# PVSS Documentation

**Jörn Heissler**

**Feb 16, 2020**

# CONTENTS:

This project is a python (>= 3.7) implementation (library and CLI) of Publicly Verifiable Secret Splitting (PVSS).

PVSS is a non-interactive cryptographic protocol between multiple participants for splitting a random secret into multiple shares and distributing them amongst a group of users. An arbitrary subset of those users (e.g. any 3 out of 5) can later cooperate to reassemble the secret.

The common use case for secret splitting is to create a highly durable backup of highly sensitive data such as cryptographic keys.

All communication between the participants is public and everyone can verify that all messages have been correctly created according to the protocol. This verification is done through non-interactive zero-knowledge proofs.

The math is based upon the paper Non-Interactive and Information-Theoretic Secure Publicly Verifiable Secret Sharing by *Chunming Tang* et al. who extended *Berry Schoenmaker*'s paper A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting which in turn is based on Shamir's Secret Sharing.

One notable difference to prior work is the addition of a receiver user: In their scheme the secret is made public while it is being reassembled, which violates the goal to keep the secret secret. To address this issue, the users no longer disclose their share of the secret but use ElGamal encryption to securely convey the share to a separate receiver user who will then reassemble the secret. Like all other communication, the encrypted share is public and it can be verified that the users followed the protocol.

CHAPTER

ONE

# INSTALLATION

PVSS's dependencies are:

- python (>= 3.7)
- **At least one of:**
  - libsodium (>= 1.0.18, recommended, for Ristretto255 group)

    On Debian (Bullseye / 11 and later) or Ubuntu (Eoan / 19.10 and later):

    ```
    # apt install libsodium23
    ```

  - gmpy2 (Group of quadratic residues modulo a large safe prime)

You can install PVSS with pip:

```
$ pip install pvss
```

And optionally:

```
$ pip install gmpy2
```

CHAPTER

TWO

# USE CASES

The generic use case of PVSS is to create a secure and durable backup of some highly valuable information.

## 2.1 Offline backup for cryptographic keys

Many applications utilize public-key cryptography and require a private key for their operation. Examples are Certificate Authorities, SSH clients, email users and web servers.

If a private key is disclosed, a lot of damage can be done, e.g. issuing false certificates, signing into SSH servers, faking email signatures or impersonating a web application to intercept data. That means that private keys must be kept private. One approach is to store the key on some Hardware Security Module which will carry out the cryptographic operations but won't allow to create a copy of the key.

On the other hand, private keys must stay available. Through hardware defects or human mistakes a private key can be easily destroyed, meaning one can no longer issue new certificates, logon to a SSH server, sign or decrypt emails or operate the web server.

For web servers, there is a trivial solution: If the private key is disclosed, revoke its certificate. If the key is destroyed, simply create a new private key and issue a new certificate.

For the other use cases, there is no easy solution. But the next best thing is PVSS:

When generating a new private key, PVSS is used to create a random secret. The private key is encrypted symmetrically with this secret, e.g. with AES-GCM. The random secret is split among $n$ semi-trusted users. It is defined that any $1tn$ of those users can cooperate to reassemble the secret.

Once access to the private key is needed, a special receiver user is created. $t$ of the users need to re-encrypt their key shares with the receiver's public key. Only the receiver can then reassemble and decrypt the private key. The key could be stored directly into some HSM and then wiped from the receiver's memory.

## 2.2 Backup of arbitrary data

Similarly, arbitrary data can be backupped securely. For each new backup job, PVSS is used to create and split a new random secret which is used to symmetrically encrypt (e.g. AES-GCM) the backupped data. The encrypted data (along with the PVSS files) is then stored with high durability in mind.

For restoring the data, any $1tn$ of the users cooperate to reassemble the secret key.

CHAPTER

THREE

# PROTOCOL WORKFLOW

The PVSS protocol consists of multiple steps. Each step yields one or multiple messsages that need to be available to the next steps.

## 3.1 Initialization

Mathematical parameters must be chosen, such as a cyclic group and several generators for it.

## 3.2 User key pair generation

Each user generates a private key (which is never disclosed to any other party), computes the public key and shares it.

## 3.3 Secret splitting

The *dealer* randomly generates a secret, computes a share for each user and encrypts each share with the corresponding user's public key. The generated secret is used to encrypt the actual payload. The encrypted payload and the encrypted shares are published.

## 3.4 Recipient key pair generation

As soon as there is need to reassemble the secret, the intended recipient of the secret generates another keypair and shares the public key.

## 3.5 Share re-encryption

Some of the users decrypt their share and re-encrypt it with the recipient's public key.

## 3.6 Secret reassembly

The recipient decrypts those shares and reassembles the secret.

CHAPTER

FOUR

# COMMAND LINE INTERFACE

A single command line utility `pvss` is provided to serve as an example on how to use the API.

Generic usage is `pvss <datadir> <command> [ARGS...]` where `datadir` is some directory which contains all public messages from the PVSS workflow.

Help for the available commands is included in the tool: `pvss --help`

## 4.1 Example

The following sequence of shell commands is executed by six different users who share a data directory. E.g. use git to synchronize it between the users. All files inside `datadir` are public. All files outside of it are private.

```
(init)     $ pvss datadir genparams rst255
(alice)    $ pvss datadir genuser Alice alice.key
(boris)    $ pvss datadir genuser Boris boris.key
(chris)    $ pvss datadir genuser Chris chris.key
(dealer)   $ pvss datadir splitsecret 2 secret0.der
(receiver) $ pvss datadir genreceiver recv.key
(boris)    $ pvss datadir reencrypt boris.key
(alice)    $ pvss datadir reencrypt alice.key
(receiver) $ pvss datadir reconstruct recv.key secret1.der
```

`secret0.der` and `secret1.der` should compare equal. The *dealer* and *receiver* can encrypt an actual payload by using that file as a shared key.

## 4.2 Directory Structure

The `datadir` is made up of:

- `parameters` - Cryptographic group and other parameters (`SystemParameters`).
- `users` - Directory with random file names for each user public key (`PublicKey`).
- `shares` - Shared of the secret (`SharedSecret`).
- `receiver` - Receiver's public key (`PublicKey`).
- `reencrypted` - Directory with random file names for re-encrypted shares (`ReencryptedShare`).

CHAPTER

FIVE

# LIBRARY USAGE

The public API is accessible through the `Pvss` class. Each instance stores the public state of a complete `PVSS` workflow. Messages created in once instance must be transferred somehow (network, git repo, etc.) and be imported into the other instances.

## 5.1 Example

The following code is equivalent to the *CLI example*, if it would be ran inside a single python process:

```
from pvss import Pvss
from pvss.ristretto_255 import create_ristretto_255_parameters

# init, genparams
pvss_init = Pvss()
params = create_ristretto_255_parameters(pvss_init)

# alice, genuser
pvss_alice = Pvss()
pvss_alice.set_params(params)
alice_priv, alice_pub = pvss_alice.create_user_keypair("Alice")

# boris, genuser
pvss_boris = Pvss()
pvss_boris.set_params(params)
boris_priv, boris_pub = pvss_boris.create_user_keypair("Boris")

# chris, genuser
pvss_chris = Pvss()
pvss_chris.set_params(params)
chris_priv, chris_pub = pvss_chris.create_user_keypair("Chris")

# dealer, splitsecret
pvss_dealer = Pvss()
pvss_dealer.set_params(params)
pvss_dealer.add_user_public_key(chris_pub)
pvss_dealer.add_user_public_key(alice_pub)
pvss_dealer.add_user_public_key(boris_pub)
secret0, shares = pvss_dealer.share_secret(2)

# receiver, genreceiver
pvss_receiver = Pvss()
pvss_receiver.set_params(params)
recv_priv, recv_pub = pvss_receiver.create_receiver_keypair("receiver")
```

(continues on next page)

(continued from previous page)

```
# boris, reencrypt
pvss_boris.add_user_public_key(alice_pub)
pvss_boris.add_user_public_key(chris_pub)
pvss_boris.set_shares(shares)
pvss_boris.set_receiver_public_key(recv_pub)
reenc_boris = pvss_boris.reencrypt_share(boris_priv)

# alice, reencrypt
pvss_alice.add_user_public_key(boris_pub)
pvss_alice.add_user_public_key(chris_pub)
pvss_alice.set_shares(shares)
pvss_alice.set_receiver_public_key(recv_pub)
reenc_alice = pvss_alice.reencrypt_share(alice_priv)

# receiver, reconstruct
pvss_receiver.add_user_public_key(boris_pub)
pvss_receiver.add_user_public_key(chris_pub)
pvss_receiver.add_user_public_key(alice_pub)
pvss_receiver.set_shares(shares)
pvss_receiver.add_reencrypted_share(reenc_alice)
pvss_receiver.add_reencrypted_share(reenc_boris)
secret1 = pvss_receiver.reconstruct_secret(recv_priv)

print(secret0 == secret1)
```

## 5.2 API reference

`pvss.qr.`**`create_qr_params`**(*pvss: pvss.pvss.Pvss, params: Union[int, str, ByteString]*) → bytes

Create and set QR parameters.

If params is str or a ByteString, assume it’s a diffie-hellman parameter file such as created by “openssl dhparam 4096”, either DER or PEM encoded.

**Parameters**

- **pvss** – Pvss object with public values
- **params** – if int, must be a safe prime, otherwise must be a DH params file with a safe prime.

**Returns** DER encoded QR system parameters.

`pvss.ristretto_255.`**`create_ristretto_255_parameters`**(*pvss: pvss.pvss.Pvss*) → bytes

Create and set Ristretto255 parameters.

**Parameters** **pvss** – Pvss object with public values

**Returns** DER encoded Ristretto255 system parameters.

**class** `pvss.`**`Pvss`**

Main class to work with Pvss. Stores all public messages and exposes the PVSS operations.

The constructor takes no parameters.

**`add_reencrypted_share`**(*data: ByteString*) → pvss.pvss.ReencryptedShare

Add a re-encrypted share to the internal state.

**Parameters** **data** – DER encoded re-encrypted share.

**Returns** Decoded reencrypted share.

**Raises** **`ValueError`** – On duplicate

**`add_user_public_key`**(*data: ByteString*) → pvss.pvss.PublicKey
Add a user public key to the internal state.

**Parameters** **`data`** – DER encoded public key

**Returns** Decoded user public key.

**Raises** **`ValueError`** – On duplicate name or public key value

**`create_receiver_keypair`**(*name: str*) → Tuple[bytes, bytes]
Create a random key pair for the receiver.

**Parameters** **`name`** – Name of key; will be included in the public key.

**Returns** DER encoded private key and public key

**`create_user_keypair`**(*name: str*) → Tuple[bytes, bytes]
Create a random key pair for a user.

**Parameters** **`name`** – Name of key; will be included in the public key.

**Returns** DER encoded private key and public key

**`property params`**
Retrieve system parameters.

**Returns** The system parameters.

**`property receiver_public_key`**
Retrieve receiver's public key.

**Returns** Receiver's public key.

**`reconstruct_secret`**(*der_private_key: ByteString*) → bytes
Decrypt the re-encrypted shares with the private key and reconstruct the secret

**Parameters** **`der_private_key`** – Receiver's DER encoded private key

**Returns** DER encoded secret

**`reencrypt_share`**(*der_private_key: ByteString*) → bytes
Decrypt a share of the encrypted secret with the private_key and re-encrypt it with another public key

**Parameters** **`der_private_key`** – A user's DER encoded private key

**Returns** DER encoded re-encrypted share

**`property reencrypted_shares`**
Retrieve the list of reencrypted shares.

**Returns** List of reencrypted shares.

**`set_params`**(*data: ByteString*) → pvss.pvss.SystemParameters
Set system parameters.

**Args** data: DER encoded system parameters.

**Returns** Decoded system parameters.

**Raises** **`Exception`** – If already set.

**`set_receiver_public_key`**(*data: ByteString*) → pvss.pvss.PublicKey
Add the receiver's public key to the internal state.

**Parameters** **`data`** – DER encoded receiver's public key.

**Returns** Decoded receiver's public key.

**Raises** **`Exception`** – On duplicate

**`set_shares`** (*data: ByteString*) → pvss.pvss.SharedSecret
Set the shares of the secret.

**Parameters** **`data`** – DER encoded secret shares.

**Returns** Decoded secret shares.

**Raises** **`Exception`** – If already set.

**`share_secret`** (*qualified_size: int*) → Tuple[bytes, bytes]
Create a secret, split it and compute the encrypted shares.

**Parameters** **`qualified_size`** – Number of shares required to reconstruct the secret

**Returns** DER encoded shared secret and the DER encoded encrypted shares

**`property shares`**
Retrieve the shares of the secret.

**Returns** Shares of the secret.

**`property user_public_keys`**
Retrieve all user public keys, as mapping from username to PublicKey.

**Returns** Mapping of username to PublicKey.

CHAPTER

SIX

# ALGORITHMS

## 6.1 Notation used in formulas

| Heissler | Schoen-makers | Tang et al. | Description |
|---|---|---|---|
| $\alpha_{j_0}$ | $\alpha_j$ | $\alpha_j$ | Secret coefficients for $f_0$ |
| $\alpha_{j_1}$ | | $\beta_j$ | Secret coefficients for $f_1$ |
| $a_i$ | | | First part of ElGamal ciphertext. |
| $a_i'$ | | | Randomized commitment for $a_i$. |
| $b_i$ | | | Second part of ElGamal ciphertext. |
| $C_j$ | $C_j$ | $C_j$ | Commitments for the coefficients $\alpha_{j_{0,1}}$. |
| $c$ | $c$ | $c$ | Challenge for zero knowledge proofs, generated by hash function. |
| $e$ | | | Identity element of (image) group. |
| $e'$ | | | Randomized commitment for $e$. |
| $f_0(i)$ | $p(x)$ | $f(x)$ | Polynomial with secret coefficients $\alpha_{j_0}$ |
| $f_1(i)$ | | $g(x)$ | Polynomial with secret coefficients $\alpha_{j_1}$ |
| $G_0$ | $G$ | $G$ | Generator for $G_q$ |
| $G_1$ | | $H$ | Generator for $G_q$ |
| $G_q$ | $G_q$ | $G_q$ | Finite cyclic group of prime order $q$, used as the image group for all group isomorphis |
| $g_0$ | $g$ | $g$ | Generator for $G_q$ |
| $g_1$ | | $h$ | Generator for $G_q$ |
| $i$ | $i$ | $i$ | Unique index for user, $i \in [0, q)$, usually $1 \le i \le n$. |
| $i'$ | $j$ | $j$ | Another iterator for user indices, used during reconstruction. |
| $j$ | $j$ | $j$ | Indices for coefficients, $0 \le jt$ |
| $k_{...}$ | $w$ | $s, t$ | Values $\in_R Z_q$ for computing the Prover's commitment in zero knowledge proofs. |
| $n$ | $n$ | $n$ | Number of users. |
| $q$ | $q$ | $q$ | Size of $G_q$ and $Z_q$ |
| $S$ | $S$ | $S$ | Shared secret $\in G_q$, generated by dealer and reconstructed by receiver. |
| $S_i$ | $S_i$ | $S_i$ | User $i$'s share of the shared secret. |
| $s_{...}$ | $r_i$ | $r_{i1}, r_{i2}$ | Responses for zero knowledge proofs. |
| $t$ | $t$ | $t$ | Size of *qualified subset* of users able to reconstruct the secret, $1 \le t \le n$. |
| $v_{0,1}$ | | | Helper variables used in zero-knowledge proof for re-encryption. |
| $w_{0,1}$ | | | Random values for ElGamal encryption. |
| $X_i$ | $X_i$ | $X_i$ | Shares with alternative generator $g_{0,1}$. |
| $X_i'$ | $a_{1i}/X_i^c$ | $a_{1i}/X_i^c$ | Randomized commitment for $X_i$. |
| $x_i$ | $x_i$ | $x_i$ | Private key $\in_R Z_q^*$ for user $i$ |
| $x_r$ | | | Private key $\in_R Z_q^*$ for recipient |
| $Y_i$ | $Y_i$ | $Y_i$ | Encrypted share for each user. |
| $Y_i'$ | $a_{2i}/Y_i^c$ | $a_{2i}/Y_i^c$ | Randomized commitment for $Y_i$. |

Table 1 – continued from previous page

| Heissler | Schoen-makers | Tang et al. | Description |
|---|---|---|---|
| $y_{i0}$ | $y_i$ | $y_{i1}$ | First public key part for users, $y_{i0} = G_0^{x_i}$ |
| $y_{i1}$ | | $y_{i2}$ | Second public key part for users, $y_{i1} = G_1^{x_i}$ |
| $y_i$ | | y_i | Product of public key parts, $y_i = y_{i0} \cdot y_{i1}$ |
| $y_i'$ | | | Randomized commitment for $y_i$. |
| $y_{r0}$ | | | First public key part for recipient, $y_{r0} = G_0^{x_r}$ |
| $y_{r1}$ | | | Second public key part for recipient, $y_{r1} = G_1^{x_r}$ |
| $Z_q$ | $Z_q$ | $Z_q$ | Additive group of integers modulo prime $q$, used as the pre-image group for all group i |
| $Z_q^*$ | $Z_q^*$ | $Z_q^*$ | $Z_q \setminus \{0\}$ |

## 6.2 Operations

The protocol consists of six steps which are executed in sequence. The last three steps can be repeated if another reconstruction is desired.

Each step requires the public input from all previous steps.

Recipients of public values must check if those values conform to this protocol, e.g. if groups are really of prime order and if group members really are inside the group.

Recipients of public values must also ensure that those value were not modified by a third party. How to accomplish this is out of scope of this chapter.

### 6.2.1 Initialization

Choose a prime order group $G_q$ in which computing discrete logarithms is infeasible. Also choose four distinct generators $g_0, g_1, G_0, G_1$ for it.

No party must know the discrete logarithm of any generator with respect to any other. Therefore those generators must be picked from $G_q$ using a public procedure which follows the concept of nothing-up-my-sleeve, e.g. by appying a cryptographic hash function to sensible input values.

The chosen group and generators are public.

**Example groups to choose from:**

- Ristretto255, a group built upon curve25519.
- Multiplicative group of quadratic residues modulo safe prime $p = 2q + 1$
- NIST P-256 (§ D.1.2.3). Should only be used if required by hardware constraints.

### 6.2.2 Key Pair generation

There are $n$ users. Each is assigned a unique integer $i \in [1, q)$. Typically those are $1 \leq i \leq n$. Each user generates a private key $x_i \in Z_q^*$ and the corresponding public key $y_{i0} = G_0^{x_i}$, $y_{i1} = G_1^{x_i}$.

The private key is kept private and the public key is published.

### 6.2.3 Secret Splitting

The dealer computes a random shared secret $S$ and splits it into encrypted shares $Y_i$ for each user $i$. The dealer also shows that those encrypted shares are consistent by producing a non-interactive zero-knowledge proof of knowledge.

To achieve this, the dealer carries out the following steps:

- Define how many shares are required to reconstruct the secret: $t \in [1, n]$.
  This is also known as the size of the *qualified subset* of users.
- Choose two polynomials with random coefficients:
  - $f_0(i) = \sum_{j=0}^{t-1} \alpha_{j0} i^j,\ \alpha_{j0} \in_R Z_q$
  - $f_1(i) = \sum_{j=0}^{t-1} \alpha_{j1} i^j,\ \alpha_{j1} \in_R Z_q$
- Compute the shared secret:
  $S = G_0^{f_0(0)} G_1^{f_1(0)} = G_0^{\alpha_{00}} G_1^{\alpha_{01}}$.
- Compute commitments for the coefficients:
  $C_j = g_0^{\alpha_{j0}} g_1^{\alpha_{j1}},\ j \in [0, t)$.
- For each user $i$, compute:
  - Random values for commitments:
    $k_{i0}, k_{i1} \in_R Z_q$
  - Encrypted share:
    $Y_i = {y_i}_0^{f_0(i)} {y_i}_1^{f_1(i)}$
  - Random commitment for $Y_i$:
    $Y_i' = {y_i}_0^{k_{i0}} {y_i}_1^{k_{i1}}$
  - Share with alternative generators:
    $X_i = g_0^{f_0(i)} g_1^{f_1(i)}$
  - Random commitment for $X_i$:
    $X_i' = g_0^{k_{i0}} g_1^{k_{i1}}$
- Compute the challenge for the zero knowledge proof using a cryptographic hash function $c = H(G, g_0, g_1, G_0, G_1, C_j, y_i, Y_i, Y_i', X_i, X_i')$ with $j \in [0, t)$ and for all users $i$.

  How those values are serialised into an input for the hash function is not important as long as it is deterministic and impossible to generate the same serialisation for different values. The output of the hash function needs to be a non-negative integer. This can be achieved e.g. by using `sha2_256` and treating the 256 bit output as an integer.
- Compute the response for the zero knowledge proof for each user $i$:
  - $s_{i0} = k_{i0} + cf_0(i)$
  - $s_{i1} = k_{i1} + cf_1(i)$

This proves knowledge of the values $f_0(i)$ and $f_1(i)$ that were used to calculate $X_i, Y_i$.

The dealer then publishes the values $t, c, C_j, Y_i, s_{i0}, s_{i1}$.

The shared secret $S$ can be used to encrypt some payload, e.g. by computing a hash over it and using it as the key for some symmetric encryption function like AES-GCM. It must then be discarded.

The polynomials $f_{0,1}$ and the random values $k_{i0,1}$ are secret and must be discarded.

The values $Y_i', X_i, X_i'$ could be made public, but other parties can recompute them. So they are discarded too.

### Verification

To verify that the public values generated by the splitting operation are consistent, the following steps are carried out:

- For each user $i$, compute:
  - $Y_i' = {y_i}_0^{s_{i0}} {y_i}_1^{s_{i1}} Y_i^{-c}$
  - $X_i = \prod_{j=0}^{t-1} C_j^{(i^j)}$
  - $X_i' = g_0^{s_{i0}} g_1^{s_{i1}} X_i^{-c}$
- Compute the challenge for the zero knowledge proof using a cryptographic hash function $c' = H(G, g_0, g_1, G_0, G_1, C_j, y_i, Y_i, Y_i', X_i, X_i')$ with $j \in [0, t)$ and for all users $i$.
- Verify that $c = c'$

## 6.2.4 Receiver key pair generation

The recipient generates a private key $x_r \in Z_q^*$ and the corresponding public key $y_{r0} = G_0^{x_r}, y_{r1} = G_1^{x_r}$.

The private key is kept private and the public key is published.

If the following two operations are executed quickly, the private key should be kept in ephemeral storage to reduce the risk of subsequential leakage.

## 6.2.5 Share reencryption

A user can decrypt their share and use ElGamal Encryption to re-encrypt it with the receiver's public key. The user needs to produce a non-interactive zero knowledge proof to show that the reencrypted secret was correctly computed.

- Decrypt share by raising it to power of the multiplicative inverse of the user's private key:
  $S_i = Y_i^{\frac{1}{x_i}}$
- Choose two random values:
  $w_0, w_1 \in_R Z_q$
- Re-encrypt decrypted share using ElGamal encryption:
  $a_i = G_0^{w_0} \cdot G_1^{w_1}$
  $b_i = S_i \cdot {y_r}_0^{w_0} \cdot {y_r}_1^{w_1}$

Next, the user needs to prove knowledge of the secret values $x_i, S_i, w_0, w_1$ such that

- $y_i = y_{i0} \cdot y_{i1} = (G_0 \cdot G_1)^{x_i}$
- $Y_i = S_i^{x_i}$
- $a_i = G_0^{w_0} \cdot G_1^{w_1}$
- $b_i = S_i \cdot {y_r}_0^{w_0} \cdot {y_r}_1^{w_1}$

hold. This can't be proven directly because $S_i$ is secret.

- Compute two helper variables:
  $v_0 = -w_0 x_i, \; v_1 = -w_1 x_i$
- Eliminate $S_i$:

$Y_i = b_i^{x_i} \cdot {y_r}_0^{v_0} \cdot {y_r}_1^{v_1}$

- The user then needs to prove
  $v_0 = -w_0 x_i \wedge v_1 = -w_0 x_i$
  which can't be done directly either.
- Instead, the user proves
  $e = a_i^{x_i} \cdot G_0^{v_0} \cdot G_1^{v_1}$
  where $e$ is the identity element of the image group.

The user thus proves knowledge of the secret values $x_i, v_0, v_1, w_0, w_1$ such that

- $y_i = (G_0 \cdot G_1)^{x_i}$
- $a_i = G_0^{w_0} \cdot G_1^{w_1}$
- $Y_i = b_i^{x_i} \cdot {y_r}_0^{v_0} \cdot {y_r}_1^{v_1}$
- $e = a_i^{x_i} \cdot G_0^{v_0} \cdot G_1^{v_1}$

hold. Compute:

- Choose random values $k_{x,v_0,v_1,w_0,w_1} \in_R Z_q$
- Random commitment for $y_i$.
  $y_i' = (G_0 \cdot G_1)^{k_x}$
- Random commitment for $Y_i$.
  $Y_i' = b_i^{k_x} \cdot {y_r}_0^{k_{v_0}} \cdot {y_r}_1^{k_{v_1}}$
- Random commitment for $a_i$.
  $a_i' = G_0^{k_{w_0}} \cdot G_1^{k_{w_1}}$
- Random commitment for $e$.
  $e' = a_i^{k_x} \cdot G_0^{k_{v_0}} \cdot G_1^{k_{v_1}}$

Compute the challenge for the zero knowledge proof using a cryptographic hash function $c = H(G, g_0, g_1, G_0, G_1, XXX, y_{r0}, y_{r1}, y_i', Y_i', a_i', e')$

Compute the response for the zero knowledge proof:

- $s_x = k_x + c x_i$
- $s_{v0} = k_{v0} + c v_0$
- $s_{v1} = k_{v1} + c v_1$
- $s_{w0} = k_{w0} + c w_0$
- $s_{w1} = k_{w1} + c w_1$

The user publishes the values $a_i, b_i, c, s_x, s_{v0}, s_{v1}, s_{w0}, s_{w1}$.

### Verification

To verify that the re-encrypted share was computed correctly, the following steps are carried out:

- $y_i' = (G_0 \cdot G_1)^{s_x} \cdot (y_{i0} \cdot y_{i1})^{-c}$
- $Y_i' = b_i^{s_x} \cdot {y_r}_0^{s_{v0}} \cdot {y_r}_1^{s_{v1}} \cdot Y_i^{-c}$
- $a_i' = G_0^{s_{w0}} \cdot G_1^{s_{w1}} \cdot a_i^{-c}$
- $e' = a_i^{s_x} \cdot G_0^{s_{v0}} \cdot G_1^{s_{v1}}$
- $c' = H(G, g_0, g_1, G_0, G_1, XXX, y_{r0}, y_{r1}, y_i', Y_i', a_i', e')$
- Verify that $c = c'$

### Completeness

An honest prover can always carry out the operations described above to convince any verifier.

### Soundness

Assuming a random oracle model, the hash function might return the value $c_0$ and in a different universe it might return $c_1$ for the same input, where $c_1 = c_0 + 1$. If the prover is somehow able to generate valid $s_{x_i0}$ and $s_{x_i1}$ with high probability, he can e.g. compute $s_{x_i1} - s_{x_i0} = (k_{x_i} + (c_0 + 1)x_i) - (k_{x_i} + c_0 x_i) = x_i$. The same idea is applied to the other secret variables. I.e. even if a "lucky" prover does not know the secret variables, he could easily compute them. We don't believe in such luck but assume that the prover knows the secrets.

This proves knowledge of values $x_i, v_0, v_1, w_0, w_1$ such that:

- $y_i = (G_0 \cdot G_1)^{x_i}$
- $a_i = G_0^{w_0} \cdot G_1^{w_1}$
- $Y_i = b_i^{x_i} \cdot {y_r}_0^{v_0} \cdot {y_r}_1^{v_1}$

To prove that $e = a_i^{x_i} \cdot G_0^{v_0} \cdot G_1^{v_1}$, remember that the verifier computes $e'$ and includes it in the hash input.

$e' = a_i^{s_{x1}} \cdot G_0^{s_{v0}} \cdot G_1^{s_{v1}} = a_i^{k_{x_i} + cx_i} \cdot G_0^{k_{v_0} + cv_0} \cdot G_1^{k_{v_1} + cv_1} = (a_i^{x_i} \cdot G_0^{v_0} \cdot G_1^{v_1})^c \cdot (a_i^{k_{x_i}} \cdot G_0^{k_{v_0}} \cdot G_1^{k_{v_1}})$

If $e = a_i^{x_i} \cdot G_0^{v_0} \cdot G_1^{v_1}$ holds, the prover can easily compute $e' = a_i^{k_{x_i}} \cdot G_0^{k_{v_0}} \cdot G_1^{k_{v_1}}$ which does not depend on $c$. Otherwise, the value of $e'$ would depend on $c$ and vice versa. It may be possible to find such a pair, but it's infeasible. So we assume that it does hold.

Next, substitute $a_i$: $e = (G_0^{w_0} \cdot G_1^{w_1})^{x_i} \cdot G_0^{v_0} \cdot G_1^{v_1} = G_0^{w_0 x_i + v_0} \cdot G_1^{w_0 x_i + v_1}$.

The prover does not know the discrete logarithm of $G_0$ with regards to $G_1$ or vice versa, so we can assume that the prover chose $v_0 = -w_0 x_i \wedge v_1 = -w_1 x_i$.

It follows that $Y_i^{\frac{1}{x_i}} = b_i \cdot {y_r}_0^{-w_0} \cdot {y_r}_1^{-w_1} = S_i$.

#### Zero Knowledge

The response values $s_{...}$ each depend on a different random number $k_{...}$ and are evenly distributed over all possible values. A verifier could generate random responses which obviously would contain no useful information at all. There is no way to distinguish an actual response from a random response.

The challenge $c$ which is provided by the prover is also computed by the verifier, so it doesn't depend on secret information either.

If a verifier can compute the discrete logarithm for any of the random commitments, they could deduce the secret value. But this is just as hard as computing the secret value directly.

### 6.2.6 Secret reconstruction

When at least $t$ users re-encrypted their shares with the receiver's public key, the receiver can reconstruct the secret:

- Decrypt each re-encrypted share:
  $S_i = b_i \cdot a_i^{\frac{1}{x_r}}$
- Reconstruct the secret:
  $S = \prod_i S_i^{\lambda_i},\ \lambda_i = \prod_{i', i' \neq i} \frac{i'}{i'-i}$
  where $i, i'$ are the user indices for all re-encrypted shares.

CHAPTER

SEVEN

# DATA STRUCTURES

PVSS is a protocol between multiple parties who must exchange a number of messages. Those messages are DER encoded ASN.1 structures. This format was chosen because it's well defined and has little overhead. Also, the zero knowledge proofs require computation of a cryptographic hash. The input to the hash function needs to be deterministic.

The contents of the messages can be accessed using any standard ASN.1 tools, e.g.:

```
$ dumpasn1 -ade message
$ openssl asn1parse -inform der -in message
```

## 7.1 Message sizes

For the Ristretto255 group, typical message sizes are:

- `Secret`: 36 Bytes.
- `PreGroupValue`: (up to) 34 Bytes.
- `ImgGroupValue`: 34 Bytes.
- `SystemParameters`: 18 Bytes.
- `PrivateKey`: (up to) 36 Bytes.
- `PublicKey`: $72 + |name|$ Bytes.
- `SharedSecret`: (up to) $44 + 34t + 106n + |names|$ Bytes.
- `ReencryptedShare`: (up to) 279 Bytes.

For the `qr_mod_p` group, the size depends on the safe prime. With a 4096 bit prime, the messages are about 12-16 times as large.

## 7.2 Object Identifiers

Prefix: `1.3.6.1.4.1.55040.1.0` (iso.org.dod.internet.private.enterprise.heissler-informatik.floss.pvss)

Parent: https://github.com/joernheissler/oids

| Suffix | Description |
|---|---|
| 0 | ASN.1 module |
| 1 | Image groups |
| 1.0 | qr_mod_p: Quadratic residues in multiplicative group modulo safe prime p |
| 1.1 | ristretto_255: https://ristretto.group/ |

## 7.3 ASN.1 module

```
PVSS-Module {
    iso(1) org(3) dod(6) internet(1) private(4) enterprise(1)
    heissler-informatik(55040) floss(1) pvss(0) id-mod-pvss(0)
} DEFINITIONS ::=

BEGIN

id-pvss OBJECT IDENTIFIER ::= {
    iso(1) org(3) dod(6) internet(1) private(4) enterprise(1)
    heissler-informatik(55040) floss(1) pvss(0)
}

id-alg OBJECT IDENTIFIER ::= { id-pvss 1 }

-- A pre group value
PreGroupValue ::= INTEGER

-- An image group value; type depends on the algorithm
ImgGroupValue ::= CHOICE {
    qrValue                 INTEGER,
    ecPoint                 OCTET STRING
}

-- System parameters, e.g. the mathematical group
SystemParameters ::= SEQUENCE {
    algorithm               OBJECT IDENTIFIER,
    parameters              ANY DEFINED BY algorithm
}

id-alg-qr OBJECT IDENTIFIER ::= { id-alg 0 }
SystemParametersQr ::=      INTEGER

id-alg-rst255 OBJECT IDENTIFIER ::= { id-alg 1 }
SystemParametersRst255 ::=  NULL

-- A user's public key
PublicKey ::= SEQUENCE {
    name                    UTF8String,
    pub0                    ImgGroupValue,
    pub1                    ImgGroupValue
}

-- A user's private key
PrivateKey ::= SEQUENCE {
    priv                    PreGroupValue
}
```

(continues on next page)

(continued from previous page)

```
-- Secret that is split and reconstructed
Secret ::= SEQUENCE {
    secret                  ImgGroupValue
}

-- Per user values of SharedSecret
Share ::= SEQUENCE {
    pub                     UTF8String,
    share                   ImgGroupValue,
    responseF0              PreGroupValue,
    responseF1              PreGroupValue
}

-- Sequence of per user values of SharedSecret
Shares ::= SEQUENCE OF Share

-- Commitments for polynomial coefficients
Coefficients ::= SEQUENCE OF ImgGroupValue

-- Shares of the secret
SharedSecret ::= SEQUENCE {
    shares                  Shares,
    coefficients            Coefficients,
    challenge               OCTET STRING
}

-- Per user hash input, used for SharesChallenge
HashInputUser ::= SEQUENCE {
    pub                     PublicKey,
    commitment              ImgGroupValue,
    randomCommitment        ImgGroupValue,
    share                   ImgGroupValue,
    randomShare             ImgGroupValue
}

-- Sequence of per user hash input, used for SharesChallenge
HashInputUsers ::= SEQUENCE OF HashInputUser

-- Input to hash function, results in SharedSecret.challenge
SharesChallenge ::= SEQUENCE {
    parameters              SystemParameters,
    coefficients            Coefficients,
    users                   HashInputUsers
}

-- Sequence of all public keys, used for ReencryptedChallenge
PublicKeys ::= SEQUENCE OF PublicKey

-- Input to hash function, results in ReencryptedShare.challenge
ReencryptedChallenge ::= SEQUENCE {
    parameters              SystemParameters,
    publicKeys              PublicKeys,
    shares                  SharedSecret,
    receiverPublicKey       PublicKey,
    randPub                 ImgGroupValue,
    randShare               ImgGroupValue,
```

(continues on next page)

(continued from previous page)

```
    randElgA                ImgGroupValue,
    randId                  ImgGroupValue
}

-- User's share after re-encryption
ReencryptedShare ::= SEQUENCE {
    idx                     INTEGER,
    elgA                    ImgGroupValue,
    elgB                    ImgGroupValue,
    responsePriv            PreGroupValue,
    responseV0              PreGroupValue,
    responseV1              PreGroupValue,
    responseW0              PreGroupValue,
    responseW1              PreGroupValue,
    challenge               OCTET STRING
}

-- Allows auto detection of a message's purpose
PvssContainer ::= CHOICE {
    parameters              [0]  SystemParameters,
    privKey                 [1]  PrivateKey,
    userPub                 [2]  PublicKey,
    recvPub                 [3]  PublicKey,
    sharedSecret            [4]  SharedSecret,
    reencryptedShare        [5]  ReencryptedShare
}

END
```

CHAPTER

# EIGHT

# SECURITY

# CHAPTER NINE

# GLOSSARY

**Dealer** Entity which carries out the split operation

**Keypair generation** Operation which users and the receiver carry out to generate a key pair consisting of a private key.

**Parameter generation** Operation which is carried out once to generate system parameters, e.g. select mathematical groups and their generators.

**Private Key** Private part of a key pair. Element of the image group.

**Public Key** Public part of a key pair. Element of the pre-image group.

**PVSS** Publicly Verifiable Secret Splitting (or … Sharing).

**Receiver** An entity with a key pair which finally reconstructs the secret.

**Reencrypted Share** One of the shares; decrypted with the user's private key and re-encrypted with the receiver's public key.

**Secret** A random secret generated by the Secret Splitting operation.

**Secret Reconstruction** Operation which takes reencrypted shares, decrypts them with the receiver's private key and outputs the secret.

**Secret Splitting** Operation which generated a random secret, splits it in multiple shares and encrypts those shares with the users' public keys.

**Share** One part of the Secret, encrypted with a user's public key.

**Share Reencryption** Operation that a user carries out to decrypt a share and re-encrypt it with the receiver's public key.

**User** An entity with a key pair which receives one of the encrypted shares and can re-encrypt it with the reveiver's public key.

CHAPTER

TEN

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX