

Implementation of the Discrete Log Version of Digital Credentials in Java

The Java implementation of the discrete log version is based on the protocol described by Professor Carlisle Adams in “Introduction to Privacy Enhancing Technologies” pages 124-130. The variable names in the Java code match the variable names used in the book, and references to book pages are provided when describing Java methods. The book provides a detailed description of the digital credentials transactions and protocols. Therefore, to keep the report short, the protocols used in Java methods are not discussed in detail.

Class DLUser

The DLUser class (Discrete Log User) represents the user (Alice) in the protocols. This class was implemented using the following libraries:

- org.bouncycastle.util.BigIntegers
- java.io.IOException
- java.math.BigInteger
- java.security.MessageDigest
- java.security.NoSuchAlgorithmException
- java.security.SecureRandom

DLUser Class Attributes and Constructor

```
public class DLUser {

    private String[] stringAttributesArray;
    private BigInteger[] xAttributesArray;
    public DLCA discreteLogCA;
    MessageDigest hash;
    private BigInteger alpha;
    private BigInteger beta;
    private BigInteger gamma;
    public BigInteger h;
    public BigInteger cZeroPrime;
    public BigInteger rZeroPrime;

    public DLUser(DLCA discreteLogCA, String ... args) throws
    NoSuchAlgorithmException {

        this.discreteLogCA = discreteLogCA;
        if(args.length > discreteLogCA.generatorsArray.length){
            throw new IllegalArgumentException("Number of Attributes in User
            Constructor is Greater than the number of CA's generators");
        }
        stringAttributesArray = args.clone();
        xAttributesArray = new BigInteger[stringAttributesArray.length];
        for(int i=0; i<stringAttributesArray.length; ++i){
            xAttributesArray[i] = new
```

```

BigInteger(stringToAscii(stringAttributesArray[i]), 16);
    }

    hash = MessageDigest.getInstance("sha-256");
    alpha = getRandomBigInteger();
    beta = getRandomBigInteger();
    gamma = getRandomBigInteger();
    h = computeH();
}

```

In the above code, “String[] stringAttributesArray” contains the string values of the user’s attributes, whereas “BigInteger[] xAttributesArray” contains the integer (numeric) values of the user’s attributes. The string attributes are converted to their integer (numeric) representations using the “stringToAscii” method. The class attribute “hash = MessageDigest.getInstance(“sha-256”)” is an instance of the sha-256 hash function which will be used for hashing integer values where applicable. Class attributes “alpha,” “beta,” and “gamma” represent random values used by Alice in the transactions as described in page 126 of the book. Class attribute “h” represents Alice’s public key (h in page 126), and the pair “cZeroPrime” and “rZeroPrime” represent the CA’s digital signature (c_0' and r_0' in page 126). The constructor takes “DLCA discreteLogCA” which is an instance of the CA authority and “String ... args” as parameters. The “String ... args” parameter is a Java “vararg” or variable argument containing Alice’s string attributes. Utilizing variable arguments in the constructor allows the program user to instantiate a DLUser object with as many attributes as needed. However, the number of attributes cannot exceed the number of DLCA’s generators.

Method getXAttributesArray()

```

public BigInteger[] getXAttributesArray() {
    return xAttributesArray;
}

```

This getter method returns Alice’s xAttributesArray (used by “DLCA” and “DLVerifier” classes).

Method concatenateBigIntegersHashed(BigInteger a, BigInteger b)

```

public BigInteger concatenateBigIntegersHashed(BigInteger a, BigInteger b) {
    byte[] aStringBytes = a.toByteArray();
    byte[] bStringBytes = b.toByteArray();
    byte[] concatBytesArray = new byte[aStringBytes.length +
bStringBytes.length];
    for (int i = 0; i < aStringBytes.length; ++i) {
        concatBytesArray[i] = aStringBytes[i];
    }
    int index = 0;

    for (int i = aStringBytes.length; i < concatBytesArray.length; ++i) {
        concatBytesArray[i] = bStringBytes[index];
    }
}

```

```

        index += 1;
    }
    byte[] resultBytes = this.hash.digest(concatByteArray);
    return new BigInteger(1, resultBytes);
}

```

This method hashes the concatenation of two BigInteger values. The result is used in computing “cZeroPrime” (corresponding to c_0' in page 126) as well as the signature verification protocol (verification equation of page 126).

Method computeH()

```

private BigInteger computeH() {
    BigInteger h = BigInteger.ONE;
    for(int i=0; i<xAttributesArray.length; ++i){
        h =
h.multiply(discreteLogCA.generatorsArray[i].modPow(xAttributesArray[i],
discreteLogCA.prime));
    }
    h=h.multiply(discreteLogCA.hZero);
    return h.modPow(this.alpha,discreteLogCA.prime);
}

```

This method computes Alice’s public key (h) and corresponds to “h” as described in page 126 of the book.

Method computeCZeroPrime()

```

private BigInteger computeCZeroPrime() {
    discreteLogCA.computeAZero();
    BigInteger gZeroBeta = discreteLogCA.g0.modPow(this.beta,
discreteLogCA.prime);
    BigInteger hPrime = BigInteger.ONE;
    for(int i=0; i<xAttributesArray.length; ++i){
        hPrime =
hPrime.multiply(discreteLogCA.generatorsArray[i].modPow(xAttributesArray[i],
discreteLogCA.prime));
    }
    hPrime=hPrime.multiply(discreteLogCA.hZero);
    hPrime= hPrime.modPow(this.gamma,discreteLogCA.prime);
    BigInteger toBeConcatenated =
gZeroBeta.multiply(hPrime).multiply(discreteLogCA.aZero);
    toBeConcatenated = toBeConcatenated.mod(discreteLogCA.prime);
    BigInteger cZeroPrime = concatenateBigIntegersHashed(this.h,
toBeConcatenated);
    return cZeroPrime;
}

```

This method computes Alice’s “cZeroPrime” (corresponding to c_0' in page 126).

Method computeCZero()

```
private BigInteger computeCZero() {  
    return (cZeroPrime.subtract(this.beta)).mod(discreteLogCA.qOrder);  
}
```

This method computes c_0 (the blinded value c_0 as described in page 126 of the book).

Method verifyRZero(BigInteger rZero, BigInteger cZero)

```
private boolean verifyRZero(BigInteger rZero, BigInteger cZero) {  
    BigInteger gZeroCzero = discreteLogCA.g0.modPow(cZero,  
discreteLogCA.prime);  
    BigInteger hPrime = BigInteger.ONE;  
    for(int i=0; i<xAttributesArray.length; ++i){  
        hPrime =  
hPrime.multiply(discreteLogCA.generatorsArray[i].modPow(xAttributesArray[i],  
discreteLogCA.prime));  
    }  
    hPrime=hPrime.multiply(discreteLogCA.hZero);  
    hPrime = hPrime.modPow(rZero, discreteLogCA.prime);  
    BigInteger verification =  
(hPrime.multiply(gZeroCzero)).mod(discreteLogCA.prime);  
    return verification.equals(discreteLogCA.aZero);  
}
```

This method verifies the “ r_0 ” received from the CA as described in page 126 of the book. If the verification equation for r_0 holds, Alice will proceed to computing “ $rZeroPrime$ ” (r_0' in page 126). Here, $gZeroCzero$ corresponds to $g_0^{c_0}$ in page 126, and $hPrime$ represents $(g_1^{x_1} \cdot g_2^{x_2} \cdot \dots \cdot g_m^{x_m} \cdot h_0)^{r_0}$ in the same page.

Method getAttributesSigned()

```
public void getAttributesSigned() {  
    cZeroPrime = computeCZeroPrime();  
    BigInteger cZero = computeCZero();  
    BigInteger rZero = discreteLogCA.computeRZero(this, cZero);  
    if(!verifyRZero(rZero, cZero)) {  
        System.out.println("Error! r received from the CA is not valid.");  
        return;  
    }  
    BigInteger numerator = rZero.add(this.gamma);  
    BigInteger denominator = (this.alpha).modInverse(discreteLogCA.qOrder);  
    this.rZeroPrime =  
(numerator.multiply(denominator)).mod(discreteLogCA.qOrder);  
}
```

This method computes Alice's "cZeroPrime" and "rZeroPrime" (corresponding to c_0' and r_0' in page 126 of the book). The two values represent Alice's digital signature. This method calls the "verifyRZero (BigInteger rZero, BigInteger cZero)" method described above to verify the r_0 received from the CA. If r_0 is not verified, an error message is printed.

Method stringToAscii(String str)

```
public String stringToAscii(String str) {
    StringBuilder sum = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        sum.append(Integer.toString(str.charAt(i), 16));
    }
    return sum.toString();
}
```

This method returns a string comprised of the concatenation of ASCII codes in the input string. The returned value is used in computing the BigInteger value of Alice's string attributes. For instance, the return value of this function is used in the below loop in the DLUser class's constructor described above:

```
for(int i=0; i<stringAttributesArray.length; ++i){
    xAttributesArray[i] = new
    BigInteger(stringToAscii(stringAttributesArray[i]), 16);
}
```

It is noteworthy that, if desired, other schemes can be used to convert Alice's string attributes to their numeric representations. For example, other encodings (such as UTF-8 and UTF-16) or cryptographic hash functions (such as SHA-256 or SHA-512) can be utilized for this purpose.

Method getRandomBigInteger()

```
public BigInteger getRandomBigInteger() {
    BigInteger min = BigInteger.TWO;
    BigInteger max = discreteLogCA.qOrder.subtract(BigInteger.ONE);
    SecureRandom secureRandom = new SecureRandom();
    return BigIntegers.createRandomInRange(min, max, secureRandom);
}
```

This method returns a random BigInteger in Z_q . Java's secureRandom and Bouncy Castle's BigIntegers libraries are used for this purpose. It is noteworthy that the prime number used by the DLCA class (the class representing the CA) is a safe prime in the form $2q + 1$ where q is also prime. Since the group order is a prime number, there is no need to check if the returned random BigInteger is coprime to the q order of the prime.

Method showAttributes(DLVerifier verifier, int ... shownAttributesIndices)

```
public boolean showAttributes(DLVerifier verifier, int ...
shownAttributesIndices) throws NoSuchAlgorithmException {

    if(shownAttributesIndices.length > xAttributesArray.length){
        throw new IllegalArgumentException("Number of Shown Attributes is
Greater than the User's Number of Attributes");
    }
    if(! verifier.verifySignature(this)){
        System.out.println("Error! The user signature cannot be verified");
        return false;
    }
    int i,j;
    int indexShown = 0;
    int indexConcealed = 0;
    int concealedArraysLength = xAttributesArray.length -
shownAttributesIndices.length;
    BigInteger w = getRandomBigInteger();
    String[] shownStringAttributesArray = new
String[shownAttributesIndices.length];
    BigInteger[] concealedXAttributesArray = new
BigInteger[concealedArraysLength];
    BigInteger[] wSArray = new BigInteger[concealedArraysLength];
    int [] concealedGeneratorsArrayIndices = new int [concealedArraysLength];
    int [] shownGeneratorsArrayIndices = new int
[shownAttributesIndices.length];
    BigInteger[] concealedRsArray = new BigInteger[concealedArraysLength];
    boolean indexFound = false;
    for(i=0; i<xAttributesArray.length; ++i){
        for(j=0;j<shownAttributesIndices.length;++j){
            if(shownAttributesIndices[j] > xAttributesArray.length-1){
                throw new IllegalArgumentException("Shown Attributes index "+
shownAttributesIndices[j] + " does not exist");
            }
            if(i == shownAttributesIndices[j]){
                indexFound = true;
            }
        }
        if(indexFound){
            shownStringAttributesArray[indexShown] =
stringAttributesArray[i];
            shownGeneratorsArrayIndices[indexShown] = i;
            ++indexShown;
        }
        if(!indexFound){
            concealedXAttributesArray[indexConcealed] =
xAttributesArray[i];
            wSArray[indexConcealed] = getRandomBigInteger();
            concealedGeneratorsArrayIndices[indexConcealed] = i;
            ++indexConcealed;
        }
        indexFound = false;
    }
    BigInteger hRaisedToWInverse = (this.h.modPow(w,
```

```

discreteLogCA.prime)).modInverse(discreteLogCA.prime);
    BigInteger concealedGeneratorsRaisedToWs = BigInteger.ONE;
    for(i=0; i<concealedArraysLength; ++i){
        concealedGeneratorsRaisedToWs =
concealedGeneratorsRaisedToWs.multiply(discreteLogCA.generatorsArray[concealedGeneratorsArrayIndices[i]].modPow(wSArray[i], discreteLogCA.prime));
    }
    BigInteger a =
(hRaisedToWInverse.multiply(concealedGeneratorsRaisedToWs)).mod(discreteLogCA.prime);
    BigInteger verifierC = verifier.getVerifierC(this);
    BigInteger cPrime =
verifierC.multiply(this.alpha.modInverse(discreteLogCA.qOrder));
    cPrime = cPrime.add(w);
    cPrime = cPrime.mod(discreteLogCA.qOrder);
    for(i=0; i<concealedArraysLength; ++i){
        concealedRsArray[i] =
verifierC.multiply(concealedXAttributesArray[i]);
        concealedRsArray[i] = concealedRsArray[i].add(wSArray[i]);
        concealedRsArray[i] = concealedRsArray[i].mod(discreteLogCA.qOrder);
    }
    return verifier.verifyShownAttributes(this, shownStringAttributesArray,
shownGeneratorsArrayIndices, concealedGeneratorsArrayIndices, cPrime,
concealedRsArray, a);
}

```

This method is used by Alice to show her attributes (convince Bob that she is the legitimate owner of her public key, and the attribute values it contain) to a DLVerifier (page 128 of the book). This method takes a DLVerifier object and the variable argument “int ... shownAttributesIndices” as parameters. The variable argument array contains the indices of the attributes Alice wishes to reveal to the verifier. Utilizing Java’s variable argument allows the user of the program to show as many attributes as desired. However, for obvious reasons, the number of shown attributes cannot exceed the number of Alice’s attributes.

The method starts by verifying Alice’s signature. If the signature cannot be verified, an error message is printed and “false” is returned. If the signature is verified, the method proceeds to generating the following:

- a random BigInteger w (corresponding to “w” in page 130).
- “String[] shownStringAttributesArray” containing the string values of the attributes Alice wishes to reveal to the verifier.
- “BigInteger[] concealedXAttributesArray” containing the numeric (BigInteger) values of concealed (not revealed) attributes.
- “BigInteger[] wSArray” (corresponding to “ w_1, w_3, \dots, w_m ” in page 130).
- “int [] concealedGeneratorsArrayIndices” and “int [] shownGeneratorsArrayIndices” containing the corresponding indices of the generators used for concealed and shown attributes in the DLCA’s generators array.
- “BigInteger[] concealedRsArray” (containing “ r_1, r_3, \dots, r_m ” in page 130).

Next, the method computes “hRaisedToWInverse” (corresponding to h^{-w} in page 130), “concealedGeneratorsRaisedToWs” (corresponding to $g_1^{w_1} \cdot g_3^{w_3} \cdot \dots \cdot g_m^{w_m}$ in page 130), “a” (corresponding to $a = h^{-w}(g_1^{w_1} \cdot g_3^{w_3} \cdot \dots \cdot g_m^{w_m})$ in page 130), “cPrime” (corresponding to c' in page 130), and fills the “concealedRsArray” with computed “ r_1, r_3, \dots, r_m ” values.

Finally, the above computed values and arrays are used to call the DLVerifier’s “verifyShownAttributes(this, shownStringAttributesArray, shownGeneratorsArrayIndices, concealedGeneratorsArrayIndices, cPrime, concealedRsArray, a)” method. If the shown attributes can be verified, the verifier returns “true.” Otherwise, the verifier returns “false.”

Sample Run of the DLUser class’s main() method

The discrete log implementation of the digital credentials protocol was verified and timed using 1000 DLUser objects as described in the “Testing and Timing” section of the report. However, a sample run of DLUser class’s main() method is provided below:

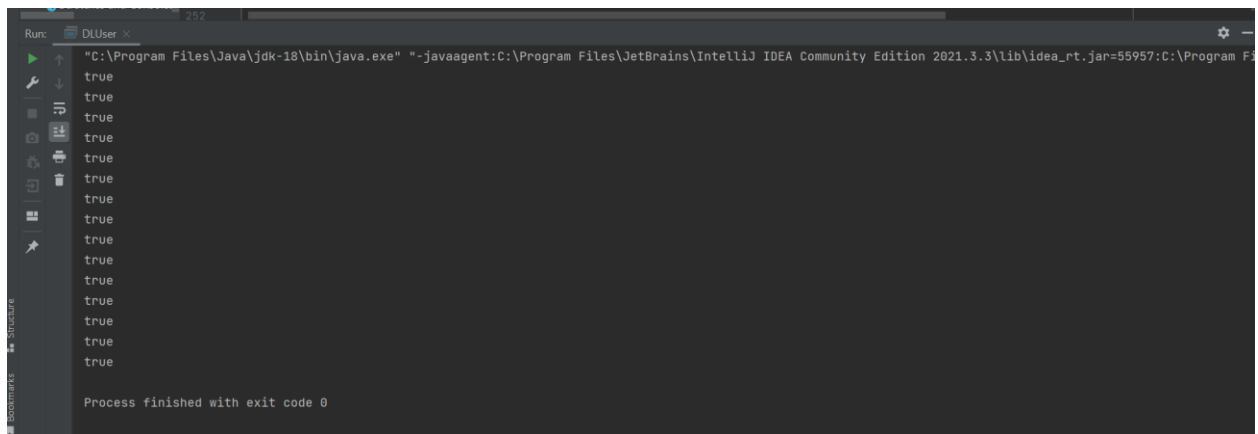
```
public static void main(String[] args) throws NoSuchAlgorithmException,
IOException, InterruptedException {

    DLCA discreteLogCA = new DLCA(20);
    DLVerifier verifier = new DLVerifier();

    DLUser Alice = new DLUser(discreteLogCA, "Alice", "Allison", "1977/09/18",
"541 5th Ave. N",
    "Saskatoon", "SK", "S7K5Z9", "Visa", "12345678",
    "2023/12");

    Alice.getAttributesSigned();
    System.out.println(verifier.verifySignature(Alice));
    System.out.println(verifier.verifyShownAttributes(Alice, 0));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6, 7));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6, 7, 8));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9));
    ;

    DLUser Bob = new DLUser(discreteLogCA, "Bob", "Wilcox", "1999/11/18");
    Bob.getAttributesSigned();
    System.out.println(verifier.verifySignature(Bob));
    System.out.println(Bob.showAttributes(verifier, 0));
    System.out.println(Bob.showAttributes(verifier, 1));
    System.out.println(Bob.showAttributes(verifier, 2, 1, 0));
}
```

DLCA Class

The DLCA class represents the CA in the protocols. The class was implemented using the below libraries:

- org.bouncycastle.util.BigIntegers
- java.math.BigInteger
- java.security.SecureRandom

DLCA Class Attributes and Constructor

```

public class DLCA {
    //BigIntegers prime, g0, ... gn, and hZero are public keys of the CA
    public BigInteger prime;
    public BigInteger qOrder;
    public BigInteger g0;
    public BigInteger[] generatorsArray;
    public BigInteger hZero;
    BigInteger wZero;
    BigInteger aZero;
    //BigIntegers xPrivateKey, y1, ... yn are random numbers which are
private keys of the CA
    private BigInteger xPrivateKey;
    private BigInteger[] yPrivateKeysArray;

    public DLCA(int numberOfGenerators){

        //3072-bit prime with generator 2 obtained from
https://www.ietf.org/rfc/rfc3526.txt
        this.prime = new
BigInteger("FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD1" +
            "29024E088A67CC74020BBEA63B139B22514A08798E3404DD" +
            "EF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C245" +
            "E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7ED" +
            "EE386BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3D" +
            "C2007CB8A163BF0598DA48361C55D39A69163FA8FD24CF5F" +
            "83655D23DCA3AD961C62F356208552BB9ED529077096966D" +
            "670C354E4ABC9804F1746C08CA18217C32905E462E36CE3B" +

```

```

"E39E772C180E86039B2783A2EC07A28FB5C55DF06F4C52C9" +
"DE2BCBF6955817183995497CEA956AE515D2261898FA0510" +
"15728E5A8AAC42DAD33170D04507A33A85521ABDF1CBA64" +
"ECFB850458DBEF0A8AEA71575D060C7DB3970F85A6E1E4C7" +
"ABF5AE8CDB0933D71E8C94E04A25619DCEE3D2261AD2EE6B" +
"F12FFA06D98A0864D87602733EC86A64521F2B18177B200C" +
"BBE117577A615D6C770988C0BAD946E208E24FA074E5AB31" +
"43DB5BFCE0FD108E4B82D120A93AD2CAFFFFFFFFFFFFFFFF", 16);

qOrder =
(this.prime.subtract(BigInteger.ONE)).divide(BigInteger.TWO);
xPrivateKey = getRandomBigInteger();
yPrivateKeysArray = new BigInteger[numberOfGenerators];
for(int i=0; i< yPrivateKeysArray.length;++i){
    yPrivateKeysArray[i] = getRandomBigInteger();
}

g0 = new BigInteger("2",16);
generatorsArray = new BigInteger[yPrivateKeysArray.length];
for(int i=0; i<generatorsArray.length; ++i){
    generatorsArray[i] = g0.modPow(yPrivateKeysArray[i], prime);
}
hZero = g0.modPow(xPrivateKey,prime);
}

```

In the above code, `BigInteger` “prime” and “qOrder” represent the prime and the order of the generator. `BigInteger` “g0” represents the base or generator of G_q (g_0 in page 125). “`BigInteger[] generatorsArray`” contains the generators g_1, \dots, g_m ($g_i = g_0^{y_i}$) as described in page 125. `BigInteger` “xPrivateKey” represents x_0 in page 125, and `BigInteger` “hZero” represents $h_0 = g_0^{x_0}$ in the same page. “`BigInteger[] yPrivateKeysArray`” is the array containing y_1, \dots, y_m as described in page 125. Thus, “xPrivateKey” and “yPrivateKeysArray” represent the CA’s private keys, and “g0”, “generatorsArray,” and “h0” represent the CA’s public keys.

The constructor takes “int numberOfGenerators” as its parameter. This allows the user of the program to instantiate a DLCA object with the desired number of generators in the “generatorsArray.” Hence, a “DLUser” object obtaining its signature from a CA can have a variable number of attributes (from 1 up to the number of the CA’s generators). The prime number used in this implementation is the 3072-bit RFC3526 prime (matching 128-bit security level) with base generator 2 (obtained from <https://www.rfc-editor.org/rfc/rfc3526>). RFC3526 primes (1536-bit, 2048-bit, 3072-bit, 4096-bit, 6144-bit, 8192-bit) are all safe primes ($p=2q+1$ where q is a prime) with base generator 2. The chosen generator ($g = 2$) satisfies $g^q \bmod p = 1$. Thus, the order of the base generator ($g = 2$) for RFC3526 primes is half the group order (<https://crypto.stackexchange.com/questions/87870/how-is-a-generator-found-for-a-group-both-in-case-of-dh-ecdh>). This describes why the order of G_q “qOrder” is calculated as $(\text{this.prime.subtract(BigInteger.ONE)).divide(BigInteger.TWO})$.

The constructor chooses a random `BigInteger` as “x0” and fills “yPrivateKeysArray” with random `BigInteger`s. As described, the value of “g0” (the base generator) is 2, and “generatorsArray” is filled with generators g_1, \dots, g_m ($g_i = g_0^{y_i}$). Finally, the value of “hZero” is calculated as $h_0 = g_0^{x_0}$.

Method getRandomBigInteger()

```
public BigInteger getRandomBigInteger() {  
  
    BigInteger min = BigInteger.TWO;  
    BigInteger max = qOrder.subtract(BigInteger.ONE);  
    SecureRandom secureRandom = new SecureRandom();  
    return BigIntegers.createRandomInRange(min, max, secureRandom);  
}
```

This method This method returns a random BigInteger in Z_q (similar to getRandomBigInteger() method in the DLUser class).

Method computeAzero()

```
public void computeAzero() {  
  
    this.wZero = getRandomBigInteger();  
    this.aZero = g0.modPow(wZero, prime);  
}
```

This method computes $a_0 = g_0^{w_0}$ as described in page 126 of the book.

Method computeUserXy(BigInteger[] userXAttributesArray)

```
private BigInteger computeUserXy(BigInteger[] userXAttributesArray) {  
    BigInteger sum = BigInteger.ZERO;  
    for(int i =0; i< userXAttributesArray.length; ++i){  
        sum =  
sum.add(userXAttributesArray[i].multiply(yPrivateKeysArray[i]));  
    }  
    return sum;  
}
```

This method computes $(x_1y_1 + \dots + x_my_m)$ as described in page 126 of the book. The return value is used for computing “rZero” (r_0 in page 126).

Method computeRZero(DLUser user, BigInteger cZero)

```
public BigInteger computeRZero(DLUser user, BigInteger cZero) {  
  
    BigInteger denominator =  
this.xPrivateKey.add(computeUserXy(user.getXAttributesArray()));  
    BigInteger numerator = this.wZero.subtract(cZero);  
    denominator = denominator.modInverse(qOrder);  
    return (numerator.multiply(denominator)).mod(qOrder);  
}
```

```
}
```

This method computes r_0 as described in page 126 of the book. The computed value is sent to the DLUser object calling the method and is used for computing r_0' by the DLUser object.

Class DLVerifier

This class represents the verifier in the digital credentials protocols. The class was implemented using the below libraries:

- import org.bouncycastle.util.BigIntegers
- import java.math.BigInteger
- import java.security.MessageDigest
- import java.security.NoSuchAlgorithmException
- import java.security.SecureRandom

DLVerifier Class Attributes and Constructor

```
public class DLVerifier {  
    public BigInteger verifierC;
```

This class has only one attribute “verifierC” which represents the “random number c” chosen by Bob (the verifier) in page 128 of the book. The class uses Java’s default constructor.

Method verifySignature(DLUser user)

```
public boolean verifySignature(DLUser user) throws NoSuchAlgorithmException {  
    BigInteger gZeroCZeroPrime =  
user.discreteLogCA.g0.modPow(user.cZeroPrime, user.discreteLogCA.prime);  
    BigInteger hRZeroPrime = user.h.modPow(user.rZeroPrime,  
user.discreteLogCA.prime);  
    BigInteger toBeConcatenated =  
(gZeroCZeroPrime.multiply(hRZeroPrime)).mod(user.discreteLogCA.prime);  
    BigInteger verification = concatenateBigIntegersHashed(user.h,  
toBeConcatenated);  
    boolean result = verification.equals(user.cZeroPrime);  
    if(!result){  
        System.out.println("Signature Cannot Be Verified");  
    }  
    return result;  
}
```

This method verifies the signature of a “DLUser” object according to the verification equation described in page 126 of the book. BigInteger “gZeroCZeroPrime” represents $g_0^{c_0'}$ in the equation (the “DLCA’s” base generator raised to the power of “DLUser’s” c_0'). BigInteger “hRZeroPrime” represents $h^{r_0'}$ in the

equation (the public key (h) of the “DLUser” object raised to the power of the user’s r_0). The method verifies the DLUser’s signature by checking if the “DLUser’s” c_0 is equal to hashed concatenated value of the “DLUser’s” h and $(g_0^{c_0} \cdot h^{r_0})$ according to the verification equation.

Method concatenateBigIntegersHashed(BigInteger a, BigInteger b)

```
public BigInteger concatenateBigIntegersHashed(BigInteger a, BigInteger b)
throws NoSuchAlgorithmException {

    byte[] aStringBytes = a.toByteArray();
    byte[] bStringBytes = b.toByteArray();
    byte[] concatByteArray = new byte[aStringBytes.length +
bStringBytes.length];
    for (int i = 0; i < aStringBytes.length; ++i) {
        concatByteArray[i] = aStringBytes[i];
    }
    int index = 0;

    for (int i = aStringBytes.length; i < concatByteArray.length; ++i) {
        concatByteArray[i] = bStringBytes[index];
        index += 1;
    }
    MessageDigest hash = MessageDigest.getInstance("sha-256");
    byte[] resultBytes = hash.digest(concatByteArray);
    return new BigInteger(1, resultBytes);
}
```

This method concatenates two BigInteger values and hashes the concatenated value (similar to “concatenateBigIntegersHashed” method of the DLUser class).

Method getVerifierC(DLUser user)

```
public BigInteger getVerifierC(DLUser user) {
    verifierC = BigIntegers.createRandomInRange(BigInteger.TWO,
user.discreteLogCA.qOrder.subtract(BigInteger.ONE), new SecureRandom());
    return verifierC;
}
```

This method returns the “random number c ” chosen by Bob (the verifier) as described in page 128 of the book. The method also sets the “DLVerifier” class’s “verifierC” attribute with the chosen random number. This value will be later used to verify the “DLUser’s” shown attributes.

Method stringToAscii(String str)

```
public String stringToAscii(String str) {
    StringBuilder sum = new StringBuilder();
```

```

    for (int i = 0; i < str.length(); i++) {
        sum.append(Integer.toString(str.charAt(i), 16));
    }
    return sum.toString();
}

```

This method returns a string comprised of the concatenation of ASCII codes in the input string (similar to “stringToAscii” method in the “DLUser” class).

Method verifyShownAttributes()

DLUser user, String[] shownStringAttributesArray, int[] shownGeneratorsArrayIndices, int[] concealedGeneratorsArrayIndices, BigInteger cPrime, BigInteger[] concealedRsArray, BigInteger a)

```

public boolean verifyShownAttributes(DLUser user, String[]
shownStringAttributesArray, int[] shownGeneratorsArrayIndices, int[]
concealedGeneratorsArrayIndices, BigInteger cPrime, BigInteger[]
concealedRsArray, BigInteger a){

    int i;
    BigInteger hRaisedToCPrime = user.h.modPow(cPrime,
user.discreteLogCA.prime);
    BigInteger hRaisedToCPrimeA =
(hRaisedToCPrime.multiply(a)).mod(user.discreteLogCA.prime);
    BigInteger shownGeneratorsRaisedToShownXAttributes = BigInteger.ONE;
    BigInteger[] shownXAttributesArray = new
BigInteger[shownStringAttributesArray.length];
    for(i=0; i< shownXAttributesArray.length; ++i){
        shownXAttributesArray[i]= new
BigInteger(stringToAscii(shownStringAttributesArray[i]),16);
    }

    for(i = 0; i<shownXAttributesArray.length; ++i){
        shownGeneratorsRaisedToShownXAttributes =
shownGeneratorsRaisedToShownXAttributes.multiply(user.discreteLogCA.generator
sArray[shownGeneratorsArrayIndices[i]].modPow((shownXAttributesArray[i].multi
ply(verifierC)), user.discreteLogCA.prime));
    }

    BigInteger concealedGeneratorsRaisedToConcealedRs = BigInteger.ONE;
    for(i=0; i< concealedGeneratorsArrayIndices.length; ++i){
        concealedGeneratorsRaisedToConcealedRs =
concealedGeneratorsRaisedToConcealedRs.multiply(user.discreteLogCA.generators
Array[
concealedGeneratorsArrayIndices[i]].modPow(concealedRsArray[i],
user.discreteLogCA.prime)
);
    }
}

```

```

        BigInteger hZeroRaisedToC = user.discreteLogCA.hZero.modPow(verifierC,
user.discreteLogCA.prime);

        BigInteger verification =
hZeroRaisedToC.multiply(concealedGeneratorsRaisedToConcealedRs).multiply(show
nGeneratorsRaisedToShownXAttributes);
        verification = verification.mod(user.discreteLogCA.prime);

        boolean result = verification.equals(hRaisedToCPrimeA);
        if(!result){
            System.out.println("Attribute(s) Cannot Be Verified");
        }

        return result;
    }
}

```

This method verifies a DLUser's shown or revealed attributes according to the protocol described in page 128 of the book. In the above code, "String[] shownStringAttributesArray" contains the string values of the attributes the DLUser object wishes to reveal, "int[] shownGeneratorsArrayIndices" contains the indices of the generators used for "shown attributes" in the DLCA's "generatorsArray," "int[] concealedGeneratorsArrayIndices" contains the indices of the generators used for "concealed" or "not revealed" attributes in the DLCA's "generatorsArray, BigInteger "cPrime" corresponds to the value "c" computed by Alice in page 130 of the book, "BigInteger[] concealedRsArray" contains the values "r₁, r₃, ..., r_m" computed by Alice, and "BigInteger a" represents the value " $a = h^{-w}(g_1^{w_1} \cdot g_3^{w_3} \cdot \dots \cdot g_m^{w_m})$ " computed by Alice in page 130 of the book.

This method computes "hRaisedToCPrimeA" corresponding to $(h^c \cdot a)$ in page 130 of the book. Next, "shownXAttributesArray" is filled with the numeric (BigInteger) representations of the revealed string attributes by calling the "stringToAscii" method for each revealed string attribute. After this step, the value "shownGeneratorsRaisedToShownXAttributes" is computed using a for loop. This value corresponds to $g_2^{cx_2}$ in page 130 of the book. It is noteworthy that Alice is only showing one attribute in book's description of the protocol, however, Alice may reveal more than one attribute and thus the value "shownGeneratorsRaisedToShownXAttributes" is calculated as described above. After this step, the value "concealedGeneratorsRaisedToConcealedRs" is computed which corresponds to " $g_1^{r_1} \cdot g_3^{r_3} \cdot \dots \cdot g_m^{r_m}$ " in page 130 of the book. Next, the value "hZeroRaisedToC" is computed which corresponds to " h_0^c " in page 130 of the book. Finally, the method verifies if the value "hRaisedToCPrimeA" (corresponding to " $h^c \cdot a$ " in the book) is equal to the value "hZeroRaisedToC.multiply(concealedGeneratorsRaisedToConcealedRs).multiply(shownGeneratorsRaisedToShownXAttributes)" which represents $(g_1^{r_1} \cdot g_2^{17c} \cdot g_3^{r_3} \cdot \dots \cdot g_m^{r_m} \cdot h_0^c)$ in page 130 of the book. If the verification holds, the method returns "true." Otherwise, an error message is printed, and the method returns "false."

Method verifyShownAttributes(DLUser user, int ... shownAttributesIndices)

```

public boolean verifyShownAttributes(DLUser user, int ...
shownAttributesIndices) throws NoSuchAlgorithmException {

```

```
return user.showAttributes(this, shownAttributesIndices);  
}
```

This is an overloaded method which calls the “showAttributes” method of the “DLUser” object. It is implemented so that a “DLVerifier” object can directly verify a user’s shown attributes.

Implementation of the Elliptic Curve Version of Digital Credentials in Java

The elliptic curve version of digital credentials was implemented using the Bouncy Castle cryptographic API (<https://www.bouncycastle.org/>). Classes and methods in the elliptic curve version use the same names, structures, variable names, and have the same functionality as their discrete log version counterparts. They are different in that arithmetic operations have been replaced with ECPoint (elliptic curve point) operations, and BigInteger values have been substituted with ECPoints where applicable.

It is worth noting that the “getInfinity()” method of a bouncy castle elliptic curve returns the infinity point, which is the identity element of the ECPoint addition operations and corresponds to BigInteger.ONE in the arithmetic multiplication operation. It is also worth mentioning that the “negate()” method of an ECPoint returns the inverse of the ECPoint.

The code is also available at <https://github.com/siavash2012/Digital-Credentials-in-Java>.

Class ECUser

The ECUser class (Elliptic Curve User) represents the user (Alice) in the protocols. This class was implemented using the following libraries:

- org.bouncycastle.math.ec.ECPoint
- org.bouncycastle.util.BigIntegers
- java.math.BigInteger
- java.security.MessageDigest
- java.security.NoSuchAlgorithmException
- java.security.SecureRandom

ECUser Class Attributes and Constructor

```
public class ECUser {  
  
    private String[] stringAttributesArray;  
    private BigInteger[] xAttributesArray;  
    public ECCA ellipticCurveCA;  
    MessageDigest hash;  
    private BigInteger alpha;  
}
```



```

private BigInteger beta;
private BigInteger gamma;
public ECPoint h;
public BigInteger cZeroPrime;
public BigInteger rZeroPrime;

    public ECUser(ECCA ellipticCurveCA, String ... args) throws
NoSuchAlgorithmException {

        this.ellipticCurveCA = ellipticCurveCA;
        if(args.length > ellipticCurveCA.generatorsArray.length){
            throw new IllegalArgumentException("Number of Attributes in User
Constructor is Greater than the number of CA's generators");
        }
        stringAttributesArray = args.clone();
        xAttributesArray = new BigInteger[stringAttributesArray.length];
        for(int i=0; i<stringAttributesArray.length; ++i){
            xAttributesArray[i] = new
BigInteger(stringToAscii(stringAttributesArray[i]), 16);
        }
        hash = MessageDigest.getInstance("sha-256");
        alpha = getRandomBigInteger();
        beta = getRandomBigInteger();
        gamma = getRandomBigInteger();
        h = computeH();
    }

```

In the above code, “String[] stringAttributesArray” contains the string values of the user’s attributes, whereas “BigInteger[] xAttributesArray” contains the integer (numeric) values of the user’s attributes. The string attributes are converted to their integer (numeric) representations using the “stringToAscii” method. The class attribute “hash = MessageDigest.getInstance(“sha-256”)” is an instance of the sha-256 hash function which will be used for hashing integer values where applicable. Class attributes “alpha,” “beta,” and “gamma” represent random values used by Alice in the transactions as described in page 126 of the book. Class attribute “ECPoint h” represents Alice’s public key (h in page 126), and the pair “cZeroPrime” and “rZeroPrime” represent the CA’s digital signature (c_0' and r_0' in page 126). The constructor takes “ECCA ellipticCurveCA” which is an instance of the CA authority and “String ... args” as parameters. The “String ... args” parameter is a Java “vararg” or variable argument containing Alice’s string attributes. Utilizing variable arguments in the constructor allows the program user to instantiate a ECUser object with as many attributes as needed. However, the number of attributes cannot exceed the number of ECCA’s generators.

Method getXAttributesArray()

```

public BigInteger[] getXAttributesArray() {
    return xAttributesArray;
}

```

This getter method returns Alice's `xAttributesArray` (used by "ECCA" and "ECVerifier" classes).

Method `concatenateBigIntegersHashed(BigInteger a, BigInteger b)`

```
public BigInteger concatenateBigIntegersHashed(BigInteger a, BigInteger b) {
    byte[] aStringBytes = a.toByteArray();
    byte[] bStringBytes = b.toByteArray();
    byte[] concatBytesArray = new byte[aStringBytes.length +
bStringBytes.length];
    for (int i = 0; i < aStringBytes.length; ++i) {
        concatBytesArray[i] = aStringBytes[i];
    }
    int index = 0;

    for (int i = aStringBytes.length; i < concatBytesArray.length; ++i) {
        concatBytesArray[i] = bStringBytes[index];
        index += 1;
    }
    byte[] resultBytes = this.hash.digest(concatBytesArray);
    return new BigInteger(1, resultBytes);
}
```

This method hashes the concatenation of two `BigInteger` values. The result is used in computing "`cZeroPrime`" (corresponding to c_0 in page 126) as well as the signature verification protocol (verification equation of page 126).

Method `computeH()`

```
private ECPoint computeH() {
    ECPoint h = ellipticCurveCA.bcCurve.getInfinity();
    for(int i=0; i<xAttributesArray.length; ++i){
        h =
h.add(ellipticCurveCA.generatorsArray[i].multiply(xAttributesArray[i]));
    }
    h=h.add(ellipticCurveCA.hZero);
    h= h.multiply(this.alpha);
    return h.normalize();
}
```

This method computes Alice's public key (`ECPoint h`) and corresponds to "`h`" as described in page 126 of the book. The "`normalize()`" method of the `ECPoint` class ensures that any projective coordinate of the `ECPoint` is 1 (coordinates reflect those of the equivalent point in an affine coordinate system). Normalizing `ECPoint "h"` is necessary since if "`h`" is not in its normal form, attempting to get its x affine coordinate will result in an exception.

Method `computeCZeroPrime()`

```

private BigInteger computeCZeroPrime() {

    ellipticCurveCA.computeAZero();
    ECPoint gZeroBeta = (ellipticCurveCA.g0).multiply(this.beta);
    ECPoint hPrime = ellipticCurveCA.bcCurve.getInfinity();
    for(int i=0; i<xAttributesArray.length; ++i){
        hPrime =
hPrime.add(ellipticCurveCA.generatorsArray[i].multiply(xAttributesArray[i]));
    }
    hPrime=hPrime.add(ellipticCurveCA.hZero);
    hPrime=hPrime.multiply(this.gamma);
    ECPoint toBeConcatenated =
gZeroBeta.add(hPrime).add(ellipticCurveCA.aZero);
    toBeConcatenated = toBeConcatenated.normalize();
    BigInteger cZeroPrime =
concatenateBigIntegersHashed(this.h.getAffineXCoord().toBigInteger(),
toBeConcatenated.getAffineXCoord().toBigInteger());
    return cZeroPrime;
}

```

This method computes Alice's "cZeroPrime" (corresponding to c_0' in page 126). It is noteworthy that for computing cZeroPrime, the x coordinates (affine x coordinates) of the two ECPoints are concatenated and hashed.

Method computeCZero()

```

private BigInteger computeCZero() {

    return (cZeroPrime.subtract(this.beta)).mod(ellipticCurveCA.qOrder);
}

```

This method computes c_0 (the blinded value c_0 as described in page 126 of the book). It is noteworthy that in the elliptic curve implementation of digital credentials, qOrder is the order of the elliptic curve used by the ECCA (elliptic curve CA).

Method verifyRZero(BigInteger rZero, BigInteger cZero)

```

private boolean verifyRZero(BigInteger rZero, BigInteger cZero) {

    ECPoint gZeroCzero = (ellipticCurveCA.g0).multiply(cZero);
    ECPoint hPrime = ellipticCurveCA.bcCurve.getInfinity();
    for(int i=0; i<xAttributesArray.length; ++i){
        hPrime =
hPrime.add(ellipticCurveCA.generatorsArray[i].multiply(xAttributesArray[i]));
    }
    hPrime=hPrime.add(ellipticCurveCA.hZero);
    hPrime = hPrime.multiply(rZero);
    ECPoint verification = hPrime.add(gZeroCzero);
    return verification.equals(ellipticCurveCA.aZero);
}

```

This method verifies the “ r_0 ” received from the CA as described in page 126 of the book. If the verification equation for r_0 holds, Alice will proceed to computing “ $rZeroPrime$ ” (r_0' in page 126). Here, $gZeroCzero$ corresponds to $g_0^{c_0}$ in page 126, and $hPrime$ represents $(g_1x_1 \cdot g_2x_2 \cdot \dots \cdot g_mx_m \cdot h_0)^{r_0}$ in the same page.

Method `getAttributesSigned()`

```
public void getAttributesSigned() {
    cZeroPrime = computeCZeroPrime();
    BigInteger cZero = computeCZero();
    BigInteger rZero = ellipticCurveCA.computeRZero(this, cZero);
    if (!verifyRZero(rZero, cZero)) {
        System.out.println("Error! r received from the CA is not valid.");
        return;
    }
    BigInteger numerator = rZero.add(this.gamma);
    BigInteger denominator = (this.alpha).modInverse(ellipticCurveCA.qOrder);
    this.rZeroPrime =
(numerator.multiply(denominator)).mod(ellipticCurveCA.qOrder);
}
```

This method computes Alice’s “ $cZeroPrime$ ” and “ $rZeroPrime$ ” (corresponding to c_0' and r_0' in page 126 of the book). The two values represent Alice’s digital signature. This method calls the “`verifyRZero` (`BigInteger rZero`, `BigInteger cZero`)” method described above to verify the r_0 received from the CA. If r_0 is not verified, an error message is printed.

Method `stringToAscii(String str)`

```
public String stringToAscii(String str) {
    StringBuilder sum = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        sum.append(Integer.toString(str.charAt(i), 16));
    }
    return sum.toString();
}
```

This method returns a string comprised of the concatenation of ASCII codes in the input string. The returned value is used in computing the `BigInteger` value of Alice’s string attributes. For instance, the return value of this function is used in the below loop in the `ECUser` class’s constructor described above:

```
for(int i=0; i<stringAttributesArray.length; ++i){
    xAttributesArray[i] = new
BigInteger(stringToAscii(stringAttributesArray[i]), 16);
}
```

It is noteworthy that, if desired, other schemes can be used to convert Alice's string attributes to their numeric representations. For example, other encodings (such as UTF-8 and UTF-16) or cryptographic hash functions (such as SHA-256 or SHA-512) can be utilized for this purpose.

Method getRandomBigInteger()

```
public BigInteger getRandomBigInteger() {  
  
    BigInteger min = BigInteger.TWO;  
    BigInteger max = ellipticCurveCA.qOrder.subtract(BigInteger.ONE);  
    SecureRandom secureRandom = new SecureRandom();  
    return BigIntegers.createRandomInRange(min, max, secureRandom);  
}
```

This method returns a random BigInteger in Z_q (the ECVerifier's elliptic curve order). Java's secureRandom and Bouncy Castle's BigIntegers libraries are used for this purpose. It is noteworthy that the order of elliptic curves provided by the Bouncy Castle API (including curve "secp256r1" used by ECCA (elliptic curve CA) in this implementation) is prime. Thus, there is no need to check if the returned random BigInteger is coprime to the order (qOrder) of the elliptic curve.

Method showAttributes(ECVerifier verifier, int ... shownAttributesIndices)

```
public boolean showAttributes(ECVerifier verifier, int ...  
shownAttributesIndices) throws NoSuchAlgorithmException {  
  
    if(shownAttributesIndices.length > xAttributesArray.length){  
        throw new IllegalArgumentException("Number of Shown Attributes is  
Greater than the User's Number of Attributes");  
    }  
  
    if(! verifier.verifySignature(this)){  
        System.out.println("Error! The user signature cannot be verified");  
        return false;  
    }  
  
    int i,j;  
    int indexShown = 0;  
    int indexConcealed = 0;  
    int concealedArraysLength = xAttributesArray.length -  
shownAttributesIndices.length;  
    BigInteger w = getRandomBigInteger();  
    String[] shownStringAttributesArray = new  
String[shownAttributesIndices.length];  
    BigInteger[] concealedXAttributesArray = new  
BigInteger[concealedArraysLength];  
    BigInteger[] wSArray = new BigInteger[concealedArraysLength];  
    int[] concealedGeneratorsArrayIndices = new int[concealedArraysLength];  
    int[] shownGeneratorsArrayIndices = new  
int[shownAttributesIndices.length];  
    BigInteger[] concealedRsArray = new BigInteger[concealedArraysLength];  
    boolean indexFound = false;
```

```

        for(i=0; i<xAttributesArray.length; ++i){
            for(j=0; j<shownAttributesIndices.length; ++j){
                if(shownAttributesIndices[j] > xAttributesArray.length-1){
                    throw new IllegalArgumentException("Shown Attributes index "+
shownAttributesIndices[j] + " does not exist");
                }
                if(i == shownAttributesIndices[j]){
                    indexFound = true;
                }
            }
            if(indexFound){
                shownStringAttributesArray[indexShown] =
stringAttributesArray[i];
                shownGeneratorsArrayIndices[indexShown] = i;
                ++indexShown;
            }
            if(!indexFound){
                concealedXAttributesArray[indexConcealed] =
xAttributesArray[i];
                wSArray[indexConcealed] = getRandomBigInteger();
                concealedGeneratorsArrayIndices[indexConcealed] = i;
                ++indexConcealed;
            }
            indexFound = false;
        }

        ECPoint hRaisedToWInverse = (this.h.multiply(w)).negate();

        ECPoint concealedGeneratorsRaisedToWs =
ellipticCurveCA.bcCurve.getInfinity();

        for(i=0; i<concealedArraysLength; ++i){
            concealedGeneratorsRaisedToWs =
concealedGeneratorsRaisedToWs.add(ellipticCurveCA.generatorsArray[concealedGe
neratorsArrayIndices[i]].multiply(wSArray[i]));
        }

        ECPoint a = (hRaisedToWInverse.add(concealedGeneratorsRaisedToWs));
        BigInteger verifierC = verifier.getVerifierC(this);
        BigInteger cPrime =
verifierC.multiply(this.alpha.modInverse(ellipticCurveCA.qOrder));
        cPrime = cPrime.add(w);
        cPrime = cPrime.mod(ellipticCurveCA.qOrder);

        for(i=0; i<concealedArraysLength; ++i){
            concealedRsArray[i] =
verifierC.multiply(concealedXAttributesArray[i]);
            concealedRsArray[i] = concealedRsArray[i].add(wSArray[i]);
            concealedRsArray[i] =
concealedRsArray[i].mod(ellipticCurveCA.qOrder);
        }

        return verifier.verifyShownAttributes(this, shownStringAttributesArray,
shownGeneratorsArrayIndices, concealedGeneratorsArrayIndices, cPrime,
concealedRsArray, a);
    }

```

This method is used by Alice to show her attributes (convince Bob that she is the legitimate owner of her public key, and the attribute values it contain) to an ECVerifier (page 128 of the book). This method takes an ECVerifier object and the variable argument "int ... shownAttributesIndices" as parameters. The variable argument array contains the indices of the attributes Alice wishes to reveal to the verifier. Utilizing Java's variable argument allows the user of the program to show as many attributes as desired. However, for obvious reasons, the number of shown attributes cannot exceed the number of Alice's attributes.

The method starts by verifying Alice's signature. If the signature cannot be verified, an error message is printed and "false" is returned. If the signature is verified, the method proceeds to generating the following:

- a random BigInteger w (corresponding to "w" in page 130).
- "String[] shownStringAttributesArray" containing the string values of the attributes Alice wishes to reveal to the verifier.
- "BigInteger[] concealedXAttributesArray" containing the numeric (BigInteger) values of concealed (not revealed) attributes.
- "BigInteger[] wSArray" (corresponding to " w_1, w_3, \dots, w_m " in page 130).
- "int [] concealedGeneratorsArrayIndices" and "int [] shownGeneratorsArrayIndices" containing the corresponding indices of the generators used for concealed and shown attributes in the ECCA's generators array.
- "BigInteger[] concealedRsArray" (containing " r_1, r_3, \dots, r_m " in page 130).

Next, the method computes ECPPoint "hRaisedToWInverse" (corresponding to h^{-w} in page 130), ECPPoint "concealedGeneratorsRaisedToWs" (corresponding to $g_1^{w_1} \cdot g_3^{w_3} \cdot \dots \cdot g_m^{w_m}$ in page 130), ECPPoint "a" (corresponding to $a = h^{-w}(g_1^{w_1} \cdot g_3^{w_3} \cdot \dots \cdot g_m^{w_m})$ in page 130), BigInteger "cPrime" (corresponding to c' in page 130) and fills the "concealedRsArray" with computed " r_1, r_3, \dots, r_m " values.

Finally, the above computed values and arrays are used to call the ECVerifier's "verifyShownAttributes(this, shownStringAttributesArray, shownGeneratorsArrayIndices, concealedGeneratorsArrayIndices, cPrime, concealedRsArray, a)" method. If the shown attributes can be verified, the verifier returns "true." Otherwise, the verifier returns "false."

Sample Run of the ECUser class's main() method

The elliptic curve implementation of the digital credentials protocol was verified and timed using 1000 ECUser objects as described in the "Testing and Timing" section of the report. However, a sample run of ECUser class's main() method is provided below:

```
public static void main(String[] args) throws NoSuchAlgorithmException {
    ECCA ellipticCurveCA = new ECCA(20);
    ECVerifier verifier = new ECVerifier();

    ECUser Alice = new ECUser(ellipticCurveCA, "Alice", "Allison",
    "1979/09/19", "541 5th Ave. N",
```

```

        "Saskatoon", "SK", "S7K5Z9", "Visa", "12345678",
        "2023/12");
    Alice.getAttributesSigned();
    System.out.println(verifier.verifySignature(Alice));
    System.out.println(verifier.verifyShownAttributes(Alice, 0));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5));
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6));

    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6, 7));

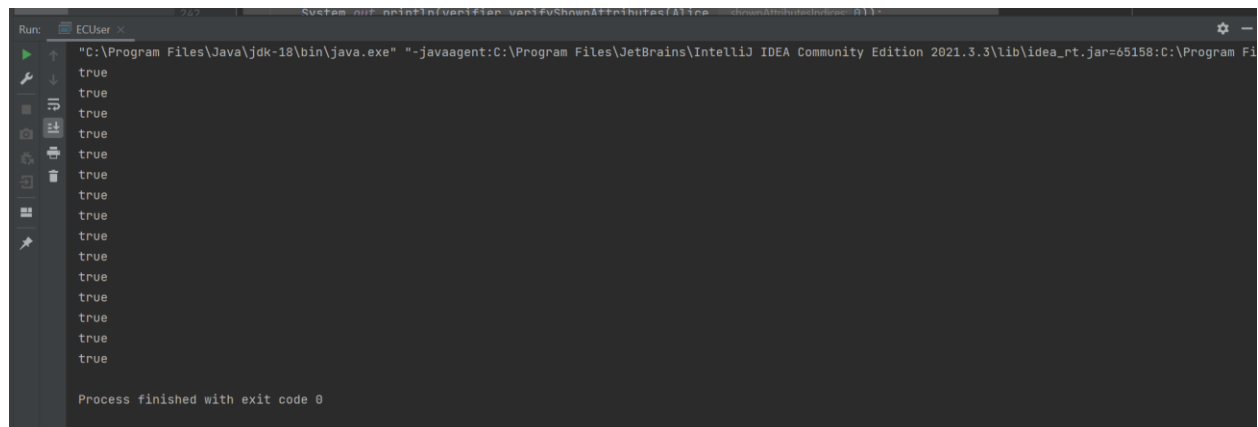
    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6, 7, 8));

    System.out.println(verifier.verifyShownAttributes(Alice, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
    ;

    ECUser Bob = new ECUser(ellipticCurveCA, "Bob", "Johnson", "1987");
    Bob.getAttributesSigned();
    System.out.println(verifier.verifySignature(Bob));
    System.out.println(Bob.showAttributes(verifier, 0));
    System.out.println(Bob.showAttributes(verifier, 1));
    System.out.println(Bob.showAttributes(verifier, 2, 1, 0));
}

```

Output



```

Run: ECUser
"C:\Program Files\Java\jdk-18\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.3.3\lib\idea_rt.jar=65158:C:\Program Fi
true
true
true
true
true
true
true
true
true
true
true
true
true
true
true
Process finished with exit code 0

```

ECCA Class

The ECCA class represents the CA in the protocols. The class was implemented using the below libraries:

- org.bouncycastle.jce.ECNamedCurveTable
- org.bouncycastle.jce.spec.ECNamedCurveParameterSpec
- org.bouncycastle.math.ec.ECPoint
- org.bouncycastle.util.BigIntegers

- java.math.BigInteger
- java.security.SecureRandom

ECCA Class Attributes and Constructor

```
public class ECCA {
    //BigIntegers prime, g0, ... gn (generatorsArray elements), and h0 are
    public keys of the CA
    public BigInteger qOrder;
    public ECPoint g0;
    public ECPoint[] generatorsArray;
    public ECPoint hZero;
    public BigInteger wZero;
    public ECPoint aZero;

    //BigIntegers xPrivateKey, y1, ... yn are random numbers which are
    private keys of the CA
    private BigInteger xPrivateKey;
    private BigInteger[] yPrivateKeysArray;
    public org.bouncycastle.math.ec.ECCurve bcCurve;
    private ECNamedCurveParameterSpec spec;

    public ECCA(int numberOfGenerators){

        spec = ECNamedCurveTable.getParameterSpec("secp256r1");
        bcCurve = spec.getCurve();
        qOrder = bcCurve.getOrder();
        xPrivateKey = getRandomBigInteger();
        yPrivateKeysArray = new BigInteger[numberOfGenerators];
        for(int i=0; i< yPrivateKeysArray.length; ++i){
            yPrivateKeysArray[i] = getRandomBigInteger();
        }

        g0 = spec.getG();
        generatorsArray = new ECPoint[yPrivateKeysArray.length];
        for(int i=0; i<generatorsArray.length; ++i){
            generatorsArray[i] = g0.multiply(yPrivateKeysArray[i]);
        }

        hZero = g0.multiply(xPrivateKey);
    }
}
```

In the above code, “bcCurve” and “qOrder” represent the elliptic curve and the order of the curve. ECPoint “g0” represents the base or generator (g_0 in page 125) and is instantiated using the get(G) method of ECNamedCurveTable.getParameterSpec(“secp256r1”). Elliptic curve “secp256r1” (also known as NIST P-256) matching the 128-bit level of security was used in this implementation.

The Bouncy Castle API supports various elliptic curves at different security levels. If desired, one can change the elliptic curve used in the implementation by simply replacing “secp256r1” with the desired curve name. I tried using curve “secp256k1” (a Koblitz curve) and it worked equally well. A list of SECP

(Standards for Efficient Cryptography) elliptic curves available in the Bouncy castle API can be printed using the below code:

```
public static void main(String[] args){

    SECNamedCurves secNames = new SECNamedCurves();
    Enumeration enSec=secNames.getNames();
    int count=0;
    while(enSec.hasMoreElements()){
        System.out.print(enSec.nextElement().toString());
        System.out.print("\t");
        ++count;
        if(count %5==0){
            System.out.println();
        }
    }
}
```

Output List of SEC Elliptic Curves Available in the Bouncy Castle API

sect283r1	sect283k1	sect163r2	secp256k1	secp160k1
secp160r1	secp112r2	secp112r1	sect113r2	sect113r1
sect239k1	secp128r2	sect163r1	secp128r1	sect233r1
sect163k1	sect233k1	sect193r2	sect193r1	sect131r2
sect131r1	secp256r1	sect571r1	sect571k1	secp192r1
sect409r1	sect409k1	secp521r1	secp384r1	secp224r1
secp224k1	secp192k1	secp160r2		

Back to describing the ECCA class implementation, “ECPPoint[] generatorsArray” contains the generators g_1, \dots, g_m ($g_i = g_0^{y_i}$) as described in page 125. BigInteger “xPrivateKey” represents x_0 in page 125, and ECPPoint “hZero” represents $h_0 = g_0^{x_0}$ in the same page. “BigInteger[] yPrivateKeysArray” is the array containing y_1, \dots, y_m as described in page 125. Thus, “xPrivateKey” and “yPrivateKeysArray” represent the CA’s private keys, and “g0”, “generatorsArray,” and “h0” represent the CA’s public keys.

The constructor takes “int numberOfGenerators” as its parameter. This allows the user of the program to instantiate an ECCA object with the desired number of generators in the “generatorsArray.” Hence, an “ECUser” object obtaining its signature from a CA can have a variable number of attributes (from 1 up to the number of the CA’s generators). The order of the elliptic curve “qOrder” is instantiated using the “getG()” method.

The constructor choses a random BigInteger as “x0” and fills “yPrivateKeysArray” with random BigIntegers. As described, the “generatorsArray” is filled with generators g_1, \dots, g_m ($g_i = g_0^{y_i}$). Finally, the value of “hZero” is calculated as $h_0 = g_0^{x_0}$.

Method getRandomBigInteger()

```
public BigInteger getRandomBigInteger() {

    BigInteger min = BigInteger.TWO;
```

```

    BigInteger max = qOrder.subtract(BigInteger.ONE);
    SecureRandom secureRandom = new SecureRandom();
    return BigIntegers.createRandomInRange(min, max, secureRandom);
}

```

This method This method returns a random BigInteger in Z_q (similar to getRandomBigInteger() method in the rest of the implementation).

Method computeAZero()

```

public void computeAZero() {
    this.wZero = getRandomBigInteger();
    this.aZero = g0.multiply(wZero);
}

```

This method computes $a_0 = g_0^{w_0}$ as described in page 126 of the book.

Method computeUserXy(BigInteger[] userXAttributesArray)

```

private BigInteger computeUserXy(BigInteger[] userXAttributesArray) {
    BigInteger sum = BigInteger.ZERO;
    for(int i =0; i< userXAttributesArray.length; ++i){
        sum =
sum.add(userXAttributesArray[i].multiply(yPrivateKeysArray[i]));
    }
    return sum;
}

```

This method computes $(x_1y_1 + \dots + x_my_m)$ as described in page 126 of the book. The return value is used for computing “rZero” (r_0 in page 126).

Method computeRZero(ECUser user, BigInteger cZero)

```

public BigInteger computeRZero(ECUser user, BigInteger cZero){
    BigInteger denominator =
this.xPrivateKey.add(computeUserXy(user.getXAttributesArray()));
    BigInteger numerator = this.wZero.subtract(cZero);
    denominator = denominator.modInverse(qOrder);
    return (numerator.multiply(denominator)).mod(qOrder);
}

```

This method computes r_0 as described in page 126 of the book. The computed value is sent to the ECUser object calling the method and is used for computing r_0' by the ECUser object.

Class ECVerifier

This class represents the verifier in the digital credentials protocols. The class was implemented using the below libraries:

- org.bouncycastle.math.ec.ECPoint
- org.bouncycastle.util.BigIntegers
- java.math.BigInteger
- java.security.MessageDigest
- java.security.NoSuchAlgorithmException
- java.security.SecureRandom

ECVerifier Class Attributes and Constructor

```
public class ECVerifier {  
    public BigInteger verifierC;
```

This class has only one attribute “verifierC” which represents the “random number c ” chosen by Bob (the verifier) in page 128 of the book. The class uses Java’s default constructor.

Method verifySignature(ECUser user)

```
public boolean verifySignature(ECUser user) throws NoSuchAlgorithmException {  
    ECPoint gZeroCZeroPrime =  
user.ellipticCurveCA.g0.multiply(user.cZeroPrime);  
    ECPoint hRZeroPrime = user.h.multiply(user.rZeroPrime);  
    ECPoint toBeConcatenated = gZeroCZeroPrime.add(hRZeroPrime);  
    toBeConcatenated = toBeConcatenated.normalize();  
    BigInteger verification =  
concatenateBigIntegersHashed(user.h.getAffineXCoord().toBigInteger(),  
toBeConcatenated.getAffineXCoord().toBigInteger());  
    boolean result = verification.equals(user.cZeroPrime);  
    if (!result) {  
        System.out.println("Signature Cannot Be Verified");  
    }  
    return result;  
}
```

This method verifies the signature of an ECUser object according to the verification equation described in page 126 of the book. ECPoint “gZeroCZeroPrime” represents $g_0^{c_0'}$ in the equation (the “ECCA’s” base generator raised to the power of the ECUser’s c_0'). ECPoint “hRZeroPrime” represents $h^{r_0'}$ in the equation (the public key (h) of the ECUser object raised to the power of the user’s r_0'). This method

verifies the ECUser's signature by verifying that the ECUser's c_0' is equal to the hashed concatenated value of the ECUser's h and $(g_0^{c_0'} \cdot h^{r_0'})$ according to the verification equation.

Method concatenateBigIntegersHashed(BigInteger a, BigInteger b)

```
public BigInteger concatenateBigIntegersHashed(BigInteger a, BigInteger b)
throws NoSuchAlgorithmException {

    byte[] aStringBytes = a.toByteArray();
    byte[] bStringBytes = b.toByteArray();
    byte[] concatByteArray = new byte[aStringBytes.length +
bStringBytes.length];
    for (int i = 0; i < aStringBytes.length; ++i) {
        concatByteArray[i] = aStringBytes[i];
    }
    int index = 0;

    for (int i = aStringBytes.length; i < concatByteArray.length; ++i) {
        concatByteArray[i] = bStringBytes[index];
        index += 1;
    }
    MessageDigest hash = MessageDigest.getInstance("sha-256");
    byte[] resultBytes = hash.digest(concatByteArray);
    return new BigInteger(1, resultBytes);
}
```

This method concatenates two BigInteger values and hashes the concatenated value (similar to “concatenateBigIntegersHashed” method in the rest of the implementation).

Method getVerifierC(ECUser user)

```
public BigInteger getVerifierC(ECUser user) {
    verifierC = BigIntegers.createRandomInRange(BigInteger.TWO,
user.ellipticCurveCA.qOrder.subtract(BigInteger.ONE), new SecureRandom());
    return verifierC;
}
```

This method returns the “random number c ” chosen by Bob (the verifier) as described in page 128 of the book. The method also sets the ECVerifier class's “verifierC” attribute with the chosen random number. This value will be later used to verify the ECUser's shown attributes.

Method stringToAscii(String str)

```
public String stringToAscii(String str) {
    StringBuilder sum = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        sum.append(Integer.toString(str.charAt(i), 16));
    }
}
```

```

    }
    return sum.toString();
}

```

This method returns a string comprised of the concatenation of ASCII codes in the input string (similar to “stringToAscii” method in the rest of the implementation).

Method verifyShownAttributes(ECUser user, String[] shownStringAttributesArray, int[] shownGeneratorsArrayIndices, int[] concealedGeneratorsArrayIndices, BigInteger cPrime, BigInteger[] concealedRsArray, ECPPoint a)

```

public boolean verifyShownAttributes(ECUser user, String[]
shownStringAttributesArray, int[] shownGeneratorsArrayIndices, int[]
concealedGeneratorsArrayIndices, BigInteger cPrime, BigInteger[]
concealedRsArray, ECPPoint a) {

    int i;
    ECPPoint hRaisedToCPrime = user.h.multiply(cPrime);
    ECPPoint hRaisedToCPrimeA = hRaisedToCPrime.add(a);
    ECPPoint shownGeneratorsRaisedToShownXAttributes =
user.ellipticCurveCA.bcCurve.getInfinity();
    BigInteger[] shownXAttributesArray = new
BigInteger[shownStringAttributesArray.length];
    for(i=0; i< shownXAttributesArray.length; ++i){
        shownXAttributesArray[i]= new
BigInteger(stringToAscii(shownStringAttributesArray[i]),16);
    }

    for(i = 0; i<shownXAttributesArray.length; ++i){
        shownGeneratorsRaisedToShownXAttributes =
shownGeneratorsRaisedToShownXAttributes.add(user.ellipticCurveCA.generatorsAr
ray[shownGeneratorsArrayIndices[i]].multiply((shownXAttributesArray[i].multip
ly(verifierC))));
    }

    ECPPoint concealedGeneratorsRaisedToConcealedRs =
user.ellipticCurveCA.bcCurve.getInfinity();
    for(i=0; i< concealedGeneratorsArrayIndices.length; ++i){
        concealedGeneratorsRaisedToConcealedRs =
concealedGeneratorsRaisedToConcealedRs.add(
user.ellipticCurveCA.generatorsArray[concealedGeneratorsArrayIndices[i]].mult
iply(concealedRsArray[i]));
    }

    ECPPoint hZeroRaisedToC = user.ellipticCurveCA.hZero.multiply(verifierC);

    ECPPoint verification =
hZeroRaisedToC.add(concealedGeneratorsRaisedToConcealedRs).add(shownGenerator
sRaisedToShownXAttributes);
}

```

```

        boolean result = verification.equals(hRaisedToCPrimeA);
        if(!result){
            System.out.println("Attribute(s) Cannot Be Verified");
        }

        return result;
    }
}

```

This method verifies an ECUUser's shown (or revealed) attributes according to the protocol described in page 128 of the book. In the above code, "String[] shownStringAttributesArray" contains the string values of the attributes the ECUUser object wishes to reveal, "int[] shownGeneratorsArrayIndices" contains the indices of the generators used for "shown attributes" in the ECCA's "generatorsArray," "int[] concealedGeneratorsArrayIndices" contains the indices of the generators used for "concealed" or "not revealed" attributes in the ECCA's "generatorsArray, BigInteger "cPrime" corresponds to the value "c" computed by Alice in page 130 of the book, "BigInteger[] concealedRsArray" contains the values "r₁, r₃, ..., r_m" computed by Alice, and "ECPoint a" represents the value " $a = h^{-w}(g_1^{w_1} \cdot g_3^{w_3} \cdot \dots \cdot g_m^{w_m})$ " computed by Alice in page 130 of the book.

This method computes "ECPoint hRaisedToCPrimeA" corresponding to $(h^c \cdot a)$ in page 130 of the book. Next, "shownXAttributesArray" is filled with the numeric (BigInteger) representations of the revealed string attributes by calling the "stringToAscii" method for each revealed string attribute. After this step, the "ECPoint shownGeneratorsRaisedToShownXAttributes" is computed using a for loop. This value corresponds to $g_2^{cx^2}$ in page 130 of the book. It is noteworthy that Alice is only showing one attribute in the book's description of the protocol, however, Alice may reveal more than one attribute and thus the value "shownGeneratorsRaisedToShownXAttributes" is calculated as described above. After this step, "ECPoint concealedGeneratorsRaisedToConcealedRs" is computed which corresponds to $g_1^{r_1} \cdot g_3^{r_3} \cdot \dots \cdot g_m^{r_m}$ in page 130 of the book. Next, "ECPoint hZeroRaisedToC" is computed which corresponds to " h_0^c " in page 130 of the book. Finally, the method verifies if the value "hRaisedToCPrimeA" (corresponding to " $h^c \cdot a$ " in the book) is equal to the value " $hZeroRaisedToC.add(concealedGeneratorsRaisedToConcealedRs).add(shownGeneratorsRaisedToShownXAttributes)$ " which represents $(g_1^{r_1} \cdot g_2^{17c} \cdot g_3^{r_3} \cdot \dots \cdot g_m^{r_m} \cdot h_0^c)$ in the book. If the verification holds, the method returns "true." Otherwise, an error message is printed, and the method returns "false."

Method verifyShownAttributes(ECUser user, int... shownAttributesIndices)

```

public boolean verifyShownAttributes(ECUser user, int...
shownAttributesIndices) throws NoSuchAlgorithmException {

    return user.showAttributes(this, shownAttributesIndices);
}

```

This is an overloaded method which calls the "showAttributes" method of the ECUUser object. It is implemented so that an ECVerifier object can directly verify a user's shown attributes.

Converting ECPoint Public Keys to Byte[] and Vice Versa

In this implementation, public keys of ECUser and ECCA classes are left as ECPoints (elliptic curve points). However, if desired, ECPoints can be converted to byte[]'s using the "getEncoded()" method of the ECPoint class. Converting byte[]'s to ECPoints is possible using the elliptic curve's "decodePoint()" method. The code below shows the encoding and decoding of ECPoints.

```
public static void main(String[] args){
    //creating a random point for testing
    ECCA testCA = new ECCA(5);
    ECPoint testPoint = testCA.g0.multiply(new BigInteger("128"));
    //use false for encoding without compression, true for encoding in
    compressed format
    byte[] testPointBytes = testPoint.getEncoded(false);
    ECPoint testPointFromBytes = testCA.bcCurve.decodePoint(testPointBytes);
    System.out.println(testPoint.equals(testPointFromBytes));
    //The points are equal, but for printing and visualizing equality,
    normalize the point to convert from Jacobian coordinates to affine
    coordinates
    System.out.println(testPoint.normalize());
    System.out.println(testPointFromBytes);
}
```

Output

```
ECCA x
"C:\Program Files\Java\jdk-18\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.3.3\lib\idea_rt.
true
(ae3f7dba0bde8b6ad7ce2f8eede4b762c556dea678f859626a9e6235a674c4f6,1c0549fc0a69995a24b8c213249db9a97940508d085f8a5c1ee0553e711f4b53,1)
(ae3f7dba0bde8b6ad7ce2f8eede4b762c556dea678f859626a9e6235a674c4f6,1c0549fc0a69995a24b8c213249db9a97940508d085f8a5c1ee0553e711f4b53,1)
```

Testing and Timing Implementations

The testing and timing of the two implementations (discrete log and elliptic curve) was carried out as follows:

An excel sheet (1000 rows, 10 columns) containing randomly generated attributes was downloaded from <https://www.fakenamegenerator.com/order.php> . The screenshot below shows the first 10 rows of the sheet.

	A	B	C	D	E	F	G	H	I	J
1	GivenName	Surname	Birthday	StreetAddress	City	State	ZipCode	CCType	CCNumber	CCExpires
2	David	Berry	9/2/1981	4984 rue Ellice	Joliette	QC	J6E 3E8	MasterCard	5301603335262701	9/2026
3	Julio	Morrison	3/8/1953	4986 Kinchant St	Chief Lake	BC	V2N 2J1	Visa	4929892151218260	11/2025
4	Kenneth	Perreault	8/30/1949	3606 Toy Avenue	Oshawa	ON	L1G 6Z8	Visa	4716085949394868	11/2026
5	Sharon	Bailey	6/29/2003	4835 rue Saint-Édouard	Trois Rivières	QC	G9A 5S8	Visa	4916812585907804	12/2026
6	Mary	Olson	12/1/1984	2086 Edson Drive	Hinton	AB	T7V 1E8	Visa	4532518496722919	2/2028
7	Lawrence	Graves	6/27/1976	3803 A Avenue	Edmonton	AB	T5J 0K7	Visa	4916927466679999	11/2025
8	Alfred	Sheehan	12/29/1939	4536 Charleton Ave	Hamilton	ON	L8N 3X3	MasterCard	5127192448079014	11/2027
9	Beth	Sampson	10/3/1951	1371 chemin Georges	Lavaltrie	QC	J0K 1H0	Visa	4916193346142076	10/2025
10	Monte	Gutierrez	12/17/1958	720 St. Paul Street	St Catharines	ON	L2V 3G2	MasterCard	5401369976126139	1/2025

The excel sheet was saved as a CSV file. The CSV file was read row by row using the “readTestFile” method implemented in the “TextIO” class. The strings read from the columns of each row were added to an ArrayList.

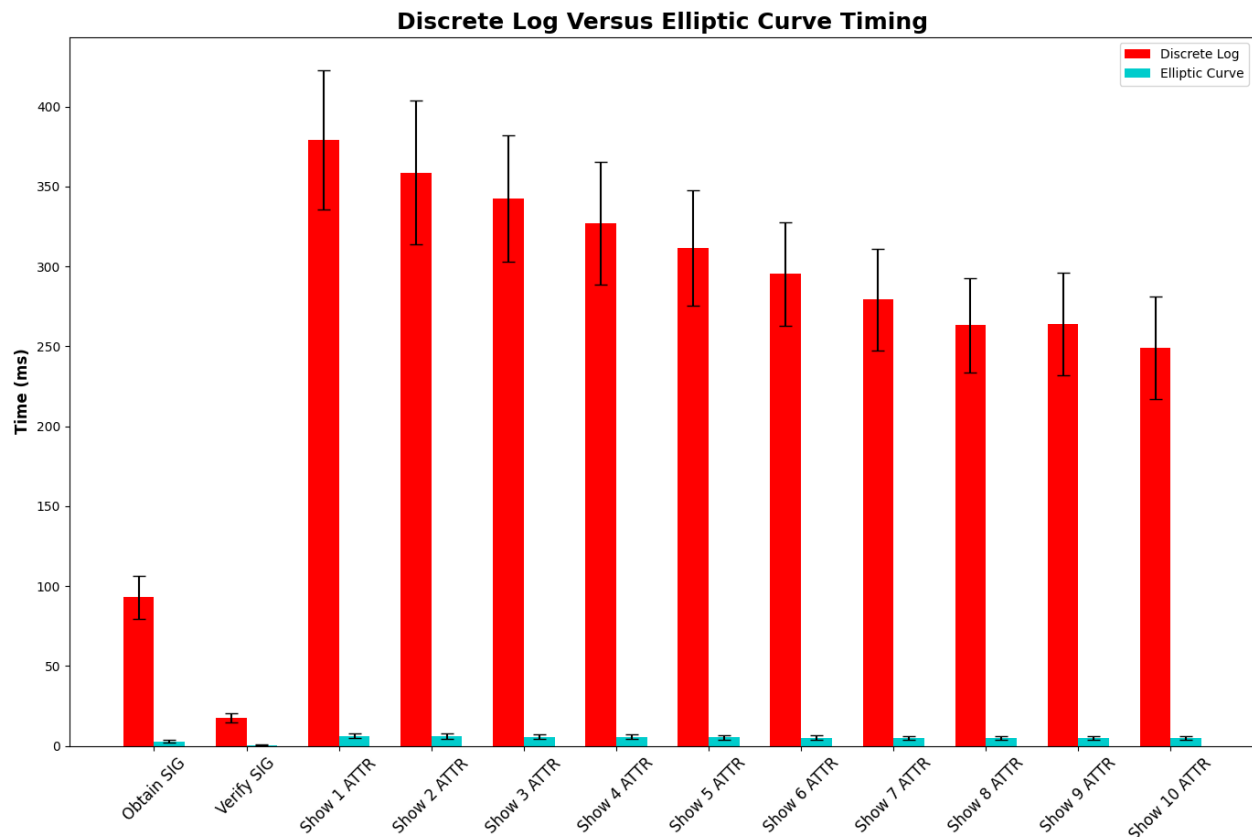
The two classes DLTest and ECTest were implemented to test the discrete log and elliptic curve implementations of the digital credentials protocols. For testing the discrete log version, the DLTest class instantiates a DLCA object, a DLVerifier object, and 1000 DLUser objects. Each DLUser object has the 10 attributes (GivenName, Surname, Birthday,CCEXpires) of one excel sheet row. Similarly, for testing the elliptic curve version, the ECTest class instantiates an ECCA object, an ECVerifier object, and 1000 ECUser objects. Each ECUser object has the 10 attributes (GivenName, Surname, Birthday,CCEXpires) of one excel sheet row.

For each of the DLUser or ECUser objects, twelve operations including obtaining signature from a CA, verifying the signature, showing 1 attribute, showing 2 attributes, showing 10 attributes were timed in nanoseconds using Java’s System.nanoTime(). The timing results were saved in two text files using the “write” method of the TextIO class.

A script (“digital_credentials_plot_test.py”) using Python’s pandas, matplotlib.pyplot, numpy, and scipy.stats libraries was used to read the two result files, produce a comparison bar chart, and carry out t tests.

Timing Results

Below is the comparison bar chart produced by the “plot_results” function of the Python script. The timings were done in nanoseconds by the Java test files, but were converted to milliseconds when producing the bar chart for better visualization:



An interesting observation from the timing graph is the inverse relationship between the number of attributes shown and the “show ATTR” time. This can be described by the fact that for each concealed or “not shown” attribute “ x_m ”, Alice computes “ $r_m = cx_m + w_m \text{ mod } q$ ” where “ c ” and “ w_m ” are random BigIntegers, and the verifier (Bob) computes $g_m^{r_m}$ (page 130 of the book). On the other hand, for each “shown attribute” “ x_i ”, The verifier computes $g_i^{cx_i}$. Since “ $cx_m + w_m$ ” is greater than “ x_m ”, the modular exponentiation time is longer for “not shown” attributes.

A screen recording of a speed test between discrete log and elliptic curve implementations can be found at <https://youtu.be/2MTRlwY2f8w> . In this test, 4 DLUser objects completed the 12 operations listed in the bar graph in 10 seconds. The same operations were completed for 237 ECUUser objects in the same time interval.

Statistical Testing

Despite the large timing differences between the two versions, 12 t tests were carried out to determine if the results are significantly different. As expected, the p value for all the t tests was 0.0. The screenshot below shows the output of the output produced by the “ttest_results” function of the Python script:

```
Obtain Signature Ttest_indResult(statistic=212.44148948656098, pvalue=0.0)
Verify Signature Ttest_indResult(statistic=190.8672787364354, pvalue=0.0)
Show 1 Attribute Ttest_indResult(statistic=270.7325530838158, pvalue=0.0)
Show 2 Attributes Ttest_indResult(statistic=246.68352961630998, pvalue=0.0)
Show 3 Attributes Ttest_indResult(statistic=268.04552022251625, pvalue=0.0)
Show 4 Attributes Ttest_indResult(statistic=265.5489185182791, pvalue=0.0)
Show 5 Attributes Ttest_indResult(statistic=266.7413067361027, pvalue=0.0)
Show 6 Attributes Ttest_indResult(statistic=283.50179008568796, pvalue=0.0)
Show 7 Attributes Ttest_indResult(statistic=271.0612772420153, pvalue=0.0)
Show 8 Attributes Ttest_indResult(statistic=276.09281916997406, pvalue=0.0)
Show 9 Attributes Ttest_indResult(statistic=253.82251749546953, pvalue=0.0)
Show 10 Attributes Ttest_indResult(statistic=239.73848955577785, pvalue=0.0)
```

