ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Improvements on Distributed Key Generation cryptography

Kopiga Rasiah

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

June 8, 2018

Responsible	**Supervisor**
Prof. Bryan Ford	Nicolas Gailly
EPFL / DEDIS	EPFL / DEDIS

Contents

1 Introduction **1**
1.1 Distributed key generation 1

2 Background **3**
2.1 Shamir secret sharing 3
2.2 (Feldman's) Verifiable secret sharing 3
2.3 (Pedersen's) Distributed Key Generation 4
2.4 Conclusion 5

3 Problematic **7**
3.1 Method on `VSS` process 7
3.2 Method on `DKG` protocol 8
3.3 Conclusion 9

4 Implementation **10**
4.1 Testing on `VSS` 10
4.2 Implementation on the `DKG` 10
4.3 Testing 13
4.4 Conclusion 15

5 Conclusion **16**
5.1 Future work 16

6 Bibliography **18**

1 Introduction

One of the most important aspect of cryptography is secrecy. A large number of cryptographic applications requires a trusted authority to hold a secret. However, with the evolving attacks on the Internet, it is difficult to maintain such security that is entrusted to a single party. Secret sharing protocols is one of the solution that overcomes this problem. It involves a dealer who chooses a secret that he divides and share among the multiple servers, inducing the decentralization of the secret. Nevertheless, it still requires to trust a third party. A distributed key generation scheme settles this constraint by allocating the secret to not one dealer, but multiple servers. By doing so, a malicious attacker will need to break into multiples locations which slows down his intrusion process. However, this protection is incomplete for the entire life-time of the secret, as break-ins into subsets of servers are not completely excluded. In this project, I make contribution to an existing implementation of `DKG` protocol where I will propose a proactive secret sharing scheme that consists of updating the shares. By refreshing the shares, the information that the attacker has stolen previously becomes obsolete. Throughout this project report, I will explain comprehensively the `DKG` protocol and how I gradually incorporate the proactive secret sharing into that protocol. I will terminate by enumerating possible future improvements.
The implementation on which I contributed was developed by the Decentralized and Distributed Systems laboratory team. It is part of the library Kyber which provides a toolbox of advanced cryptographic primitives.

1.1 Distributed key generation

Distributed key generation is a cryptographic protocol that does not rely on any third party nor dealer. It allows a set of n nodes to jointly contribute to the generation of a shared private and public key set, and only trusts a specified number of servers such that any quantity of servers greater than the threshold is able to reveal the secret, but a smaller subset cannot. The nodes create and distribute among themselves shares that once combined, they compose the private key. The particularity is that the private key does not need to be reconstructed to allow performing signatures or encryptions, every operations can be conducted distributively. On the other hand, the public key can be revealed and used in the clear. In order to have access to the private key, one has to retrieves at least t shares so that he can reconstruct it. Consequently it is required to have a threshold of t honest nodes, in order to construct a successful key pair.
The `DKG` protocol is mainly used for decryption of shared cipher texts or group digital signatures: the public key can be used by anyone to encrypt a secret, e.g a password, which necessitate a threshold of group member to decrypt it. The private key on the other hand is useful to digitally sign to

a document. It requires again a least t agents for a group to sign and the private key is never constructed in any location.

2 Background

2.1 Shamir secret sharing

Shamirs scheme is an algorithm in cryptography that allows you to divide a secret value into multiple unique parts, called shares, by evaluating a polynomial at certain indices. The secret z is set to be the first coefficient in the t-degree polynomial function, i.e $f(0) = z$. By having at least t shares, one can recover the polynomial function using Lagrange interpolation and thus retrieve the secret.

2.2 (Feldman's) Verifiable secret sharing

A secret sharing protocol is said to be verifiable if the participants have the ability to verify whether their shares are legitimate and that the dealer is honest. It also allows to expose nodes that are dishonest. In case of this situation, the remaining honest nodes can discard malicious participants (including the dealer) and carry on the protocol as the qualified group. Feldman's `VSS` protocol offers this scheme.
Based on Shamirs secret sharing scheme, Feldman's `VSS` protocol works so: a dealer - distinguished participant that lead the distribution - first generates a random t degree polynomial function. The polynomial function must be such that the coefficients are randomly uniformly chosen over a cyclic group G of prime order p. G must be a group where the discrete logarithm problem is supposed to be hard. That is, given g and s, it is difficult to retrieve a such that $a \times g = s$, for any a,$g \in G$. The dealer also constructs the public polynomial

$$F(x) = f(x) * g = g \times z + g \times a_1 x + g \times a_2 x^2 + ... + g \times a_{t-1} x^{t-1}$$

where the coefficients are called the commitments. The public key will be Z $= z \times g$, which is output in the clear. Since $z, g \in G$, it is difficult to retrieve z from Z. The dealer then proceeds with the sharing process: for each node P_i, he evaluates his polynomial on i, and the result $s_i = f(i)$ will be sent privately to the node i. He also broadcast the commitments to all the nodes.
For the verification process, each node i check the following equality:

$$g * s_i == F(i) == g \times z + g \times a_1 i + g \times a_2 i^2 + ... + g \times a_{t-1} i^{t-1}$$

If any of the nodes fails to get the equality, it sends a complaint to the dealer, who then reveals the share given to i, so that the other participants can verify the equality, leading to disqualify the liar.
If at least t participants reveal their shares, the polynomial function can be retrieved using Lagrange interpolation, and the secret value is recovered by evaluating the polynomial at 0. Hence why it is required to have a threshold of t honest parties.

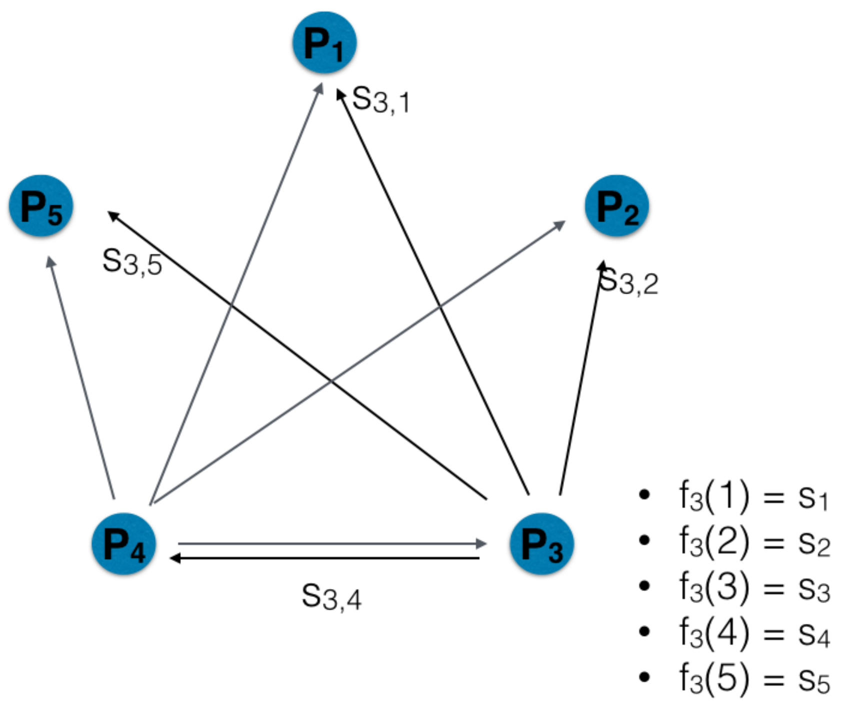


Figure 1: A DKG process, only the distribution of node 3 is shown. All the nodes do the same process.

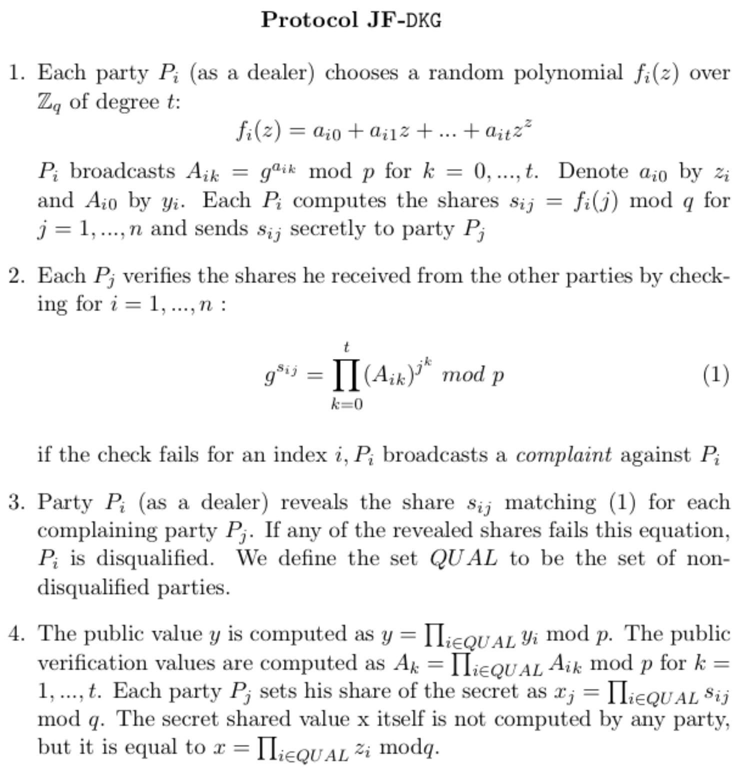
2.3 (Pedersen's) Distributed Key Generation

As stated before, the Feldman's `VSS` process has the property to have a single dealer who knows the secret value. It is therefore easy to break in a single point. To overcome this constraint, Pedersen introduce the first `DKG` protocol, based on Feldmans `VSS` process. In this protocol, every node runs simultaneously a `VSS` instance as a dealer. Each party i generates random secret z_i and constructs a polynomial function $f_i(x)$, such that $f(0) = z_i$, then distributes the shares according to the `VSS` protocol and broadcast the commitments as well (see Figure 1).

At the end, the secret value s is taken to be the sum of the properly shared z_i's:

$$s = \sum_{i=1}^{n} f_i(0)$$

and the final share of a node j corresponding to the secret will be the sum


Protocol JF-DKG

1. Each party P_i (as a dealer) chooses a random polynomial $f_i(z)$ over $\mathbb{Z}_q$ of degree t:
   $$f_i(z) = a_{i0} + a_{i1}z + ... + a_{it}z^z$$
   P_i broadcasts $A_{ik} = g^{a_{ik}} \bmod p$ for $k = 0, ..., t$. Denote a_{i0} by z_i and A_{i0} by y_i. Each P_i computes the shares $s_{ij} = f_i(j) \bmod q$ for $j = 1, ..., n$ and sends s_{ij} secretly to party P_j
2. Each P_j verifies the shares he received from the other parties by checking for $i = 1, ..., n$:
   $$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} \; mod \; p \tag{1}$$
   if the check fails for an index i, P_i broadcasts a *complaint* against P_i
3. Party P_i (as a dealer) reveals the share s_{ij} matching (1) for each complaining party P_j. If any of the revealed shares fails this equation, P_i is disqualified. We define the set $QUAL$ to be the set of non-disqualified parties.
4. The public value y is computed as $y = \prod_{i \in QUAL} y_i \bmod p$. The public verification values are computed as $A_k = \prod_{i \in QUAL} A_{ik} \bmod p$ for $k = 1, ..., t$. Each party P_j sets his share of the secret as $x_j = \prod_{i \in QUAL} s_{ij} \bmod q$. The secret shared value x itself is not computed by any party, but it is equal to $x = \prod_{i \in QUAL} z_i \bmod q$.

Figure 2: The DKG protocol - R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin

of the shares he received, i.e

$$s_j = \sum_{i=1}^{n} f_i(j)$$

The commitments are computed as the sum of all polynomial $F(x) = \sum_{i=1}^{n} F_i(x)$, and the public key will be $F(0) = s \times g$

By contrast to VSS protocol, the secret is henceforth constructed nowhere, is known to no one unless one manages to compromises t final shares.

2.4 Conclusion

Shamir's secret sharing algorithm allows to share a secret in multiple locations which forces the adversary to increase his break-ins. Extended by Shamir's algorithm, Feldman's verfiable secret sharing enables the node and the dealer to verify their shares, so that in case of any presence of adversary, the latter will immediately be discarded. Based on Feldman's VSS protocol, the distributed key generation protocol's advantage is that the protocol does not need to have any trusted party, which avoids the single point failure. The next section will introduce the proactive secret sharing method, which will increase the security by a notch.

3 Problematic

We have seen that the `DKG` protocol with uniform output distribution guarantees security, as it forces the adversary to attack multiples servers. However that security is assured only on short-term: an attacker has the secret's life time to break it. It is therefore a bad cryptographic practice to keep long term keys without renewing them. A technique called Proactive secret sharing overcomes this hurdle: it enables to issue new shares without changing the underlying distributed secret. In other words it consists of updating the shares periodically so that the attackers do not have the time to steal the shares. Indeed, we know that the shares are the points of a polynomial function evaluated at certain indices. Knowing t shares, i.e points, allows to recover the t-degree polynomial function. However, by renewing the shares, i.e the points, it changes the underlying polynomial. In consequence, if the attacker still possesses less than t obsolete shares, the latter will not contribute to the recovery of the new polynomial as they do not correspond to it at all. In short, refreshing shares periodically changes the polynomial - periodically, and if the malicious adversary could not steal at least t shares during the same period of time, he will not be able to retrieve the polynomial - and thus the secret value. The scheme has the advantage to reduce the time available for the adversary to break t shares. In this project, we are going to use this technique to update the shares. I will however not use periodicity, which I leave for future work. As stated in the introduction, this technique aims to improve the security of the protocol.

3.1 Method on `VSS` process

In order to progress gradually, I first worked on a simple `VSS` instance. We assume that the first round of `VSS` protocol is already done, that is the nodes have already received their shares. Now the dealer needs to refresh the shares of the fellow nodes, without affecting the secret value. In order to do so, he should regenerate shares that once combined with the previous one, "interpolates" to a new polynomial whose first coefficient is the same secret, because we do not want the secret value to change. Consequently, the dealer needs to create a new random polynomial whose first coefficient is 0. He then produces "intermediate" shares from it and transmits them during the second round of `DKG` so that the nodes can add them to their initial shares. The reason why we use an extra random polynomial with coefficient 0 instead of the same coefficient is that a dealer does not have to store the first coefficient anywhere z_i. If he happens to lose it he can still refresh the shares.

	initial share at node i		$f(i) = z_i + a_1 i + ... + a_{t-1} i^{t-1}$
$+$	share to be add	$+$	$g(i) = 0 + b_1 i + ... + b_{t-1} i^{t-1}$
	refreshed share		$h(i) = z_i + c_1 i + ... + c_{t-1} i^{t-1}$

where $c_i = a_i + b_i$. The secret remained unchanged.

Here we clearly see that the polynomial $h(x)$ is different from $f(x)$. Since the polynomial has changed, the initial shares that the attacker has stolen become useless; unless he has already stolen t shares from the same polynomial at the same period of time.

If the shares are updated, so must be the commitments (while the public key remains unchanged). The computations goes along the same way as for the shares: the dealer computes the commitments of the newly generated polynomial and broadcasts them to the nodes:

$$\begin{array}{rr} & F(i) = z_i \times g + a_1 i \times g + ... + a_{t-1} i^{t-1} \times g \\ + & G(i) = 0 \times g + b_1 i \times g + ... + b_{t-1} i^{t-1} \times g \\ \hline & H(i) = z_i \times g + c_1 i \times g + ... + c_{t-1} i^{t-1} \times g \end{array}$$

where $c_i = a_i + b_i$. The public key remained unchanged.

Now the node has a new refreshed set of shares $h(i)$ and commits $F()$.

3.2 Method on DKG protocol

My next step consisted on developing the exact same scheme, but on the DKG protocol. Although the illustration in section 3.1 refers to a VSS instance, we know that DKG protocol executes n VSS protocols in parallel. Consequently, the proactive secret sharing scheme for a DKG protocol is exactly the same, it only extends for multiple dealers.
Let's assume that the initial DKG round has already been done. Now each participant i generates a new random polynomial $g_i(x)$ with secret 0, and makes a distribution of shares evaluated with $g_i(x)$ among the nodes. As described in section 2.1.3 the initial share of the node j is $s_j = \sum_{i=1}^{n} f_i(j)$. Remember that the secret value is

$$s = \sum_{i=1}^{n} f_i(0)$$

which should remain unchanged. The intermediate share is

$$g_j = \sum_{i=1}^{n} g_i(j)$$

Consequently, the updated share will be:

$$s_j + g_j = \sum_{i=1}^{n} (f_i + g_i)(j)$$

. The secret value remains the same:

$$s + \sum_{i=1}^{n} g_i(0) = s + 0 = s$$

3.3 Conclusion

An attacker is still able to break multiple servers if he can seize more that $t - 1$ shares. He does the latter by the means of a mathematical method, namely the Lagrange interpolation. However by updating the shares - which induces changing the underlying polynomial - the attacker's old information becomes obsolete. The next section will demonstrate the implementation of the scheme I have described above.

4 Implementation

4.1 Testing on VSS

I tested the scheme in section 3.1 on a VSS instance in a different way. Specifically, instead of adding shares from different polynomial together, the code I implemented adds a random polynomial with secret 0 to the initial polynomial, and checks whether we can recover the same secret and whether the new commits tallies with the updated shares.

```
n := 10
t := n/2 + 1
polyf := NewPriPoly(Pick(RandomStream())
polyh := NewPriPoly(nil)  //Intermediate polynomial
polyg := poly.Add(polyh) // Updated polynomial

sharesOfNewPoly := polyg.Shares(n)
recovered := RecoverSecret(g, sharesNewPoly, t, n)

if !recovered.Equal(polyg.Secret()) { test.Fatal("recovered secret
does not match initial value")
}
```

You can find more tests and details on VSS scheme implementation in the annexed code:
https://github.com/rkopiga/BachelorProject/blob/master/BachelorProject/dedis/kyber/share/poly_test.go#L13
https://github.com/rkopiga/BachelorProject/blob/master/BachelorProject/dedis/kyber/share/poly_test.go#L39
https://github.com/rkopiga/BachelorProject/blob/master/BachelorProject/dedis/kyber/share/poly_test.go#L70

Now comes the main part of my project where I implement and test the scheme developed in section 3.2 in the dkg package in kyber library. I mainly add code in share/dkg/dkg.go and share/dkg/dkg_test.go.

4.2 Implementation on the DKG

The code already has a function `func NewDistKeyGenerator` which is the main component of the protocol and corresponds to one dealer. It is the main function that corresponds roughly to a VSS instance with a dealer. The DistKeyGenerator contains the following fields:

```
DistKeyGenerator{
dealer,
verifiers, //set of nodes that are qualified to send/receive nodes
threshold,
random,
long:         longterm,  //Diffie-Hellman private key
pub:          pub,      //Diffie-Hellman public key
// not to be confused with the distributed private/public key.
participants: participants,
index:        index    // index of the dealer
}
```

In vss.go, the dealer generate himself a random polynomial function with a random secret value (i.e the first coefficient of the polynomial, not the private key) and stores it in an array at the outset. The polynomial is in consequence not available in the distributed key generator. Therefore, I first need to implement a function which is very similar to the function DistKeyGeneration, but takes a parameter "secret", so that while creating the dealer, it can give the secret as an argument to generate the polynomial:

```
func initDistKeyGenerator(suite Suite, longterm kyber.Scalar,
participants []kyber.Point, t int, secret kyber.Scalar){
    ownSec := secret
    dealer := vss.NewDealer(ownSec, //other necessary parameters)
    newDistKeyGenerator.dealer = dealer
    return newDistKeyGenerator
}
```

Then I use two auxiliary function, one is NewDistKeyGenerator which creates a random secret coefficient and calls initDistKeyGenerator :

```
func NewDistKeyGenerator(suite Suite, longterm kyber.Scalar,
participants []kyber.Point, t int) (*DistKeyGenerator, error) {
ownSecret := suite.Scalar().Pick(suite.RandomStream())
return initDistKeyGenerator(suite, longterm, participants, t, ownSecret)
}
```

It returns a DistKeyGenerator for the initial round of the DKG.

The other one is NewDistKeyGeneratorWithoutSecret which secret coefficient is 0 and calls initDistKeyGenerator as well:

```
func NewDistKeyGeneratorWithoutSecret(suite Suite, longterm kyber.Scalar,
participants []kyber.Point, t int) (*DistKeyGenerator, error) {
ownSecret := suite.Scalar().Zero()
return initDistKeyGenerator(suite, longterm, participants, t, ownSecret)
}
```

This function is used for the second round of the DKG, that is distributing the intermediate shares so that the nodes can refresh their shares.

Now that the participants have received their shares, they need to add them to their initial shares. DKG package provides the field DistKeyShare with contains the index and the final share of a node. Hence, we need to create another function that adds two DistKeyShares together, in order to get an updated share:

To add two shares, I implemented a new method Renew, that is called on a instance d of type DistKeyShare, and have the intermediate share g as the parameter. The type of the share is a scalar. It takes the share d and add to the share g giving a new scalar-type share. Before adding to shares, the code checks first whether the intermediate share is legitimate, i.e the "intermediate" public key is indeed 0. A way to check this is to verify the equivalence whether G(0) == 0 * G (generator of a group) which must be necessarily 0. If it is not the case, it means the first coefficient is not 0 which will destroy the private and the public key. This is possibly an act of a malicious adversary trying to corrupt the shares.

```
//Check G(0) = 0*G.
if !g.Public().Equal(Mul(suite.Scalar().Zero(), G)) {
return nil, errors.New("wrong renewal function")
}
```

Then I check whether we are adding together the shares of the same node as to avoid adding the intermediate share of a node to another one. If it does so, it means there is an issue with the implementation and it disrupts the whole DKG protocol: since the same method is utilized for the initial distribution of the share, it means that a node received someone else's share (but a correct implementation would have already returned an error).

```
//Check whether they have the same index
if d.Share.I != g.Share.I {
return nil, errors.New("not the same party")
}
```

Then I finally sum the shares together, and also updates the commitments with a component-wise addition, like explained in section 3.2.

```
newShare := suite.Scalar().Add(d.Share.V, g.Share.V)
newCommits := make([]kyber.Point, len(d.Commits))
for i := range newCommits {
newCommits[i] = suite.Point().Add(d.Commits[i], g.Commits[i])
}
```

You can find the complete implementation of this code in the annexed code: https://github.com/dedis/kyber/blob/master/share/dkg/pedersen/dkg.go

4.3 Testing

Now that I have implemented the main code, I make some tests to see whether it works correctly. The first test is to verify whether we can recover the **same** secret after updating the shares. This test checks that the refreshing process does not destroy the private key of a client. `fullExchange()` executes a whole `DKG` round, where n nodes run `VSS` instances in parallel. At the end of it, each instance, along with its attributes (dealer's index, distributed deals...), will be stored in the array dkgs[]. Similarly `fullExchangeWithRenewal()` executes the refreshing round of DKG, where n nodes distributes the intermediates shares. Those instances with their attributes will be stored in another array called `dkgsReNew[]`.

```
fullExchange()
fullExchangeWithRenewal()
```

If the exchanges have been correctly executed, the integrity of the shares must be conserved. Therefore, when the code fetch the share of each instance, it checks, for each instance of the `DKG`, ,that the share it is not nil. The shares are stored in the order of the index, which means that `dkg[i]` and `dkgsRenew[i]` correspond to instances of node i. `dks` is the initial distributed key-share of a node and `dksNew` is the intermediate distributed key share of the same node. The code then sum the two shares in `disKeyShares[i] = dks.Renew(dksNew)`. The code finally checks if the index of the initial node is the same as the index of the intermediate share as to avoid mixing up the shares.

```
distKeyShares := DistKeyShare[]
for i range (dkgs.size()){
dks, err := dkg[i].DistKeyShare() //Initial share
require.Nil(err)
require.NotNil(dks)
dksNew := dkgsReNew[i].DistKeyShare() //Intermediate share
disKeyShares[i], _ = dks.Renew( dksNew)  //Updating
assert.Equal(t, dkg.index, uint32(dks.Share.I))
}
```

The code now checks that the commitments are the same for each updated share by doing `assert.True(t, checkDks(dks, dkss[0]))`. It has to be the same, since the commitments are the coefficients of the sum of each public polynomial. If it is not, it would disrupt the secret recovery process since the underlying polynomial is not the same for every node. Then using the shares, the code recovers the secret with the function `RecoverSecret(shares)` and checks that it is not nil. It should not be nil since the right parameters were given from the beginning.

```
//The commitments of every DistKeyShare must be the same
shares := shares[]
for i, dks := range dkss {
assert.True(t, checkDks(dks, dkss[0])) //commitments must
                                       //be same for every nodes.
shares[i] = dks.Share
}
secret, err := share.RecoverSecret(shares)
assert.Nil(t, err)
```

Finally, in order to check whether the refreshing process did not affect the private key, we take it's public key, i.e we multiply the secret with the generator $g \in G$: `commitSecret := Mul(secret, g)` and we compare it with the public key of a initial share (`formerDks := dkgs[0].DistKeyShare()`), which must be true if the above steps are executed accordingly.

```
//Check whether we have indeed the same secret before updating:
//secret*G must equal F(0)
formerDks := dkgs[0].DistKeyShare()
commitSecret := Mul(secret, g)
assert.Equal(t, formerDks.Public().String(), commitSecret.String())
```

I also implemented another test that verify if the conditions in `Renew` are verified, that is, if given invalid parameters it should return an error. This test execute again `fullExchange()` and `fullExchangeWithRenewal()`. The test first checks that two VSS instances of different index cannot be added. For that, it simply takes initial share of node 2 (`dks := dkgs[2].DistKeyShare()`) and the intermediate share of node 1 `dks1 := dkgsReNew[1].DistKeyShare()`) and sum them: `newDsk1, err := dks.Renew(dkg.suite, dks1)`. This should return an error and the returned must be nil.

```
dks := dkgs[2].DistKeyShare()
//Check when they don't have the same index
dks1 := dkgsReNew[1].DistKeyShare()
newDsk1, err := dks.Renew(dkg.suite, dks1)
assert.Nil(t, newDsk1)
assert.Error(t, err)
```

Next, the code checks that it cannot add an intermediate share that has the first coefficient different from 0 (the secret is not equal to 0). For that, the code simply adds two shares from the same `DKG` round, since while those shares were generated, their polynomial did not have the first coefficient equaling to 0. It again should return an (nil,error) pair for the share and error variables.:

```
//Check the last coeff is not 0 in g(x)
```

```
dks3, _ := dkgs[1].DistKeyShare()
newDsk3, err3 := dks.Renew(dkg.suite, dks3)
assert.Nil(t, newDsk3)
assert.Error(t, err3)
```

Finally, the test checks whether the code works, i.e adding an initial share and the intermediate share both corresponding to the same node should return an updated share, and no error :

```
//Finally, check whether it works
dks2 := dkgsReNew[2].DistKeyShare()
newDks, err := dks.Renew(dkg.suite, dks2)
assert.Nil(t, err)
assert.NotNil(t, newDks)
}
```

You can find the complete implementation of this code in the annexed code: `https://github.com/dedis/kyber/blob/master/share/dkg/pedersen/dkg_test.go`

4.4 Conclusion

Implementing the code did not differ drastically from the mathematical point of view: it is a simple application of the mathematical scheme. Several appropriate condition tests such as checking the indices and the commitments before renewing shares will avoid failure. Although my implementation of the renewal process for `VSS` protocol is correct, it should have consisted of code that adds two shares and not two polynomials, as it will tally with my scheme in section 3.1.

5 Conclusion

The goal of the project is to improve the security of a protocol that is under ongoing development, namely by incorporating the proactive secret sharing scheme into `DKG` protocol. First I presented a share renewal scheme for the `VSS` protocol, which facilitated the elaboration of such scheme for a `DKG` protocol and allowed to implement it in the Kyber library. By updating the shares, we can see that the chances of an attack by intrusion is lowered, and is even significantly reduced if the updates are executed periodically. However my implementation execute the renewal phase *by hand*, whereas in real-life protocol this should be automated for a period of time. This is one of the important feature in proactive secret sharing that needs to be implemented, but this is beyond the scope of my project.
As for the code that I implemented, it has successfully merged into the kyber library.

5.1 Future work

Periodical renewal. As for the future improvements regarding the core of my project, periodical updates must be implemented: indeed, in this project, the renewal of the shares are done manually: that is, a function such as `fullExchangeWithRenewal` needs to be called externally, in order to updates the shares. In real life, it is advised to have a function that automate this process. In broad outline the life time of a secret is split into **periods** of time (could be a week, a month, or a year), and in the beginning of the period time the refreshing process is executed. The shorter the period is, the more difficult the break-ins of t shares becomes. This method establish a better security to the `DKG` protocol.
Share recovery Also, the protocol should be able to retrieve lost or corrupt shares: over the life time of a secret an attacker can destroy the shares. If he manages to destroy up to $n - t + 1$ shares (in a (n,t)`DKG` protocol), the secret will entirely be destroyed. Therefore, restoring corrupt shares is necessary. For this, the corrupted share needs to be detected. This is easy if in the updating phase since the commitments of the destroyed share will differ from the others.
Other improvements on `DKG` protocol We can extend the implementation of the `DKG` by adding and/or removing the nodes. This group modification protocols allows the to have a control on the nodes. Removing nodes is utilized when certain participants should no longer be able to reconstruct the secret. Adding allows to increase redundancy
Real-world application Moreover the renewal protocol implemented in Kyber library can be used in applications that are under development. I am currently working on the library Drand, a distributed randomness beacon daemon, written in Golang. It involves a set of nodes that produce

collective, publicly verifiable, random values at fixed intervals using pairing-based threshold cryptography. The nodes need to generates private/public distributed key in order to sign message related to the generation of randomness.

6 Bibliography

References

[1] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk and Tal Rabin. *Secure Distributed Key Generation for Discrete-Log Based Cryptosystems.* 19 May 2006

[2] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk and Moti Yung *Proactive Secret Sharing or: How to Cope With Perpetual Leakage* Watson Research Center, Yorktown Heights.

[3] Aniket Pundlik Kate *Distributed Key Generation and Its Application* Ontario, Canada, 2010

[4] Aniket Kate *Reseach Summary: Distributed Cryptography*

[5] Distributed key generation - Wikipedia `https://en.wikipedia.org/wiki/Distributed_key_generation`

[6] Shamir's secret sharing - Wikipedia `https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing`