve fixed-base multiexponentiations.

3.6.2 Non-Interactive Version of the Proof

– We deterministically derive a challenge by hashing inputs that include `state`, the auxiliary data R_j, the common input N_i, and the initial protocol message (P, Q, A, B, T, σ). We write the resulting function as $e = \texttt{ChallengeNI}^{L}_{\texttt{fac}}(\texttt{state}, R_j, N_i, (P, Q, A, B, T, \sigma))$.

– A proof is computed as follows: compute initial message (P, Q, A, B, T, σ) as described above; compute $e = \texttt{ChallengeNI}^{L}_{\texttt{fac}}(\texttt{state}, R_j, N_i, (P, Q, A, B, T, \sigma))$; next compute (z_1, z_2, w_1, w_2, v) as described above, using challenge e. Output the proof $((P, Q, A, B, T, \sigma), (z_1, z_2, w_1, w_2, v))$. We write the resulting function as $\texttt{ProveNI}^{L}_{\texttt{fac}}(\texttt{state}, R_j, N_i, (p_i, q_i))$.

– A party verifies a proof $\phi = ((P, Q, A, B, T, \sigma), (z_1, z_2, w_1, w_2, v))$ by first computing

$$e = \texttt{ChallengeNI}^{L}_{\texttt{fac}}(\mathsf{state}, R_j, N_i, (P, Q, A, B, T, \sigma))$$

and then verifying as described above, using the challenge e. We write the resulting function as $\texttt{VerifyNI}^{L}_{\texttt{fac}}(\mathsf{state}, R_j, N_i, \phi)$.

3.7 Π^{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.

– $\texttt{Commit}_{\texttt{sch}}() \to (A; \alpha)$
$\alpha \leftarrow \mathbb{Z}_q$
$A = \alpha \cdot G$
return (α, A)

– $\texttt{Challenge}_{\texttt{sch}}() \to e$
return $e \leftarrow \mathbb{Z}_q$

– $\texttt{Prove}_{\texttt{sch}}(\alpha, e, x) \to z$
return $z = \alpha + ex \bmod q$

– $\texttt{Verify}_{\texttt{sch}}(z, A, e, X)$
accept iff $z \cdot G = A + e \cdot X$.

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.

Input. Party index i, context separation string $\mathtt{sid}$, security level L.

Round 1.

- Generate 4κ-bit safe primes p_i, q_i using the algorithm from Section 2.1.
- Compute $N_i = p_i q_i$ and $\phi = (p_i - 1)(q_i - 1)$, and create a Paillier decryption key sk_i using those values.
- Sample $r \leftarrow \mathbb{Z}^*_{N_i}$ and $\lambda \leftarrow \mathbb{Z}_\phi$, and compute $t_i = r_i^2 \bmod N_i$ and $s_i = t_i^\lambda \bmod N_i$.
- Compute $\hat{\psi}_i = \mathtt{ProveNI}^L_{\mathtt{prm}}(\mathtt{sid}||i, (N_i, s_i, t_i), (\phi, \lambda))$.
- Sample $\rho_i, u_i \leftarrow \{0,1\}^\kappa$, and compute $V_i = H(\mathtt{sid}\|n\|i\|N_i\|s_i\|t_i\|\hat{\psi}_i|\rho_i\|u_i)$.
- Send V_i to all parties.

Round 2.

- Receive V_j from all parties.
- (**Reliability check.**) Optionally, if the reliability check is enabled:
  - Compute $h_i = H(V_0\|\ldots\|V_{n-1})$ and send h_i to all parties.
  - Upon receiving h_j from all parties, abort if $h_i \neq h_j$ for some $j \in [n]$.
- Send $(N_i, s_i, t_i, \hat{\psi}_i, \rho_i, u_i)$ to all parties.

Round 3.

- Receive $(N_j, s_j, t_j, \hat{\psi}_j, \rho_j, u_j)$ from all parties.
- For all $j \in [n]$, set $R_j = (N_j, s_j, t_j)$; let $\vec{R} = (R_j)_{j\in[n]}$.
- For $j \neq i$:
  - Assert $V_j = H(\mathtt{sid}\|n\|j\|N_j\|s_j\|t_j\|\hat{\psi}_j\|\rho_j\|u_j)$.
  - Assert N_j is at least $8 \cdot \kappa - 1$ bits in length
  - Assert $\mathtt{VerifyNI}^L_{\mathtt{prm}}(\mathtt{sid}||j, (N_j, s_j, t_j), \hat{\psi}_j)$.
  - Construct Paillier encryption key from N_j.
- Compute $\rho = \bigoplus_j \rho_j$.
- Compute $\psi_i = \mathtt{ProveNI}^L_{\mathtt{mod}}(\mathtt{sid}||i\|\rho, N_i, (p_i, q_i))$.
- For $j \neq i$ do:
  - Compute $\phi_i^j = \mathtt{ProveNI}^L_{\mathtt{fac}}(\mathtt{sid}||i\|\rho, R_j, N_i, (p_i, q_i))$.
  - Send (ψ_i, ϕ_i^j) to P_j.

Output.

- Receive (ψ_j, ϕ_j^i) from all parties.

- For $j \neq i$ do:
  - Assert $\texttt{VerifyNI}^L_{\texttt{mod}}(\texttt{sid}||j\|\rho, N_j, \psi_j)$.
  - Assert $\texttt{VerifyNI}^L_{\texttt{fac}}(\texttt{sid}||j\|\rho, R_i, N_j, \phi^i_j)$.
- For $j \in [n]$ (including $j = i$), precompute a fixed-based multiexponentiation table T_j as described in Section 2.4. Let $\vec{T} = (T_j)_{j\in[n]}$.
- Return $(p_i, q_i, \vec{R}, \vec{T})$.

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 [1].

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

This protocol is based on [1, 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, number of signers n, context separation string $\texttt{sid}$, security level L, curve $\mathbb{E}$ with generator G of prime order q.

Round 1.

- Sample $x_i \leftarrow \mathbb{Z}_q$, and set $X_i = x_i \cdot G$.
- Sample $\mathsf{rid}_i \leftarrow \{0,1\}^\kappa$.
- Compute $(A_i; \tau_i) = \texttt{Commit}_{\texttt{sch}}()$.
- **(HD-wallets.)**
  - If HD-wallets support enabled, sample local chain code contribution $c_i \leftarrow \{0,1\}^{256}$ (32-bytes string)
  - Otherwise, set $c_i = \perp$
- Sample $u_i \leftarrow \{0,1\}^\kappa$ and set $V_i = H(\texttt{sid}\|n\|i\|\mathsf{rid}_i\|X_i\|A_i\|u_i\|c_i)$.
- Send V_i to all parties.

Round 2. Upon receiving V_j from all parties:

- **(Reliability check.)** Optionally, if the reliability check is enabled:
  - Compute $h_i = H(V_0\|\ldots\|V_{n-1})$ and send h_i to all parties.
  - Upon receiving h_j from all other parties: abort if $h_i \neq h_j$ for some $j \in [n]$.
- Send $(\mathsf{rid}_i, X_i, A_i, u_i, c_i)$ to all parties.

Round 3. Upon receiving $(\mathsf{rid}_j, X_j, A_j, u_j, c_j)$ from all other parties:

- Abort if $V_j \neq H(\texttt{sid}\|n\|j\|\mathsf{rid}_j\|X_j\|A_j\|u_j\|c_j)$ for some $j \in [n]$.
- Set $\mathsf{rid} = \bigoplus_j \mathsf{rid}_j$.

- **(HD-wallets.)** If HD-wallets support enabled:
  - Set chain code $c = \bigoplus_j c_j$
- Set $e_i = H(\mathtt{sid}\|i\|\mathsf{rid}\|X_i\|A_i)$ and compute $\psi_i = \mathtt{Prove}_{\mathtt{sch}}(\tau_i, e_i, x_i)$.
- Send ψ_i to all parties.

Output. Upon receiving ψ_j from all other parties:

- For all $j \neq i$:
  - Set $e_j = H(\mathtt{sid}\|j\|\mathsf{rid}\|X_j\|A_j)$.
  - Assert $\mathtt{Verify}_{\mathtt{sch}}(\psi_j, A_j, e_j, X_j)$.
- Set $X = \sum_j X_j$, $\vec{X} = (X_j)_{j\in[n]}$.
- Output $(X, x_i, \vec{X}, c)$.

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

Input. Party index i, threshold parameter t, number of signers n, context separation string $\mathtt{sid}$, security level L, curve E with generator G of prime order q.

Round 1.

- Sample $s_{i,0}, \dots, s_{i,t-1} \leftarrow \mathbb{Z}_q$. 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 $F_i(x) = f(x) \cdot G$.
- Compute $\sigma_{i,j} = f_i(j+1)$ for all $j \in [n]$.
- Sample $\mathsf{rid}_i \leftarrow \{0,1\}^\kappa$.
- Compute $(A_i, \tau_i) \leftarrow \mathtt{Commit}_{\mathtt{sch}}()$.
- **(HD-wallets.)**
  - If HD-wallets support enabled, sample local chain code contribution $c_i \leftarrow \{0,1\}^{128}$ (32-bytes string)
  - Otherwise, set $c_i = \perp$
- Sample $u_i \leftarrow \{0,1\}^\kappa$ and compute $V_i = H(\mathtt{sid}\|n\|i\|t\|\mathsf{rid}_i\|\vec{S}_i\|A_i\|u_i\|c_i)$.
- Send V_i to all parties.

Round 2. Upon receiving V_j from all parties:

- **(Reliability check.)** Optionally, if the reliability check is enabled:
  - Compute $h_i = H(V_0\|\cdots\|V_{n-1})$, and send h_i to all parties.
  - Upon receiving h_j 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. Upon receiving $(\mathsf{rid}_j, \vec{S}_j, A_j, u_j, c_j)$ and $\sigma_{j,i}$ from all parties:

- For each party $j \neq i$:

– Check that $\vec{S}_j$ has length t.
– Assert $V_j = H(\mathtt{sid}\|n\|j\|t\|\mathsf{rid}_j\|\vec{S}_j\|A_j\|u_j\|c_j)$.
– Define $F_j(x) = \sum_{k\in[t]} x^k \cdot S_{j,k}$.
– Assert $\sigma_{j,i} \cdot G = F_j(i+1)$.

– Compute $\mathsf{rid} = \bigoplus_{j\in[n]} \mathsf{rid}_j$.
– **(HD-wallets.)** If HD-wallets support enabled:
  – 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 $e_i = H(\mathtt{sid}\|i\|\mathsf{rid}\|X_i\|A_i)$.
– Compute Schnorr proof $\psi_i = \mathtt{Prove}_{\mathtt{sch}}(\tau_i, e_i, \sigma_i)$.
– Send ψ_i to all parties.

Output. Upon receiving ψ_j from all parties:

– For $j \neq i$: set $e_j = H(\mathtt{sid}|j\|\mathsf{rid}\|X_j\|A_j)$ and assert $\mathtt{Verify}_{\mathtt{sch}}(\psi_j, A_j, e_j, X_j)$.
– Compute $Y = \sum_{j\in[n]} S_{j,0}$.
– Create identity mapping $I : [n] \to \mathbb{Z}_q \setminus 0$, $I(i) = i+1$.
– 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 [1], 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 [1, Figure 7], although we have corrected some typos and eliminated some extraneous parts. (In particular, we do not have identifiable abort.)

Input. Number of parties n, party index $i \in [n]$, secret share x_i, list of signers' public-key shares $\vec{X} = \{X_j\}_{j\in[n]}$, Paillier private key sk_i, list of signers' auxiliary data $\vec{R} = ((s_j, t_j, N_j))_{j\in[n]}$, context separation string $\mathtt{sid}$, security level L, elliptic curve $\mathbb{E}$ with generator G.

Round 1.

– Sample $k_i, \gamma_i \leftarrow \mathbb{Z}_q$, $\rho_i, v_i \leftarrow \mathbb{Z}^*_{N_i}$, and set $G_i = \mathtt{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(\gamma_i; v_i)$, $K_i = \mathtt{enc}^{\mathsf{crt}}_{\mathsf{sk}_i}(k_i; \rho_i)$.
– For $j \neq i$ compute $\psi^0_{j,i} = \mathtt{ProveNI}^L_{\mathtt{enc}}(\mathtt{sid}\|i, R_j, (N_i, K_i), (k_i, \rho_i))$.

- Send (K_i, G_i) to all parties, and for $j \neq i$ send $\psi^0_{j,i}$ to P_j.

Round 2. Upon receiving $(K_j, G_j, \psi^0_{i,j})$ from all parties, do:

- **(Reliability check.)** Optionally, if the reliability check is enabled:
  - Compute $h_i = H(K_0 \| G_0 \| \cdots \| K_{n-1} \| G_{n-1})$ and send h_i to all parties.
  - Upon receiving h_j from all parties, abort if $h_i \neq h_j$ for some $j \in [n]$.
- For $j \neq i$, assert $\texttt{VerifyNI}^L_{\texttt{enc}}(\texttt{sid} \| j, R_i, (N_j, K_j), \psi^0_{i,j})$.
- Compute $\Gamma_i = \gamma_i \cdot G$.
- For $j \neq i$ do:
  - Sample $r_{i,j}, \hat{r}_{i,j} \leftarrow \mathbb{Z}_{N_i}$, $s_{i,j}, \hat{s}_{i,j} \leftarrow \mathbb{Z}_{N_j}$, and $\beta_{i,j}, \hat{\beta}_{i,j} \leftarrow \pm 2^{\ell'}$.
  - Compute $D_{j,i} = (\gamma_i \odot K_j) \oplus \texttt{enc}_{N_j}(-\beta_{i,j}; s_{i,j})$ and $F_{j,i} = \texttt{enc}^{\texttt{crt}}_{\texttt{sk}_i}(-\beta_{i,j}; r_{i,j})$.
  - Compute $\hat{D}_{j,i} = (x_i \odot K_j) \oplus \texttt{enc}_{N_j}(-\hat{\beta}_{i,j}; \hat{s}_{i,j})$ and $\hat{F}_{j,i} = \texttt{enc}^{\texttt{crt}}_{\texttt{sk}_i}(-\hat{\beta}_{i,j}; \hat{r}_{i,j})$.
  - Compute $\psi_{j,i} = \texttt{ProveNI}^{\mathbb{E},L}_{\texttt{aff-g}}(\texttt{sid} \| i, R_j, (N_j, N_i, K_j, D_{j,i}, F_{j,i}, \Gamma_i); (\gamma_i, -\beta_{i,j}, s_{i,j}, r_{i,j}))$ and $\hat{\psi}_{j,i} = \texttt{ProveNI}^{\mathbb{E},L}_{\texttt{aff-g}}(\texttt{sid} \| i, R_j, (N_j, N_i, K_j, \hat{D}_{j,i}, \hat{F}_{j,i}, X_i); (x_i, -\hat{\beta}_{i,j}, \hat{s}_{i,j}, \hat{r}_{i,j}))$.
  - Compute $\psi'_{j,i} = \texttt{ProveNI}^{\mathbb{E},L}_{\texttt{log*}}(\texttt{sid} \| i, R_j, (N_i, G_i, \Gamma_i, G); (\gamma_i, v_i))$.
  - Send $(\Gamma_i, D_{j,i}, F_{j,i}, \hat{D}_{j,i}, \hat{F}_{j,i}, \psi_{j,i}, \hat{\psi}_{j,i}, \psi'_{j,i})$ to P_j.

Round 3.

1. Upon receiving $(\Gamma_j, D_{i,j}, F_{i,j}, \hat{D}_{i,j}, \hat{F}_{i,j}, \psi_{i,j}, \hat{\psi}_{i,j}, \psi'_{i,j})$ from P_j, do:
   - Assert $\texttt{VerifyNI}^{\mathbb{E},L}_{\texttt{aff-g}}(\texttt{sid} \| j, R_i, (N_i, N_j, K_i, D_{i,j}, F_{i,j}, \Gamma_j), \psi_{i,j})$.
   - Assert $\texttt{VerifyNI}^{\mathbb{E},L}_{\texttt{aff-g}}(\texttt{sid} \| j, R_i, (N_i, N_j, K_i, \hat{D}_{i,j}, \hat{F}_{i,j}, X_j), \hat{\psi}_{i,j})$.
   - Assert $\texttt{VerifyNI}^{\mathbb{E},L}_{\texttt{log*}}(\texttt{sid} \| j, R_i, (N_j, G_j, \Gamma_j, G), \psi'_{i,j})$.
2. Compute $\Gamma = \sum_{j \in [n]} \Gamma_j$ and $\Delta_i = k_i \cdot \Gamma$.
3. For $j \neq i$, do:
   - Compute $\alpha_{i,j} = \texttt{dec}_{(p_i,q_i)}(D_{i,j})$ and $\hat{\alpha}_{i,j} = \texttt{dec}_{(p_i,q_i)}(\hat{D}_{i,j})$.
   - Compute $\psi''_{j,i} = \texttt{ProveNI}^{\mathbb{E},L}_{\texttt{log*}}(\texttt{sid} \| i, R_j, (N_i, K_i, \Delta_i, \Gamma); (k_i, \rho_i))$.
4. Compute $\delta_i = \gamma_i k_i + \sum_{j \neq i} (\alpha_{i,j} + \beta_{i,j}) \bmod q$ and $\chi_i = x_i k_i + \sum_{j \neq i} (\hat{\alpha}_{i,j} + \hat{\beta}_{i,j}) \bmod q$.
5. Send $(\delta_i, \Delta_i, \psi''_{j,i})$ to each P_j.

Output

1. Upon receiving $(\delta_j, \Delta_j, \psi''_{i,j})$ from P_j, assert $\texttt{VerifyNI}^{\mathbb{E},L}_{\texttt{log*}}(\texttt{sid} \| j, R_i, (N_j, K_j, \Delta_j, \Gamma), \psi''_{i,j})$.
2. Compute $\delta = \sum_j \delta_j$, and do:
   - Assert $\delta \cdot G = \sum_j \Delta_j$.
   - Set $R = \delta^{-1} \cdot \Gamma$ and output (R, k_i, χ_i).

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

Input. Size of signing set t, identities of parties in signing set, index $i \in [t]$, index map[4] $S : [t] \to [n]$, key share $K_{S(i)}$, context separation string `sid`, security level L, curve $\mathbb{E}$ with generator G.

HD-wallets inputs: additive shift $\in \mathbb{F}_q$ [5] (shift $= 0$ disables HD derivation)

Setup. The key share $K_{S(i)}$ contains `min_signers`, number of key holders n, secret share $x'_{S(i)}$, parties' public shares $\vec{X}' = (X'_j)_{j\in[n]}$, a map $I : [n] \to \mathbb{F}_q \setminus \{0\}$, Paillier secret key $\mathsf{sk}_{S(i)}$, parties' Paillier keys $\vec{N}' = (N'_j)_{j\in[n]}$, and parties' auxiliary information $\vec{R}' = (s_j, t_j, \hat{N})_{j\in[n]}$.

Step 1. Set $\mathsf{sk}_i = \mathsf{sk}_{S(i)}$ and $\vec{R} = (R'_{S(j)})_{j\in[n]}$. Then:

- If shares are additive[6] shares of the private key, set $x_i = x'_{S(i)}$, $\vec{X} = (X'_{S(j)})_{j\in[n]}$.
- If shares are Shamir secret shares of the private key:
  - For $j \in [t]$, compute Lagrange coefficient $\lambda_j = \prod_{m\in[t]\setminus\{j\}} \frac{I(S(m))}{I(S(m))-I(S(j))} \bmod q$.
  - Compute $x_i = \lambda_i \cdot x'_{S(i)}$.
  - For $j \in [n]$, compute $X_j = \lambda_j \cdot X'_{S(j)}$; then set $\vec{X} = \{X_j\}_{j\in[t]}$.

Step 2. (HD-wallets) If HD-wallets support enabled:

- Set $X_0 := X_0 + \text{shift} \cdot G$
- If $i = 0$, set $x_0 := x_0 + \text{shift}$
- Note: output signature will be valid for public key $Y + \text{shift} \cdot G$

Step 3. Call the non-threshold presigning protocol from the previous section using inputs $t, i, x_i, \vec{X}$, sk_i, $\vec{R}$, `sid`, $L, \mathbb{E}$.

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.

Local signing. The input is a presignature $S_i = (R, k_i, \chi_i)$, a hashed message m, and, if HD wallets support is enabled, additive shift $\in \mathbb{F}_q$ [5] (shift $= 0$ disables HD derivation). Do:

- If **HD-wallets** support enabled:
  Set $\chi := \chi + k \cdot \text{shift}$
- Set $r = R|_x$ and $\sigma_i = km + r\chi$.
- Output (r, σ_i).

[4] $S(i)$ is the index that P_i had at key-generation time.
[5] Deriving additive shift is up to specific standard of HD-wallets, e.g. see [4] or [2]
[6] In this case we have $t = n$.

Combining presignatures. The input is the public key Y, partial signatures $\{(r_i, \sigma_i)\}_{i=0}^{n-1}$, and hashed message m. The function does:

– Assert $r_0 = r_1 = \cdots = r_{n-1}$.

– Let $\sigma = \sum_j \sigma_j$.

– If (r, σ) is not a valid signature on hashed message m with respect to public key Y, then abort. Otherwise, output (r, σ).

References

[1] 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`.

[2] 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`.

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

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