Camenisch-Lysyanskaya Signatures

The Camenisch-Lysyanskaya signature is used in anonymity-enhanced cryptography. The original paper was published by Jan Camenisch and Anna Lysyanskaya in 2001 [paper]. In the following example - due to processing constraints - we only sign and verify one message. We will also blindly sign a message and verify it [article]:

Determine

Press button to test

Outline

We have barely got to the point where we can digitally sign our documents, and where many industries still rely on wet signatures. With this, we create a public key (pk) and a private key (sk) and then use our private key to sign something. This then creates a signature (S). Our public key then validates the entity which signed it. But whenever we sign a document, it often reveals our identity, and, possibly other parts of our identity (such as our age, address, and so on). In many cases, though, such as being served in a bar, Peggy should just have to prove that she is over 18 years old, and not have to reveal her name, date of birth and address.

So in a paper-based system, Peggy will have some ID which has a unique identifier that only a trusted entity will be able to create. This might be a driver's licence which has a government watermark on the card. But in an electronic system how do we create the equivalent? Well, normally we use public key encryption to prove identity.

In a credential based system, we create credentials which are signed by a given entity. For example, if Peggy (the prover) is over 18, she will create a credential that will be signed by her private key of the entity which proves that she is over 18 to Victor (the verifier):

```
Name: Peggy
Address: 10 Alice Road
Credential 1: Over 18
-- Signed: Peggy.
```

But what if Peggy doesn't want to reveal her identity, and stay anonymous? How can she now reveal that I am over 18, without revealing anything else about her credentials, and for Peggy to get the credentials in an anonymous way? For this, we need an anonymous credential system, and one of the most widely used is the Camenisch-Lysyanskaya (CL) signature.

Within Camenisch-Lysyanskaya signature (known as a signature with efficient protocols), Peggy can pass Victor a signature which proves "I am over 18", and which will not reveal any other of her credentials. It thus allows:

- Secure two-party computation. This allows a signer (such as the DVLC) to issue a signature to Peggy (the owner of the signature) without learning all the messages that are being signed, or the full signature.
- Zero-knowledge proof. This allows Peggy to prove the required signature on a number of messages, without giving away the signature (or additional sub-information on the messages).

And so Peggy is happy. She has asked the DVLC to prove that she is over 18, and they have provided the signature, of which they have no way of knowing how she is using it, and then Peggy can use this in the bar—or anywhere else—to prove she is over 18.

Signing process

With key pairing we have two cyclic groups (G_1 and G_2), and which are of an order of a prime number (n). A pairing on (G_1, G_2, G_T) defines the function $e : G_1 \times G_2 \rightarrow G_T$, and where g_1 is the generator for G_1 and g_2 is the generator for G_2. If we have the points of U_1 and U_2 on G_1 and V_1 and V_2 on G_2, we get the bilinear mapping of:

$$e(U_1 + U_2, V_1) = e(U_1, V_1) \times e(U_2, V_1)$$

$$e(U_1, V_1 + V_2) = e(U_1, V_1) \times e(U_1, V_2)$$

If U is a point on G_1, and V is a point on G_2, we get:

$$e(aU, bV) = e(U, V)^{ab}$$

If G_1 and G_2 are the same group, we get a symmetric grouping ($G_1 = G_2 = G$, and the following commutative property will apply:

$$e(U^a, V^b) = e(U^b, V^a) = e(U, V)^{ab} == e(V, U)^{ab}$$

The computation of e should be efficient in computation time.

Key generation

We first create a secret key (sk) and a public key (pk), and where the secret key is defined by (x, y, z_i). In this case we have i messages to sign. The public key is then (G, G_T, X, Y, Z_i, W_i), and where g is a point on the G_1 curve and:

$$X = g^x$$

$$Y = g^y$$

$$Z_i = g^{z_i}$$

$$W_i = Y^{z_i}$$

Signature generation

Next produce a random value (α) and we will sign to produce five values (a, b, c, A, B):

$$a = g^\alpha$$

$$b = a^y$$

$$c = a^x commitment^{\alpha x y}$$

$$A_i = a^{z_i}$$

$$B_i = A_i^y$$

The $commitment$ becomes $\prod_{i=1}^{s} Z_i^{m_i}$, and where s is the number of messages (m_i).

Verification

We can verify the signatures by checking the pairings of:

Pairing of (a, Z_i) and (g, A_i)

Pairing of (a, Y) and (g, b)

Pairing of (A_i, Y) and (g, B_i)

Pairing of (g, c) and $(X, a) \prod_{i=1}^{s} (X, B_i^{m_i})$

Proof of signature

For the first mapping (and where g is the generator value for the mapping):

$$e(a, Z_i) = e(g, A_i)$$

$$e(g^\alpha, g^{z_i}) = e(g, a^{z_i})$$

$$e(g^\alpha, g^{z_i}) = e(g, (g^\alpha)^{z_i})$$

$$e(g^\alpha, g^{z_i}) = e(g, (g^{\alpha z_i})$$

$$e(g, g)^{\alpha z_i}) = e(g, g)^{\alpha z_i})$$

And the second one:

$e(a, Y) = e(g, b)$

$e(g^\alpha, g^y) = e(g, a^y)$

$e(g^\alpha, g^y) = e(g, (g^\alpha)^y)$

$e(g^\alpha, g^y) = e(g, g^{\alpha y})$

$e(g, g)^{\alpha y} = e(g, g)^{\alpha y}$

And the third one:

$e(A_i, Y)$ and $e(g, B_i)$

$e(a^{z_i}, g^y)$ and $e(g, A^{y_i})$

$e((g^\alpha)^{z_i}, g^y)$ and $e(g, (a^{z_i})^{y_i})$

$e(g^{\alpha z_i}, g^y)$ and $e(g, (g^\alpha)^{z_i y_i})$

$e(g, g)^{\alpha z_i y}$ and $e(g, g)^{\alpha z_i y}$

Coding

The outline coding using the library from [here] is

```
package edu.jhu.isi.CLSign.helloworld;

import edu.jhu.isi.CLSign.CLSign;
import edu.jhu.isi.CLSign.keygen.KeyPair;
import edu.jhu.isi.CLSign.keygen.PublicKey;
import edu.jhu.isi.CLSign.keygen.SecretKey;
import edu.jhu.isi.CLSign.proof.Proof;
import edu.jhu.isi.CLSign.sign.Signature;
import it.unisa.dia.gas.jpbc.Element;
import it.unisa.dia.gas.plaf.jpbc.field.z.ZrElement;

import java.util.ArrayList;
import java.util.List;


public class helloworld {

        public static void main(String[] args){
                System.out.println("=====Generating keys");
        final int messageSize = 1;
        final KeyPair keyPair = CLSign.keyGen(messageSize);
        final PublicKey pk = keyPair.getPk();
        final SecretKey sk = keyPair.getSk();


                System.out.println("Secret key size (Z):\t"+sk.getZ().size());
                System.out.println("Public key size (Z):\t"+pk.getZ().size());
                System.out.println("Public key size (W):\t"+pk.getW().size());
                System.out.println("Public key (X):\t"+pk.getX());
                System.out.println("Public key (Y):\t"+pk.getY());
        for (int i = 0; i < messageSize; i++) {
                System.out.println("Public key (Z):\t"+pk.getZ(i));
                System.out.println("Public key (W):\t"+pk.getW(i));
        }
        System.out.println("Public key (G1):\t"+pk.getPairing().getG1());
        System.out.println("Public key (G2):\t"+pk.getPairing().getG2());

        List<ZrElement> messages = new ArrayList<>();

        for (int i=0;i<messageSize;i++)
        {
                ZrElement mess = (ZrElement) keyPair.getPk().getPairing().getZr().newRandomElement().getImmutable();
                        messages.add(mess);
        }

                System.out.println("=====Signing messages");
                Signature sigma = CLSign.sign(messages, keyPair);
                System.out.println("Sigma (a):\t"+sigma.getA());
                System.out.println("Sigma (b):\t"+sigma.getB());
                System.out.println("Sigma (c):\t"+sigma.getC());
                List<Element> Alist = sigma.getAList();
                System.out.println("Sigma (A):\t"+Alist.get(0));
                List<Element> Blist = sigma.getBList();
                System.out.println("Sigma (B):\t"+Blist.get(0));



                System.out.println("======Verifying signatures");
                boolean rtn=CLSign.verify(messages, sigma, keyPair.getPk());
                System.out.println("Check signature: "+rtn);

                System.out.println("=====Blind Signing messages");
                final Element commitment = CLSign.commit(messages, keyPair.getPk());
                final Proof proof = CLSign.proofCommitment(commitment, messages, keyPair.getPk());
                sigma = CLSign.signBlind(commitment, proof, keyPair);

                System.out.println("======Verifying signatures");
                rtn=CLSign.verify(messages, sigma, keyPair.getPk());
                System.out.println("Check signature: "+rtn);


        }
}
```

We create our pairing with:

```
int rBits = 160;
int qBits = 512;
final TypeACurveGenerator pairingGenerator = new TypeACurveGenerator(rBits, qBits);
final PairingParameters params = pairingGenerator.generate();
pairing=PairingFactory.getPairing(params);
```

And then with the pairing we can then determine our generator values:

```
final Element generator = pairing.getG1().newRandomElement().getImmutable();
final Element generatorT = pairing.getGT().newRandomElement().getImmutable();
```

To generate $X = g^x$, we can implement:

```
final Element X = generator.powZn(sk.getX());
```

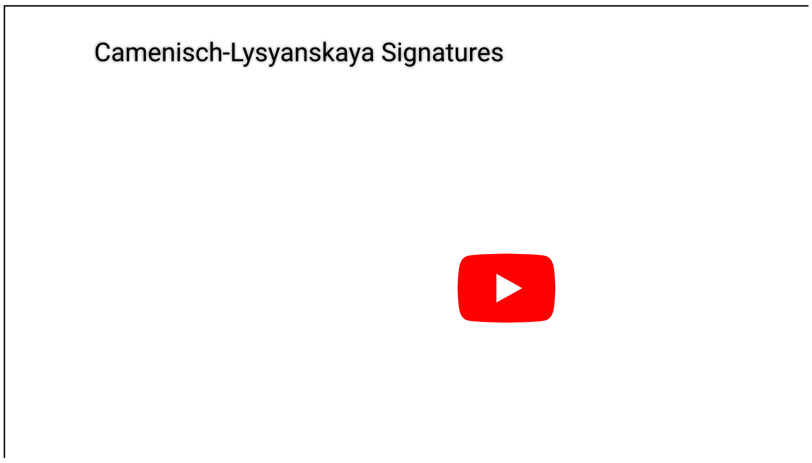
When we are signing we use the private key (sk) an the public key (pk). In the following we use the y value from the secret key to determine a^y:

```
final SecretKey sk = keys.getSk();
b = a.powZn(sk.getY()).getImmutable();
```

In the code we check the pairing of $e(a, Y)$ and $e(g, b)$ with:

```
final Pairing p = pk.getPairing();
if (!p.pairing(sigma.getA(), pk.getY()).isEqual(p.pairing(pk.getGenerator(), sigma.getB()) {}
```

Presentation



Referencing this page

Ref: Buchanan, William J (2023). *Camenisch-Lysyanskaya Signatures*. Asecuritysite.com. https://asecuritysite.com/encryption/cl

Bib: @misc{asecuritysite_48239, title = {Camenisch-Lysyanskaya Signatures}, year={2023}, organization = {Asecuritysite.com}, author = {Buchanan, William J}, url = {https://asecuritysite.com/encryption/cl}, note={Accessed: May 15, 2023}, howpublished= {\url{https://asecuritysite.com/encryption/cl}} }