Vote

In this assignment, you'll implement a cryptographic voting protocol based on the widely used Helios protocol. In particular, you'll explore how we can use zero-knowledge proofs, blind signatures, and homomorphic encryption in practice.

Contents

1 Background Knowledge **2**

1.1 Additively Homomorphic Encryption 2

1.2 Threshold Encryption 3

1.3 Zero-Knowledge Proofs 4

1.4 Blind Signatures 7

1.5 Some Resources 8

1.6 Putting It All Together 8

2 Assignment Specification **12**

2.1 Functionality 12

2.2 Support Code 15

2.3 Libraries: CryptoPP 15

3 Getting Started **16**

3.1 Running 16

3.2 Testing 16

1 Background Knowledge

In this assignment, you'll build a cryptographically secure voting platform. There are four programs involved:

1. A **Registrar** that checks that all voters are registered to vote only once
2. A **Tallyer** that posts votes on a public bulletin board
3. **Arbiters** that generate the election parameters and decrypt the final result
4. **Voters** that can vote, view, and verify the final result

All of these parties will interact to conduct an election.

We *highly* recommend reading Cryptographic Voting - A Gentle Introduction in full, or at least referencing it when writing your ZKPs. It will help immensely in understanding the mathematics of this assignment. **Note:** on page 35, while calculating the verifier challenges, it should be $c_0 := c - c_1$ and $c_1 := c - c_0$.

1.1 Additively Homomorphic Encryption

In standard encryption, encrypted data must be decrypted before it can be meaningfully altered. Indeed, being able to alter a ciphertext to produce a meaningful change in the corresponding plaintext is called **malleability**, and is usually undesirable. In particular, a malleable encryption scheme cannot be used in authenticated encryption. However, being able to compute over encrypted data could be very useful, as it allows multiple parties to compute over shared data without leaking the data itself or coordinating beforehand. Encryption schemes that allow for computation over their ciphertexts are called **homomorphic encryption** schemes. Of those, some may only allow either addition or multiplication (called **additively** and **multiplicatively** homomorphic, respectively), while those allowing both are called **fully** homomorphic.

In this project, we'll explore an additively homomorphic encryption scheme. Formally, an additively homomorphic encryption scheme is an encryption scheme with an additional algorithm HomAdd such that for any two messages m_0, m_1, we have that

$$\mathsf{Dec}(\mathsf{HomAdd}(\mathsf{Enc}(m_0), \mathsf{Enc}(m_1))) = m_0 + m_1$$

where HomAdd is a homomorphic addition operation. In other words, we can construct a ciphertext for $m_0 + m_1$ using ciphertexts for m_0 and m_1 individually. Similar definitions exist for multiplicatively homomorphic and fully homomorphic schemes.

We have actually already seen a simple *multiplicatively* homomorphic encryption scheme: ElGamal encryption. To see why, consider the two ciphertexts below for messages m_1 and m_2:

$$c_1 = (g^{r_1}, h^{r_1} \cdot m_1) \text{ and } c_2 = (g^{r_2}, h^{r_2} \cdot m_2).$$

Observe that we can construct a ciphertext for $m_1 \cdot m_2$ by multiplying component-wise to obtain

$$c = (g^{r_1+r_2}, h^{r_1+r_2} \cdot (m_1 \cdot m_2)).$$

We can apply the same idea to convert this encryption scheme into an *additively* homomorphic encryption scheme by instead encoding our messages as g^m instead of m; then, the above scheme becomes combining

$$c_1 = (g^{r_1}, h^{r_1} \cdot g^{m_1}) \text{ and } c_2 = (g^{r_2}, h^{r_2} \cdot g^{m_2})$$

into

$$c = (g^{r_1+r_2}, h^{r_1+r_2} \cdot g^{m_1+m_2}).$$

One glaring issue with this adaptation is that in order to decrypt and recover m_1+m_2 we need to solve the discrete logarithm problem. However, for our purposes, we will only be encrypting small values and combining them a small number of times, so a brute-force, linear-time approach is perfectly fine. Note that this doesn't compromise the security of encryption as the secret key sk and the random r's are still expected to be very large, so a brute-force approach without knowledge of sk is still computationally infeasible.

1.2 Threshold Encryption

Let's say we will be using homomorphic encryption that allows anyone to add their vote to a publicly tracked value. So far, a single party (arbiter) holds the decryption key sk and can check the value at any time they want. This isn't necessarily desirable: it would be nice if decryption keys could be split amongst multiple parties (arbiters) and a ciphertext can only be **jointly** decrypted by all the parties together. This is known as **threshold encryption** and is also achievable with ElGamal encryption.

In threshold ElGamal encryption, n parties (arbiters) will get together and each generate a keypair $(\mathsf{sk}_i, \mathsf{pk}_i)$ where $\mathsf{pk}_i = g^{\mathsf{sk}_i}$. Each party publishes pk_i and keeps sk_i private. They will then multiply their public values together and obtain $\mathsf{pk} = \prod_i \mathsf{pk}_i = g^{\sum_i \mathsf{sk}_i}$. Encryption should use this combined public key pk, and its corresponding secret key sk is **secret shared** among the n parties.

In order to jointly decrypt a ciphertext $c = (c_1, c_2)$ that is encrypted using this public key, each party can partially decrypt the ciphertext, and then the parties can combine their partial decryptions to get a full decryption. To compute a partial decryption of

c, each party computes $c_1^{\mathsf{sk}_i}$. Then, multiplying all partial decryptions together retrieves $\prod c_1^{\mathsf{sk}_i} = c_1^{\sum \mathsf{sk}_i} = c_1^{\mathsf{sk}}$, which can then be used to decrypt the second component of the ciphertext, namely $g^m = c_2/c_1^{\mathsf{sk}}$.

1.3 Zero-Knowledge Proofs

Let's say that our protocol allows voters to encrypt 1 or 0 and post it to the public bulletin board as a vote for or against a particular policy. How will we know that the voters haven't cheated and posted an encryption of 500, without decrypting every ciphertext and checking that it is, in fact, 1 or 0? **Zero-knowledge proofs** allow us to prove this fact, among many others, without revealing any other information. It is a powerful cryptographic tool that allows us to build trust without unnecessarily revealing information. We'll explore three zero-knowledge proof protocols to get the hang of things.

1.3.1 Proving Correct Encryption

The first ZKP we'll explore is a protocol to prove that a ciphertext $c = (c_1, c_2)$ is an ElGamal encryption of 0 under a public key pk, where the witness is the randomness r used in the encryption, in particular $c_1 = g^r$ and $c_2 = \mathsf{pk}^r$. We can think of (pk, c_1, c_2) as a Diffie-Hellman tuple with witness r. The protocol is as follows.

- **Message 1:** The prover samples a random r' from $\mathbb{Z}_q$ and sends $(A = g^{r'}, B = \mathsf{pk}^{r'})$ to the verifier.
- **Message 2:** The verifier chooses a random value σ from $\mathbb{Z}_q$ and sends it to the prover.
- **Message 3:** The prover sends back $r'' = r' + \sigma \cdot r \mod q$ to the verifier.
- **Verify:** Finally, the verifier verifies that $g^{r''} = A \cdot c_1^{\sigma}$ and $\mathsf{pk}^{r''} = B \cdot c_2^{\sigma}$.

Similarly, we can prove a ciphertext $c = (c_1, c_2)$ is an encryption of 1 under a public key pk, where the witness is the randomness r used in the encryption, in particular $c_1 = g^r$ and $c_2 = \mathsf{pk}^r \cdot g$. Notice that can re-write the ciphertext as $c_1 = g^r$ and $\mathbf{c_2/g} = \mathsf{pk}^{\mathbf{r}}$ and think of

$$(\mathsf{pk}, c_1, c_2/g)$$

as a new Diffie-Hellman tuple with witness r.

In other words, we can use the above ZKP to prove that $(c_1, c_2/g)$ is an encryption of 0 (with randomness r).

As we discussed in class, this **sigma protocol** satisfies completeness, is a proof of knowledge of r, and is honest-verifier zero-knowledge. A more detailed explanation can be found in the lecture notes and readings.

1.3.2 Proving OR Statement

The ZKP we need in the project is a proof that a ciphertext is an encryption of **either 0 or 1**. Proving AND statements is straightforward: simply prove both statements. However, proving OR statements is significantly more difficult since one of the statements could be false. We'll approach this ZKP in steps and build up to a protocol that works.

Consider the aforementioned ZKP that $c = (c_1, c_2)$ is an encryption of 0. Notice that the prover can actually cheat in the ZKP if she knows σ before sending the first-round message. In particular, she can first randomly sample the third-round reponse r'' from $\mathbb{Z}_q$, and then compute the first-round message by $A = g^{r''}/c_1^{\sigma}$ and $B = \mathsf{pk}^{r''}/c_2^{\sigma}$, which will end up verifying correctly. This is exactly how we prove honest-verifier zero-knowledge of the protocol. We can use this observation to generate a ZKP for an OR statement.

The prover wants to prove that a ciphertext $c = (c_1, c_2)$ is an encryption of either 0 or 1 (without revealing whether it is an encryption of 0 or 1). Suppose c is an encryption of 1 and the prover knows the randomness r. (If c is an encryption of 0, the protocol follows similarly.) At a high level, the prover will perform two ZKPs simultaneously, one proving c is an encryption of 0 and one proving c is an encryption of 1. For the one proving c is an encryption of 1 (which she has a witness), she behaves honestly; for the one proving c is an encryption of 0 (which she does not have a witness), she leverages a simulated protocol transcript. The protocol is as follows.

- **Prepare fake transcript:** The prover uses the above trick to "simulate" a valid sigma protocol for the (false) statement that $c = (c_1, c_2)$ is an encryption of 0. In particular, she randomly samples r_0'' and σ_0 from $\mathbb{Z}_q$, and then computes $A_0 = g^{r_0''}/c_1^{\sigma_0}$ and $B_0 = \mathsf{pk}^{r_0''}/c_2^{\sigma_0}$. She now has a "fake" transcript $((A_0, B_0), \sigma_0, r_0'')$ that verifies correctly.

- **Message 1:** The prover sends the first message for both ZKPs.

  - For the one proving c is an encryption of 0, she sends (A_0, B_0) from the simulated transcript.

  - For the one proving c is an encryption of 1, she samples r_1' from $\mathbb{Z}_q$ and sends

$(A_1 = g^{r'_1}, B_1 = \mathsf{pk}^{r'_1})$.

- **Message 2:** The verifier chooses a random value σ from $\mathbb{Z}_q$ and sends it to the prover.
- **Message 3:** The prover first computes $\sigma_1 = \sigma - \sigma_0 \mod q$, where σ_0 is from the simulated transcript.
  - For the proof that c is an encryption of 0, she sends back the challenge σ_0 along with the third message r''_0 from the simulated proof.
  - For the proof that c is an encryption of 1, she generates a response honestly based on the challenge σ_1 and her witness r, namely $r''_1 = r'_1 + \sigma_1 \cdot r \mod q$, and sends it back to the verifier along with σ_1.
- **Verify:** Finally, the verifier verifies the following:
  - $\sigma_0 + \sigma_1 = \sigma \mod q$.
  - For the proof that c is an encryption of 0, verify that $g^{r''_0} = A_0 \cdot c_1^{\sigma_0}$ and $\mathsf{pk}^{r''_0} = B_0 \cdot c_2^{\sigma_0}$.
  - For the proof that c is an encryption of 1, verify that $g^{r''_1} = A_1 \cdot c_1^{\sigma_1}$ and $\mathsf{pk}^{r''_1} = B_1 \cdot (c_2/g)^{\sigma_1}$.

This protocol is known as **Disjunctive Chaum-Pedersen** (DCP), or the **Sigma-OR** protocol. A more detailed explanation can be found in the lecture notes and readings.

1.3.3 ZKP for Partial Decryption

We explore another ZKP that proves that a partial decryption of a ciphertext $c = (c_1, c_2)$ is correct. Say that the partial decryption with regard to a partial public key pk_i is d_i. The prover wants to prove that d_i is a correct partial decryption of c with regard to pk_i, where the witness is the partial secret key sk_i. That is, $\mathsf{pk}_i = g^{\mathsf{sk}_i}$ and $d = c_1^{\mathsf{sk}_i}$. We can think of (c_1, pk_i, d) as a Diffie-Hellman tuple with witness sk_i. The protocol is as follows.

- **Message 1:** The prover samples a random r from $\mathbb{Z}_q$ and sends $(A = g^r, B = c_1^r)$ to the verifier.
- **Message 2:** The verifier chooses a random value σ from $\mathbb{Z}_q$ and sends it to the prover.

- **Message 3:** The prover sends back $s = r + \mathsf{sk}_i \cdot \sigma \mod q$ to the verifier.
- **Verify:** Finally, the verifier verifies that $g^s = A \cdot \mathsf{pk}_i^\sigma$ and $c_1^s = B \cdot d^\sigma$.

1.3.4 Non-Interactive Zero-Knowledge (NIZK)

All the zero-knowledge proofs we have discussed above are only honest-verifier zero-knowledge (HVZK); namely, it is zero-knowledge against an honest verifier who samples the challenge σ uniformly at random. The Fiat-Shamir heuristic allows us to transform these sigma protocols into non-interactive zero-knowledge (NIZK) proofs in the random oracle model. Instead of asking the verifier to sample σ, we compute σ from a hash function computed on the ZK statement along with the first-round message. For example, in ZKP for partial decryption, we compute $\sigma = H(\mathsf{pk}_i, c, d, A, B)$, where H is a hash function modeled as a random oracle.

1.4 Blind Signatures

Looking ahead, in our anonymous online voting protocol, each voter will be authorized by the registrar and obtain a signature certifying such authorization.

When the voter goes to the registrar, she will need to present her identity. In order to unlink the voter's encrypted vote from her identity, the voter will want her message m (encrypted vote) to be signed by the registrar, with the caveat that she doesn't want to the registrar to know what m is. This is what **blind signatures** seek to achieve.

At a high level, the requester (voter) can cryptographically *blind* the mssage before sending it to the signer (registrar). The signer then signs the *blinded message* and sends it back to the requester. Finally, the requester can *unblind* the signature to obtain a valid signature on the *original message*. The signer shouldn't be able to link the blinded message to the unblinded message or signature.

Surprisingly, our old friend RSA signatures can be modified to achieve blind signatures.

First, let's recall how RSA signatures work. Suppose the signer has a public verification key $\mathsf{vk} = (N, e)$ and a private signing key $\mathsf{sk} = d$. To sign a message m, the signer computes

$$\sigma = H(m)^d \mod N$$

To verify the signature, anyone can check if

$$H(m) = \sigma^e \mod N$$

This holds because

$$\sigma^e = (H(m)^d)^e = H(m)^{de} = H(m) \mod N$$

To achieve **blind signatures**, the requester must first hide the (hashed) message $H(m)$ from the signer. We do this by sampling a random r from $\mathbb{Z}_N^*$ and computing

$$m' = H(m) \cdot r^e \mod N$$

The requester sends m' to the signer, which effectively hides m and $H(m)$, who signs it as if it were the original (hashed) message. Specifically, the signer computes

$$\sigma' = (m')^d \mod N$$

and sends this signature σ' back to the requester. Note that

$$\sigma' = (H(m) \cdot r^e)^d = H(m)^d \cdot r^{ed} = H(m)^d \cdot r \mod N$$

As such, the requester can unblind the signature by computing

$$\sigma = \sigma' \cdot r^{-1} \mod N$$

To verify the signature, anyone can check if

$$H(m) = \sigma^e \mod N$$

1.5 Some Resources

The following papers are incredibly useful for gaining a full understanding of protocols like ours:

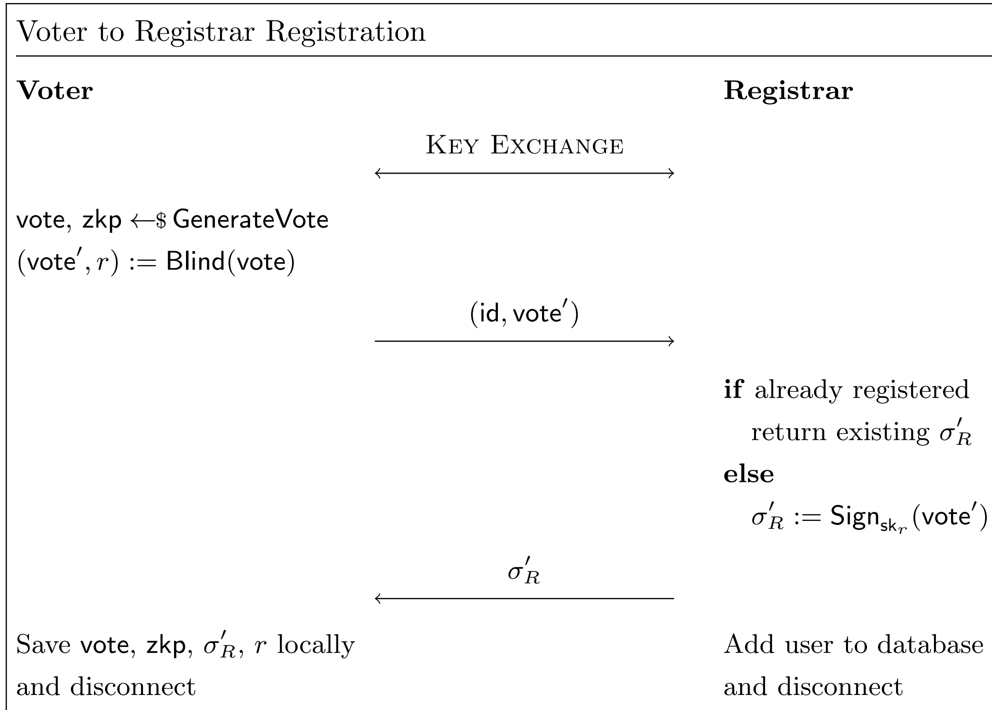
- Cryptographic Voting — A Gentle Introduction (pg. 31-35 are **extremely** helpful). **Note:** on page 35, while calculating the verifier challenges, it should be $c_0 := c - c_1$ and $c_1 := c - c_0$.
- Helios

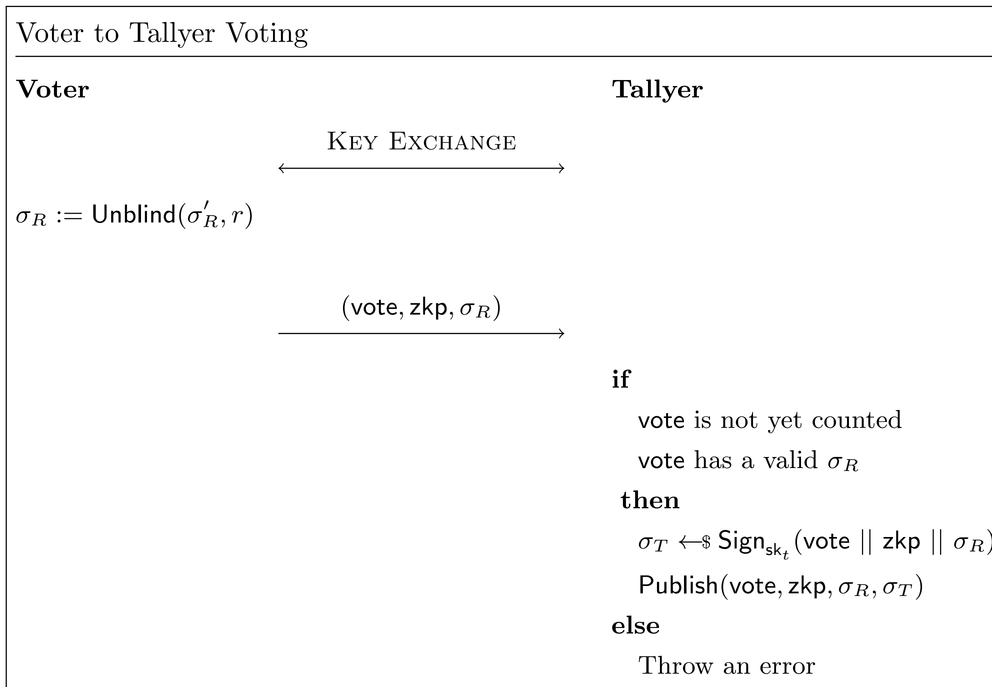
1.6 Putting It All Together

The following diagrams explain how the protocols work together from the perspective of a voter. The first presents a general overview of the system. The last two show the registration and voting process, respectively.

Overall Architecture

Voter	Registration	Registrar
		Check voter's registration
		Sign a blind signature on voter's vote
Voter	Vote	**Tallyer**
		Check voter's vote
		Publish vote, ZKP, signatures
	Everybody has voted	
		Arbiter Collect all votes Verify ZKPs and signatures for all votes Homomorphically add up votes Publish partial decryption and ZKP
	Arbiters have adjudicated	
Voter Collect all votes and partial decryptions Verify ZKPs and signatures for all votes Homomorphically add up votes Verify partial decryption ZKPs Combine partial decryptions to get election result		





In short, we proceed in the following steps: setup, registration, voting, and verification.

1.6.1 Setup

- To set up, each arbiter will generate an ElGamal keypair $(\mathsf{sk}_i, \mathsf{pk}_i)$ and publish pk_i.
- To obtain the election key, voters will multiply all of the arbiter pk_i together to obtain pk.

1.6.2 Registration

- On setup, the registrar has access to an RSA signature keypair $(\mathsf{vk}_r, \mathsf{sk}_r)$.
- First, the voter then encrypts their vote using pk and generates a ZKP that proves that the encrypted vote is either 0 or 1.
- Then, the voter blinds the encrypted vote and sends it to the registrar for signing.
- The registrar ensures that the voter hasn't voted before and signs their (blinded) encrypted vote using sk_r, and then sends the blind signature σ'_R back to the voter.

1.6.3 Voting

- On setup, the arbiters have generated an ElGamal public key pk.
- On setup, the tallyer has access to a RSA keypair $(\mathsf{vk}_t, \mathsf{sk}_t)$.
- The voter then unblinds the registrar's signature and sends the encrypted vote, ZKP, and unblinded signature from the registrar to the tallyer.
- The tallyer verifies the unblinded signature, and then signs the encrypted vote, ZKP, and unblinded signature using sk_t.
- The tallyer publishes the encrypted vote, ZKP, unblinded signature, and (tallyer's) signature on a public bulletin board.

1.6.4 Partial Decryption

- On setup, the arbiters have access to their partial ElGamal secret key sk_i (for the partial ElGamal public key pk_i).

- Arbiters verify the ZKPs, blind signatures, and tallyer's signatures on all votes.
- Compute the homomorphic addition of all valid votes.
- Each arbiter generates a partial decryption and ZKP using sk_i.
- The arbiter publishes the partial decryption and ZKP.
- Voters can recover the final result by combining all partial decryptions and verifying all ZKPs and signatures.

2 Assignment Specification

Please note: you may NOT change any of the function headers defined in the stencil. Doing so will break the autograder; if you don't understand a function header, please ask us what it means and we'll be happy to clarify.

2.1 Functionality

You will primarily need to edit `src/drivers/crypto_driver.cxx`, `src/pkg/arbiter.cxx`, `src/pkg/election.cxx`, `src/pkg/registrar.cxx`, `src/pkg/tallyer.cxx`, and `src/pkg/voter.cxx`. The following is an overview of relevant files:

- `src/cmd/arbiter.cxx` is the main entrypoint for the `auth_arbiter` binary. It calls the `Arbiter` class.
- `src/cmd/registrar.cxx` is the main entrypoint for the `auth_registrar` binary. It calls the `Registrar` class.
- `src/cmd/tallyer.cxx` is the main entrypoint for the `auth_tallyer` binary. It calls the `Tallyer` class.
- `src/cmd/voter.cxx` is the main entrypoint for the `auth_voter` binary. It calls the `Voter` class.
- `src/drivers/crypto_driver.cxx` contains all of the cryptographic protocols we use in this assignment.
- `src/pkg/arbiter.cxx` Implements the `Arbiter` class.

- `src/pkg/registrar.cxx` Implements the `Registrar` class.
- `src/pkg/tallyer.cxx` Implements the `Tallyer` class.
- `src/pkg/voter.cxx` Implements the `Voter` class.
- `src/pkg/election.cxx` Implements the `Election` class, which holds most of the interesting cryptographic operations.

The following roadmap should help you organize concerns into a sequence:

- **Registration**: Implement voter registration.
- **Vote Generation**: Implement vote generation and verification.
- **Partial Decryption**: Implement partial decryption and verification.

Some tips:

- Don't use CryptoPP's ElGamal library to implement (`EG_generate`). Instead, do keygen using `Integer`s and exponentiation. The `nbtheory` library is fair game!
- If you ever can't find a value, check if it's in one of the configs.
- Comments in the `messages.hpp` file should help figure out which fields are which in the ZKPs. We follow the convention in the Gentle Introduction linked above.
- Debugging ZKPs can be very hard! Inspect each value along the way to be sure that they are what you expect. In particular, ensure that values don't zero out.

2.1.1 Registration

Implement registration between the Voter and Registrar. Once you do so, Voters should be able to receive a blind signature on their encrypted vote.

Application functions:

- `RegistrarClient::HandleRegister(...)`
- `VoterClient::HandleRegister(...)`

2.1.2 Vote Generation

Implement vote generation and verification, then allow Voters to vote with the Tallyer. Once you do so, votes should be put into the database.

Cryptographic functions:

- `CryptoDriver::EG_generate()`
- `CryptoDriver::RSA_BLIND_blind(...)`
- `CryptoDriver::RSA_BLIND_unblind(...)`
- `ElectionClient::GenerateVote(...)`
- `ElectionClient::VerifyVoteZKP(...)`

Application functions:

- `TallyerClient::HandleTally(...)`
- `VoterClient::HandleVote(...)`

2.1.3 Partial Decryption

Implement partial decryption and verification, then allow Arbiters to partially decrypt the election result.

Cryptographic functions:

- `ElectionClient::PartialDecrypt(...)`
- `ElectionClient::VerifyPartialDecryptZKP(...)`
- `ElectionClient::CombineVotes(...)`
- `ElectionClient::CombineResults(...)`

Application functions:

- `ArbiterClient::HandleAdjudicate(...)`

- `VoterClient::DoVerify()`

2.2 Support Code

Read the support code header files before coding so you have a sense of what functionality we provide. This isn't a networking class, nor is it a software engineering class, so we try to abstract away as many of these details as we can so you can focus on the cryptography.

The following is an overview of the functionality that each support code file provides.

- Everything from prior assignments is unchanged. Hooray!

2.3 Libraries: CryptoPP

You may find the following functions useful:

- `CryptoPP::EuclideanMultiplicativeInverse`
- `CryptoPP::a_times_b_mod_c`

  Note that `CryptoPP::ModularMultiplication` is not available in the version of CryptoPP we are using.
- `CryptoPP::ModularExponentiation`
- `CryptoPP::Integer::Zero`
- `CryptoPP::Integer::One`

You may find the following wiki pages useful during this assignment:

- CryptoPP nbtheory
- CryptoPP Blind Signature

3 Getting Started

To get started, get your stencil repository here and clone it into the `devenv/home` folder. From here you can access the code from both your computer and from the Docker container.

To prevent `crypto_driver.cxx` solutions to earlier assignments being leaked in later assignments, we ask that you copy your code from `crypto_driver` functions implemented in the last assignment into this one. The functions you copy over now will not need to be copied over in the following assignment.

3.1 Running

To build the project, `cd` into the `build` folder and run `cmake ..`. This will generate a set of Makefiles building the whole project. From here, you can run `make` to generate a binary you can run, and you can run `make check` to run any tests you write in the `test` folder.

3.2 Testing

You may write tests in any of the `test/**.cxx` files in the Doctest format. We provide `test/test_provided.cxx`, feel free to write more tests in this file directly. If you want to add any new test *files*, make sure to add the file to the `cmake` variable, `TESTFILES`, on line 7 of `test/CMakeLists.txt` so that `cmake` can pick up on the new files. Examples have been included in the assignment stencil. To build and run the tests, run `make check` in the `build` directory. If you'd like to see if your code can interoperate with our code (which is what it will be tested against), feel free to download our binaries here - we try to keep these up to date, so if you're unsure about the functionality of our binaries, please ask us on Ed!