Lambert-Josh-TermProject

A proof of concept for “Fast Multiparty Threshold ECDSA with Fast Trustless Setup” by Gennaro and Goldfeder.

This work complies with the JMU honor code. I did not give or receive unauthorized help on this assignment.

Executive Summary:

This proof-of-concept implementation focuses on the threshold ECDSA scheme described by Gennaro and Goldfeder in "Fast multiparty threshold ecdsa with fast trustless setup." The code is organized into several files, including *keygen.rs*, *signing.rs*, *pedersen.rs*, *sss.rs*, *verify.rs*, and *main.rs*.

This code does not contain a network layer implementation. Parties are handled as structs within a single execution of the code, with each party struct containing all of the information an individual node on the network would contain. This simulates the multiparty key generation and signing procedure without a full implementation of the network layer.

A quick note on compilation and execution: the compiled file gennaro-rs is provided. This is because the JMU STU systems run the Rust stable toolchain, and building this project requires the nightly toolchain. My user account does not have sufficient rights to change the rust toolchain. To build the project on a standalone linux system (as i’ve done in my homelab), the command ‘rustup override set nightly’ can be run from within the project directory to set the correct toolchain for the features used in this project.

keygen.rs contains the key generation phase, which defines several structs and functions for secure key generation and sharing among parties. The main function, `keygen`, orchestrates the process of generating initial keys, performing Feldman verifiable secret sharing, updating parties with secret shares, reconstructing shared secrets, and computing the group's public key.

signing.rs implements the signature generation phase, providing a secure way for a group of parties to generate a joint ECDSA signature on a message without revealing individual private key shares. The main function, `mta_protocol`, generates public key shares, computes the aggregated public key, and follows a series of steps to generate the joint ECDSA signature.

pedersen.rs implements Pedersen commitments, cryptographic primitives used to commit to a value without revealing it, ensuring privacy and integrity of the involved parties' secrets in threshold ECDSA schemes.

sss.rs implements Shamir's Secret Sharing, a cryptographic method for distributing a secret among multiple parties, where a certain threshold of the parties must collaborate to reconstruct the secret. This ensures that the private key is securely shared among participating parties.

verify.rs implements signature verification, validating that a given signature is produced by the holders of the private key shares corresponding to the aggregated public key. The `is_valid` function takes the aggregated public key, signature, and message as input and returns a boolean value indicating the validity of the signature.

main.rs provides a main function that brings together the previously documented files to demonstrate how to initialize parties, perform key generation, and run the MTA protocol for signing a message.

Code Documentation - *keygen.rs*

Low-Level Design Document for the Key Generation Function

Title: Fast Multiparty Threshold ECDSA with Fast Trustless Setup - Key Generation Function

1. **Overview**

The key generation function is a critical component of the fast multiparty threshold ECDSA protocol with a fast trustless setup. It generates the shared secret and public keys for participating parties by implementing the steps outlined in the Gennaro and Goldfeder paper. This low-level design document provides a detailed explanation of the key generation process, along with the associated data structures and functions.

1. **Key Data Structures**

- `Parameters`: Represents the parameters for the threshold ECDSA protocol, including the threshold, the number of parties, and the Paillier modulus bit length.
- `Party`: Represents a single party in the threshold ECDSA protocol, with fields for the party's index, public key, public key share, secret share, and Paillier encryption and decryption keys.
- `PartyInitialKeys`: Represents the initial keys generated for each party, including the secret key, public key, Paillier key pair, and Pedersen commitment.

- `FeldmanSecretShare`: Represents a secret share within the Feldman VSS scheme, containing the share's index, value, public key share, and Paillier key pair.
- `SharedSecret`: Represents the combined secret shares to form the shared secret.
- `FinalState`: Represents the final state of the key generation process, containing the parameters, parties, shared secret, and aggregated public key.

1. **Key Functions**

- `keygen(params: Parameters, parties: &mut Vec<Party>) -> Result<FinalState, KeygenError>`: The main key generation function that executes the steps outlined in the Gennaro and Goldfeder paper, generating the shared secret and public keys.
- `generate_initial_keys(params: &Parameters, parties: &mut Vec<Party>) -> Result<HashMap<PartyIndex, PartyInitialKeys>, KeygenError>`: Generates the initial keys for each party, including the secret keys, public keys, Paillier key pairs, and Pedersen commitments.
- `combine_shared_secrets(secret_shares: Vec<Scalar>, initial_keys: HashMap<u32, PartyInitialKeys>) -> SharedSecret`: Combines the secret shares to obtain the shared secret.
- `compute_public_key(parties: &[Party]) -> Result<AffinePoint, KeygenError>`: Computes the final aggregated public key.

1. **Key Generation Process**

The key generation function, `keygen`, follows the steps outlined in the Gennaro and Goldfeder paper:

1. Generate the initial keys for each party, including the secret keys, public keys, Paillier key pairs, and Pedersen commitments.
2. Create Feldman VSS schemes for each party and distribute the secret shares.
3. Validate the received shares for each party.
4. Combine the secret shares to obtain the shared secret.
5. Compute the final aggregated public key.
6. **Error Handling**

The key generation function includes error handling for the following scenarios:

- Key generation errors.

- Missing public keys for a party.
- Invalid secret shares.

These errors are represented by the `KeygenError` enum, which provides a description of the error and the associated party or share index.

Summary

This code provides an implementation of the key generation phase described by Gennaro and Goldfeder in "Fast multiparty threshold ecdsa with fast trustless setup." The implementation consists of several structs and functions that define and manipulate the parameters, parties, and cryptographic elements required for secure key generation and sharing. The main function, `keygen`, orchestrates the key generation process by generating initial keys for each party, performing Feldman verifiable secret sharing, updating each party with their corresponding secret share, reconstructing the shared secret, and computing the public key of the group.

Code Documentation - *signing.rs*

The purpose of the function is to generate a signature for a given message using the shared private key generated in the key generation phase of the Gennaro and Goldfeder threshold ecdsa scheme.

Components

1. mta_protocol

```
pub fn mta_protocol(private_key_shares: Vec<Scalar>, message: &str)
```

Description: The main function for generating a threshold ECDSA signature for a given message using the provided private key shares.

Inputs:

- `private_key_shares: Vec<Scalar>`: A vector of private key shares for the participating signers.
- `message: &str`: The message to be signed.

Outputs: None.

Steps:

1. Determine the number of signers `n`.
2. Generate public key shares from the private key shares.
3. Compute the aggregated public key.
4. Generate random nonces `k_i` for each signer.
5. Compute public nonces `R_i` for each signer.
6. Compute the combined public nonce `R`.
7. Compute the challenge `c`.
8. Compute partial signatures `s_i` for each signer.
9. Compute the combined signature `s`.
10. Verify the signature.

2. lagrange_coefficient

```
fn lagrange_coefficient(i: usize, signers: &[usize]) -> Scalar
```

Description: Computes the Lagrange coefficient for the specified signer index `i`.

Inputs:

- `i: usize`: The index of the signer.
- `signers: &[usize]`: A slice of signer indices.

Outputs:

- `Scalar`: The computed Lagrange coefficient.

Signature Generation Algorithm

The algorithm for signature generation follows the Gennaro and Goldfeder algorithm, which consists of the following steps:

1. Generate a random nonce `k_i` for each signer `i`.
2. Compute the public nonce `R_i = k_i * G`, where `G` is the base point of the elliptic curve.
3. Each signer `i` sends their public nonce `R_i` to all other signers.

4. All signers compute the combined public nonce `R = sum(R_i)`.
5. Compute the challenge `c = H(R || m)`, where `H()` is the hash function, `||` denotes concatenation, and `m` is the message to be signed.
6. Each signer `i` computes their partial signature `s_i = k_i + c * x_i`, where `x_i` is their private key share.
7. Each signer `i` sends their partial signature `s_i` to all other signers.
8. All signers compute the combined signature `s = sum(s_i) mod n`, where `n` is the order of the elliptic curve group.
9. The final signature is `(R, s)`.

Assumptions

- The code assumes that the private key shares have been generated and distributed securely during the key generation phase.
- The code assumes that the participating signers are honest and follow the protocol correctly.

Error Handling

- The code does not have explicit error handling as it assumes that the inputs are correct and the signers are honest.
- The code checks whether the signature is valid or not at the end of the `mta_protocol` function.

Code Documentation - *feldman.rs*

In the context of this threshold ECDSA key generation process, the Feldman VSS is used to distribute secret shares among the parties.

The Feldman VSS is used in the following steps of the key generation process:

- Distributing secret shares to each party.
- Validating the received shares for each party.
- Combining the secret shares to obtain the shared secret key.

The key generation process integrates the Feldman VSS implementation as follows:

1. Generate the initial keys for each party, including secret keys, public keys, Paillier key pairs, and Pedersen commitments.
2. Create Feldman VSS schemes for each party and distribute the secret shares.
3. Validate the received shares for each party.
4. Combine the secret shares to obtain the shared secret key.
5. Compute the final aggregated public key.

Code Documentation - *pedersen.rs*

This code implements Pedersen commitments, which are cryptographic primitives used to commit to a value without revealing it. They are used in the context of threshold ECDSA schemes to maintain privacy and integrity of the involved parties' secrets.

The code defines several structs to represent various components of Pedersen commitments:

1. `Commitment`: Represents a commitment to a value.
2. `CommitmentOpening`: Represents the opening of a commitment, which is needed to reveal the committed value.
3. `VerifierPublicKey`: Represents the public key of the verifier.
4. `CommitmentValue`: Represents the value to be committed.
5. `Committer`: A struct representing a party that creates the commitment.
6. `CommitVerifier`: A struct representing a party that verifies the commitment.

The `CommitmentValue` struct has an associated function `from_u64` to create a `CommitmentValue` instance from a u64 value.

The `CommitVerifier` struct has the following associated methods:

1. `init`: Initializes a `CommitVerifier` and its corresponding `VerifierPublicKey`. It generates a random scalar `a` and computes the public key `H` as `H = k256_generator() * a`.
2. `receive_commitment`: Receives a `Commitment` and stores it within the `CommitVerifier` struct.
3. `verify`: Verifies the received commitment using the commitment opening and the committed value. It checks if `C == C2`, where `C` is the received commitment,

and `C2` is computed as `C2 = k256_generator() * r + H * m`, with `r` being the opening and `m` being the committed value.

The `Committer` struct has an associated method `commit` that takes a reference to a `CommitmentValue` and a reference to a `VerifierPublicKey`. The method creates a commitment to the given value and returns the commitment and its opening. It computes the commitment `C` as `C = k256_generator() * r + pub_key_point * val_scalar`, where `r` is a random scalar, and `val_scalar` is the scalar representation of the commitment value.

Code Documentation - *sss.rs*

This code implements Shamir's Secret Sharing; a cryptographic method for distributing a secret among multiple parties, where a certain threshold of the parties must collaborate to reconstruct the secret. It is used in threshold ECDSA schemes to ensure that the private key is securely shared among the participating parties.

The code defines a `SSS` struct that represents a Shamir's Secret Sharing instance, containing the polynomial and the shares generated from the secret.

The `SSS` struct has the following associated methods:

1. `create`: This method takes a reference to a `Scalar` secret, a `threshold` value, and the `num_parties` for which the secret should be shared. The method constructs a polynomial of degree `threshold - 1` with the secret as the constant term and random coefficients for the remaining terms. It then generates shares for each party by evaluating the polynomial at distinct non-zero points. The method returns an instance of the `SSS` struct with the polynomial and the shares.
2. `polynomial`: This method returns a reference to the polynomial used in the secret sharing scheme.
3. `shares`: This method returns a reference to the generated shares for the parties.
4. `eval_polynomial`: This private method takes a reference to a polynomial and a reference to a `Scalar` x-value. It evaluates the polynomial at the given x-value and returns the result. It is used by the `create` method to generate shares for the parties.

Code Documentation - *verify.rs*

This code implements signature verification; a process crucial for validating that a given signature is produced by the holders of the private key shares corresponding to the aggregated public key.

The `is_valid` function takes three arguments: the `aggregated_public_key`, the `signature` as a tuple containing the `AffinePoint` R and the `Scalar` s, and the `message` to be verified. The function returns a boolean indicating whether the signature is valid or not.

The signature verification process consists of the following steps:

1. Parse the input signature (R, s) and the aggregated public key A.
2. Compute the challenge c = H(R || m), where H() is the hash function, || denotes concatenation, and m is the message to be verified.
3. Calculate s * G, where G is the base point of the elliptic curve.
4. Calculate c * A + R, where A is the aggregated public key.
5. Verify that the two values computed in steps 3 and 4 are equal. If they are equal, the signature is valid; otherwise, it is invalid.

The function prints the computed values for s * G and c * A + R, which can be useful for debugging purposes.

In summary, this code provides an implementation of signature verification for the threshold ECDSA scheme described by Gennaro and Goldfeder. The `is_valid` function takes the aggregated public key, signature, and message as input and returns a boolean value indicating the validity of the signature.

Code Documentation - *main.rs*

This code provides a main function that brings together all the previously documented files to provide a proof-of-concept implementation of the threshold ecdsa scheme described by Gennaro and Goldfeder in "Fast multiparty threshold ecdsa with fast trustless setup." The main function demonstrates how to initialize the parties, perform the key generation process, and run the MTA protocol for signing a message.

The main function proceeds as follows:

1. Define the message to be signed as a string: "Hello, world!".
2. Set the parameters for the key generation process, including the threshold value, the number of parties, and the Paillier modulus bit length.

3. Initialize the parties using the `initialize_parties` function. This function takes the number of parties as input and returns a vector of `Party` instances with default values for each of their fields.
4. Run the key generation process by calling the `keygen` function, passing the parameters and the mutable reference to the vector of parties. The keygen function returns the final state, which contains the aggregated private key and the aggregated public key.
5. If the key generation process is successful, print the details of each party, including their public key, public key share, Paillier encryption and decryption keys, and secret share. Additionally, print the aggregated private key and aggregated public key from the final state.
6. If the key generation process fails, print an error message.
7. Extract the private key shares from a subset of parties (in this example, parties 1 and 2).
8. Call the `mta_protocol` function with the extracted private key shares and the message to be signed.

The `initialize_parties` function creates a vector of `Party` instances. Each `Party` is initialized with an index and default (None) values for the public key, public key share, secret share, and Paillier encryption and decryption keys. The function takes the number of parties as input and returns a vector of initialized `Party` instances.

Potential Future Improvements

Here is a list of potential future improvements for the provided code:

- **Optimization:** Optimize the code for better performance, especially for larger numbers of parties or larger key sizes.
- **Error handling:** Improve error handling to provide more informative and specific error messages for different scenarios.
- **Concurrency:** Implement concurrency or parallelism to speed up the key generation, signing, and verification processes.
- **Network communication:** Integrate a network layer to allow parties to communicate securely over the internet or other networks, making the implementation more practical for real-world applications.

- **Testing:** Develop a comprehensive test suite to ensure the correctness and robustness of the implementation.
- **Benchmarks:** Create benchmarks to measure the performance of the code and track improvements over time.
- **Code organization:** Refactor the code into separate modules, making it more modular and easier to maintain and extend.
- **Documentation:** Enhance the documentation, including inline comments and code examples, to make it more accessible for other developers and users.
- **Security audit:** Arrange for an external security audit to identify and fix potential vulnerabilities and weaknesses in the implementation.
- **Adapt to cryptographic library updates:** Update the code to leverage any new features or improvements in the underlying cryptographic libraries used.
- **Support for additional curves:** Extend the implementation to support additional elliptic curves beyond the K256 curve, allowing users to choose their preferred curve for specific use cases.
- **Usability improvements:** Implement a user-friendly command-line interface (CLI) or graphical user interface (GUI) to facilitate interaction with the code and its functionalities.

References:

https://www.ingwb.com/binaries/content/assets/insights/themes/distributed-ledger-technology/ing-releases-multiparty-threshold-signing-library-to-improve-customer-security/threshold-signatures-using-secure-multiparty-computation.pdf

https://eprint.iacr.org/2019/114.pdf

https://www.cs.umd.edu/~gasarch/TOPICS/secretsharing/feldmanVSS.pdf

https://web.mit.edu/6.857/OldStuff/Fall03/ref/Shamir-HowToShareASecret.pdf