Documentation

Author: Cemal Kaygusuz (cemal.kaygusuz@stud.tu-darmstadt.de)

Supervisor: Lucas Schabhüser (lschabhueser@cdc.informatik.tu-darmstadt.de)

Introduction

This short documentation explains the functionality of the commitment scheme library and how to use it. It was built during a practical course in cryptographies and is licensed under the Apache License 2.0.

Requirements

To run the script, you need:

- gcc or any other C Compiler. Only tested with gcc
- Pairing-Based Cryptography (PBC) - https://crypto.stanford.edu/pbc/
- Libsodium - https://github.com/jedisct1/libsodium

A makefile is provided with the library. You can adjust it to your needs if you want to use a different compiler or installed the libraries on a different path.

How to call the functions in commitment (example1.c)

The library is designed to be called in this scheme:

1. Call Setup() to generate the CommitmentKey

Setup
Generates the CommitmentKey
-
@return ***CK** – A CommitmentKey

Setup() doesn't take any parameters other than the security parameter λ and n which are global. It creates a pairing parameter of type F with λ bits and outputs a CommitmentKey.

2. Call KeyGen() to generate a KeyPair

KeyGen
Generates a KeyPair
@param ***CK** – A CommitmentKey
@return ***KP** – A KeyPair

KeyGen() takes the CommitmentKey generated in Setup() and outputs a KeyPair containing PublicKey and SecretKey.

3. Generate n random Z_R ring elements r

4. Generate n random messages m, whereas a message is a vector of size T containing Z_R ring elements

$m = (m_1, m_2, \dots, m_T)^T$
with $m_1, \dots, m_T$ being random Z_R ring elements

5. Generate an arbitrary long dataset δ

6. Call PrivateCommit() n times with identifiers = 1…n to generate n Authenticators

PrivateCommit
Generates an Authenticator
@param **sk** – A SecretKey @param **m** – A message @param **r** – A randomly selected Z_R ring element @param ***dataset** – An arbitrary long dataset @param **datasetBytes** – The amount of bytes dataset holds @param ***identifier** – An identifier with o <= identifier <= n @param ***CK** – A CommitmentKey
@return ***KP** - A KeyPair

PrivateCommit() takes the SecretKey from the KeyPair we got from KeyGen(), a message m and randomness r at the subindex identifier which he also pass as parameter, the dataset, datasetBytes and the CommitmentKey.

For example, when we call PrivateCommit() for the first time, we pass the function the identifier o and therefore m[o] and r[o].

7. Generate a LinearFunction with random values in it

8. Call FunctionCommit() to generate a FunctionCommitment

FunctionCommit
Generates a FunctionCommitment
@param **pk** – A PublicKey @param **lf** – A LinearFunction @param ***CK** – A CommitmentKey
@return ***ret** – A FunctionCommitment

FunctionCommit() is straight-forward. It calculates $\prod (h2_i\ f_i)$, i=o to n and returns the product as FunctionCommitment.

9. Call Eval() to generate a final Authenticator

Eval
Generates a final Authenticator
@param **lf** – A LinearFunction @param **myAuthenticators[n]** – n Authenticators @param ***CK** – A CommitmentKey
@return ***auth** – A final Authenticator

Eval() combines the n Authenticators into a final one and returns it.

10. Generate a final message: $\sum (f_i\ m_i)$ **,** i=o to n

11. Generate a final element r: $\sum (f_i\ r_i)$ **,** i=o to n

12. Call PublicCommit() to generate a Commitment

PublicCommit
Generates a Commitment
@param ***CK** – A CommitmentKey @param **m[T]** – A message m @param **r** – A randomly generated Z_R ring element
@return ***ret** – A Commitment

PublicCommit() computes the Commitment out of the final elements m and r from step **10** and **11** and the CommitmentKey

13. Call FunctionVerify(). It should output 1

FunctionVerify
Returns 1 or 0 depending on whether the conditions hold
@param **pk** – A PublicKey @param **auth** – The final Authenticator @param **c** – A Commitment @param **f** – A FunctionCommitment @param ***dataset** – An arbitrary long dataset @param **datasetBytes** – The amount of bytes dataset holds @param ***CK** – A CommitmentKey
@return **1** if signature check returns true and $e(V,Z) = e(U,g_2) * f * e(C,Y)$ holds, **0** otherwise

The final step is to call FunctionVerify() with the final Authenticator from Eval(), the Commitment from PublicCommit() and the FunctionCommitment from FunctionCommit(). It first checks whether the signature is correct. If that isn't the case, the function returns 0. Otherwise it continues to check the pairings. If everything works as intended, it should return 1

Calling Share/Reshare/Reconstruct (example2.c)

1. Set pairing parameters as well as N and t

2. Generate a random message m being a single Z_R ring element (not a vector in this case)

3. Call Share(m) to generate secret shares $s_1, \ldots, s_N$

Share
Generates secret shares from a message
@param ***m** – A message @param ***CK** – A CommitmentKey
@return **sshares[N]**– The secret shares

4. Call Reshare(s_1, …, s_N) to generate new secret shares s'_1, …, s'_N

Reshare
Generates new secret shares out of existing ones
@param **sshares[N]** – The secret shares @param ***CK** – A CommitmentKey
@return **sshares2[N]**– The new secret shares

5. Choose a random subset G from {1,…,N} of size t

6. Call Reconstruct(s'_1, …, s'_N, G) to generate a message m'

Reconstruct
Generates a new message from secret shares and a subset
@param **subset[t]** – A subset of {1,…,N} of size t @param **sshares[N]** – The secret shares @param ***CK** – A CommitmentKey
@return ***ret** – A message

7. If everything worked, m = m' should hold now

Calling VKeyGen/Probgen/Compute/Verify (example3.c)

1. Set pairing parameters as well as n, T, N and t

2. Generate a random LinearFunction

3. Call VKeyGen to generate a KeyTriple (a struct holding SecretKey, VerificationKey and EvaluationKey)

VKeyGen
Generates SecretKey, VerificationKey and EvaluationKey
@param **lf** – A LinearFunction @param ***kp** – A KeyPair @param ***CK** – A CommitmentKey
@return ***kt** – A KeyTriple

4. Call ProbGen with n different inputs for m and r just like in the steps **3** – **6** in the commitment library

ProbGen
Generates an AuthAndShare object
@param **sk** – A SecretKey @param **m[T]** – A message being a vector of size T @param ***r** – A Z_R ring element @param ***dataset** – An arbitrary long dataset @param **datasetBytes** – The amount of bytes dataset holds @param ***identifier** – An identifier @param ***CK** – A CommitmentKey
@return ***as** – An AuthAndShare object

5. Call Compute with the results of step **4** to generate a single AuthAndShare struct

Compute
Generates an AuthAndShare object out of n existing ones
@param ***ek** – An EvaluationKey @param **as[n]** – n AuthAndShare objects @param ***CK** – A CommitmentKey
@return ***asReturn** – An AuthAndShare object

6. Call Verify with the VerificationKey, Dataset and the result of step **5**. If everything worked, Verify should return 1

Verify
Checks if FunctionVerify holds
@param ***vk** – A VerificationKey @param ***as** – An AuthAndShare object @param **dataset** – An arbitrary long dataset @param **datasetBytes** – The amount of bytes dataset holds @param ***CK** – A CommitmentKey
@return **1** if FunctionVerify return 1, **0** otherwise