0.1 MPC protocol

We now describe our implementation of an actively secure MPC protocol.

0.1.1 Commitments (commit.rs)

Our implementation of the commitment functionality that is used in the online phase can be found in commit.rs.

commit

The commit function simply takes two Vec<u8> values called v and r, along with a PlayerState state as input. Then v is the value that we wish to commit to, while r is the randomness that we use when committing.

In this method we utilize the implementation of sha256 found in the sha2 package to hash the concatenation of v and r, which yields the commitment c.

c is then broadcast to all players using the facilitator.

open

This function takes a commitment c and a value o as input, and these are both Vec<u8>. The o value is then supposed to satisfy that $o = v||r$.

When calling this function it hashes o using sha256, and then checks whether h(o) = c. If this is indeed the case, then we return Ok(o), and otherwise we return an error.

0.1.2 ZKPoPK (zk.rs)

In zk.rs the zero knowledge proof of plaintext is implemented for each party to run. The zero-knowledge proof has been implemented as in figure 10 of "multiparty computation from somewhat homomorphic encryption" [**?**]. The file is divided into 2 different functions. To generate a zero knowledge proof, we must call the function make_zkopk() which takes 6 arguments

- **params** The parameters of the PLWE instantiation.
- **x**: The plaintext for which we intend to generate the proof.
- **r**: The randomness to be used.
- **c**: The ciphertexts generated in the preprocessing protocol.
- **diag**: A boolean to indicate whether the diagonal element should be checked.
- **Pk**: The public key to be used.

make_zkopk() will generate 3 values (a,z,T), which will be used in the verification of the proof. To verify the proof we call the function verify_zkopk, which takes as input 6 arguments as well

- **a**: value generated by the make_zkopk function.

- **z**: value generated by the make_zkopk function.
- **t**: value generated by the make_zkopk function.
- **c**: The ciphertexts generated in the preprocessing protocol.
- **params** The parameters of the PLWE instantiation.
- **Pk**: The public key to be used.

verify_zkopk will return a boolean indicating whether or not the proof was valid.

0.1.3 Preprocessing phase (prep.rs)

The file prep.rs contains all of the methods that are called in the preprocessing phase to generate values for the online phase.

The file also contains a type definition AngleShare, which for some value $\langle v \rangle$ is a pair of the form $(v_i, \gamma(v)_i)$ for some player i. Additionally, the file also contains a definition MulTriple, which contains three AngleShare values, and represent a players shares of the $\langle \cdot \rangle$ values in a multiplicative triple.

reshare

Reshare takes as input a Parameters struct params, a Ciphertext e_m, a PlayerState state , and an Enc enum enc, which can take on values NewCiphertext or NoNewCiphertext.

The first thing done in this method is to sample a value f_i uniformly from Z_t using sample_single from prob.rs. Then, f_i is encrypted to get e_f_i. The facilitator is then used to broadcast e_f_i, and subsequently receive the encrypted shares from the other parties, which are then homomorphically added to get e_f. Then, we compute e_m_plus_f = add(params, e_m, &e_f).

The ddec method from mod.rs is then called with e_m_plus_f as input to get the plaintext m_plus_f.

Then we set m_i to be $e_{m+f} - f_i \bmod t$ if the player calling the method is the player with index 0, and $-f_i \bmod t$ otherwise. If enc = NewCiphertext, we now encrypt m_plus_f using encrypt_det where we use a triple of 1-polynomials instead of the randomness to make encryption deterministic. Now, we use add from encrypt.rs to homomorphically subtract the encrypted shares e_{f_i} from the encryption of m_plus_f to get e_m_prime. Afterwards, we return (Some(e_m_prime), m_i)

If enc = NoNewCiphertext, we instead just return (None, m_i).

p_angle

The p_angle method takes a Parameters struct params, an Integer v_i, a Ciphertext e_v, and a PlayerState state as input.

The first thing done in p_angle is to homomorphically multiply e_v and state .e_alpha to get e_v_mul_alpha. Then, reshare is called to get gamma_i, which is a share of an Integer, namely the plaintext in e_v_mul_alpha. Lastly, the method outputs (v_i, gamma_i), which is a share of $\langle v \rangle$.

As can be seen from the implementation we omit the public δ value as done in [**?**], such that the MAC's now instead satisfy $\alpha v = \sum_i \gamma(v)_i$.

initialize

The initialize method takes a Parameters struct params, and a mutable PlayerState state as input.

First, we sample a uniformly random Integer from $[0,t)$ using the sample_single method with params.t as input, and set the alpha_i variable of the player state to this value. This represents the given players share of the global key. We then encrypt state.alpha_i to get an encrypted share e_alpha_i. Now, e_alpha_i is broadcast using state.facilitor, and each player then uses their facilitator to receive e_{α_i} from the other players. The e_{α_i}'s are then homomorphically added to get e_α, and the result is then assigned to the e_alpha variable of state. Finally, we run zpopk from zk.rs in a loop *sec* times.

Notice how we do not compute the personal keys β_i as done in [**?**]. As mentioned earlier these are not needed when we use the trick described in section **??**.

pair

The pair methods takes a Parameters struct params, and a PlayerState state as input.

The method first samples a uniformly random Integer r_i from Z_t. Now, r_i is encrypted to get e_r_i, which is the broadcast using the facilitator. All players then again use the facilitator to receive the encrypted shares from the other parties, which are then homomorphically added to compute e_r. Then, we call zkpopk with e_r_i as input.

Following this we compute the given players share of $\langle r \rangle$ with a call to p_angle with r_i and e_r as arguments, which returns r_angle.

Again, we use the trick explained in section **??**, so we don't need the values in the bracket representation, and therefore we just return (r_i, r_angle).

triple

The triple method takes a Parameters struct params, along with a PlayerState state as arguments.

First, we sample a_i, b_i uniformly at random from Z_t, which are both then encrypted, and then encrypted shares are then broadcast.

Then, when the player receives the encrypted shares of a and b from the other players, then these are homomorphically added to get e_a and e_b.

Following this, we generate shares a_angle and b_angle with calls to p_angle using a_i, e_a and b_i, e_b as input respectively.

Now, the player calling the method has shares of $\langle a \rangle$ and $\langle b \rangle$, and we need to compute a share of $\langle c \rangle$.

To do this we compute e_c by homomorphically multiplying e_a and e_b, and then we call reshare with e_c and NewCiphertext as input to get a new ciphertext e_c_prime and c_i, which is a share of c.

This allows us to call p_angle using c_i and e_c_prime to get c_angle.

Finally, we return a MulTriple containing a_angle, b_angle, and c_angle.

0.1.4 Online phase (prep.rs)

The code related to the online phase can be found in prep.rs.

give_input

This method takes a Parameters struct params, an Integer x_i, a (Integer , AngleShare) pair called r_pair , and a PlayerState state .

First, the method broadcasts a message BeginInput to indicate that the player calling the method wants to give some input.

Then, the player receives all shares of r from the other players using the facilitator, and opens r by computing $r = r_1 + \ldots + r_n \mod p$.

The value eps = x_i − r is then computed, and subsequently broadcast to all players.

The AngleShare corresponding to $\langle r \rangle + \varepsilon$ is then computed and returned.

receive_input

The receive_input method takes a Parameters struct params, an Integer x_i, a (Integer , AngleShare) pair called r_pair , and a PlayerState state .

The first thing that the method does is to receive a message using the facilitator, and if this is not BeginInput, then we panic.

Following this we take r_i from r_pair , and send it to the player p_i, which is the player that is providing input, and thus also the player that sent the initial BeginInput message.

Now, we receive ε from p_i, and then use this to compute and return the AngleShare corresponding to $\langle x_i \rangle = \langle r \rangle + \varepsilon$ for the given player.

add

Add simply takes two AngleShare values x and y as input.

These are then used to compute (x.0 + y.0, x.1 + y.1), and the resulting AngleShare is then returned.

partial_opening

To partially open some value v where player i holds share v_i, each player calls partial_opening with a Parameters struct params, an Integer to_share, and a PlayerState state as arguments. When partially opening we send all shares to some designated player, and in this case we simply let all players send their share to player P_1, who has index 0.

To do this we first send the calling players share of v, to_share, to P_1 using the facilitator. Player P_1 then computes $v = v_1 + \ldots + v_n \mod p$, which is subsequently broadcast to all players using the facilitator.

Finally, the method returns the value v received from P_1.

triple_check

The triple_check method takes a Parameters struct params, two MulTriple values abc_triple, fgh_triple, an Integer t_share and a mutable PlayerState state as arguments.

The first thing done is to use the facilitator to broadcast the players share of t, t_share. The player then uses the facilitator to receive shares of t from the other players, and then compute $t = t_1 + \ldots + t_n \mod p$.

Now, we compute $t \cdot \langle a \rangle - \langle f \rangle$ and save the resulting AngleShare in variable rho_share. Following this we call partial_opening with rho_share.0 as input to get ρ. Lastly, we push (rho, rho_share.1) to state.opened, to ensure that we check the MAC of ρ in maccheck.

Now, we use the same approach as for ρ to compute $\langle b \rangle - \langle g \rangle$ and store the resulting AngleShare in variable sigma_share. Then we call partial_opening with sigma_share.0 as input to get σ. Then we push (sigma, sigma_share.1) to state.opened.

We then compute $t \cdot \langle c \rangle - \langle h \rangle - \sigma \cdot \langle f \rangle - \rho \cdot \langle g \rangle - \sigma \cdot \rho$, call partial_opening with the resulting AngleShare as input to get the variable zero.

The last thing done is to check whether zero is 0. If this is the case, then we return Ok, otherwise we return Err.

multiply

The multiply method takes a Parameters struct params, two AngleShare values x and y, two MulTriple values abc_triple and fgh_triple, an Integer t_share, and a PlayerState state as arguments.

First, we call triple_check with params, abc_triple, fgh_triple, t_share, and state as input. If the call to triple_check returns Err, then we panic and abort the protocol, otherwise we proceed.

Following this we compute the AngleShare corresponding to $\langle x \rangle - \langle a \rangle$, so that we get eps_share. We then call partial_opening with eps_share.0 as input to get eps. Then we push (eps, eps_share.1) to state.opened.

Now we compute the AngleShare corresponding to $\langle y \rangle - \langle b \rangle$, so we get delta_share, then call partial_opening with delta_share.0 as input to get delta. And now we push (delta, delta_share.1) to state.opened.

Finally, we compute the AngleShare corresponding to $\langle z \rangle = \langle c \rangle + \varepsilon \langle b \rangle \delta \langle a \rangle + \varepsilon \delta$ and return it.

maccheck

The maccheck method takes a Parameters struct params, a Vec<(Integer, Integer)> of $(a_j, \gamma(a_j)_i)$ pairs named to_check, and a PlayerState state as arguments.

First, we use the rand package to sample random 32-byte values s_i and r.

Then we use the commit method from commit.rs, to commit to s_i using r as randomness.

Following this we use the facilitator to receive the commitments from all n parties. Then once all of the commitments have been received the parties broadcast the value $o = s_i || r$, such that all parties then open the commitments that they have received and get the seed s_i for all players i.

Now, we XOR the received seeds to get $s = s_1 \oplus ... \oplus s_n$. This is followed by using the rand package and the seed s to sample a vector rng_seed with n entries and with values in the range $[0, p)$.

Then the method uses the values $a_1, ..., a_t$ stored in $\text{state}.\text{opened}$ to compute $a = \sum_{j=1}^{t} r_j a_j$.

After generating a we use it to compute $\gamma_i = \sum_{j=1}^{t} r_j \gamma(a_j)_i$ and $\sigma_i = \gamma_i - \alpha_i a$. Then we convert σ_i to bytes and store the result in sigma_i_bytes.

Afterwards we need to commit to sigma_i_bytes, and to do this we sample 32 bytes of randomness r as done earlier. This is followed by committing to sigma_i_bytes using r as randomness.

Now we use the facilitator to receive commitments from all n players, and once these have been received we broadcast $o = sigma\_i\_bytes || r$.

Then we use the o values received to open the σ commitments received earlier, such that we get bytes corresponding to σ_i from all players i. Then, we convert the bytes into values of the Integer type.

Finally, we add the n σ_i values received and check that indeed the sum is equal to $\text{Integer} :: \text{ZERO}$. If this is the case, then we return true, otherwise we return false.

output

The output method is called by a player when that player is ready to output some value $\langle y \rangle$. The method takes a Parameters struct params, AngleShare y_angle, and a PlayerState state as arguments.

To output a value we first call the maccheck method on $\text{state}.\text{opened}$, to check that the MAC values are correct for all of the values v in the $\langle r \rangle$ representation that have been opened. If this returns false, then the check was unsuccessful and we panic to abort the protocol.

Then we broadcast y_angle.0, which is the players share y_i of the output value y. This is followed by receiving shares from all players, and then computing $y = y_1 + ... + y_n \mod p$.

Now, we once again call the maccheck method but this time we use the $(\text{y}, \text{y\_angle}.1)$ as input, which is the opened value y and the players corresponding MAC value. If this returns false, then we once again panic to abort the protocol.

If we did not abort at any point during the method, then we return the final result y.