DSS Signature and Hash Chain Based Authentication for Bitcoin Transaction and Proof-of-Work in Blockchain

ECE409
3/28/18
Hao Nan Wei
20524122
hnwei@edu.uwaterloo.ca

Scope

This project uses the Digital Signature Standard (DSS) and hash chain-based authentication to implement a blockchain of length 3 with a simplified network structure.

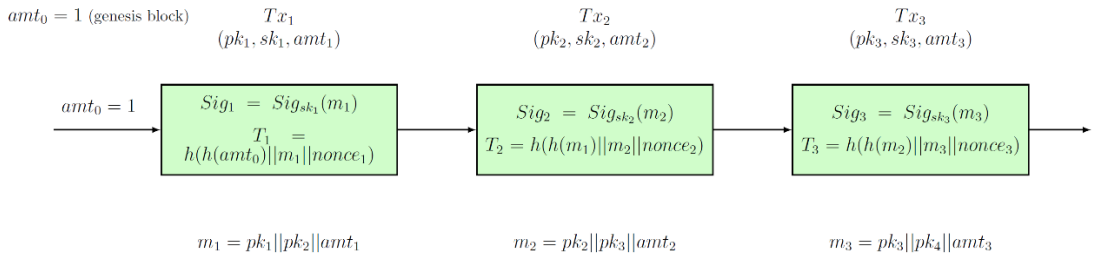

Figure 1: A block chain of length 3

System Parameters

To work with DSS, the following parameters need to be established:

- A prime number p, whose length is a multiple of 64 bits lying between 512 and 1024 bits
- A 160-bit prime number q of p – 1
- An element g ϵ GF(p) with order q
- A cryptographic hash function h
- A PRNG for creation of message key in DSS

We also assume the following parameters:

- $amt_0 = amt_1 = amt_2 = 1$ (integer)
- $nonce_i$, i = 1,2 are 128-bit
- SHA3-224 is the hash function, h, to be used
- DSS message keys are randomly chosen each time

Initial System Parameters

The following parameters were initially chosen (given in project):

- p = 168199388701209853920129085113302407023173962717160229197318545484823101018386724351964316301278642143567435810448472465887143222934545154943005714265124445244247988777471773193847131514083030740407543233616696550197643519458134465

700691569680905568000063025830089599260400096259430726498683087138415465107499 (1024 bits)

- q = 959452661475451209325433595634941112150003865821 (160 bits)
- g = 943891927763273985898453269803498145264338690934127823454309460592065688040051816008558259061429672718725043775029963900638123183435133537262152973355498432995364505138912569755859623649866375135353179362670798771770711847430626954864269888988371113567502852 (1024 bits)
- sk_1 = 432398415306986194693973996870836079581453988813 (159 bits)
- pk_1 = 493360183248080935347335488404117524857260585278296306689674805688547564165674962162949190519101486861866227068697023216644650947032473686465068210152903024809904501302806169292269172462551470632923017242976806834012586361821855991241311700775484507542940837288850755169851449449849200101384928972720692571601023 bits)

The initial data was preloaded into the program:

```
public static void loadData() {
    p = new BigInteger("168199388701209853920129085113302407023173962717160229197318545484823101018386724351964316301278642143567
    q = new BigInteger("959452661475451209325433595634941112150003865821");
    g = new BigInteger("943891927763273985898453269803498145264338690934127823454309460592065688040051816008558259061429672718725
    sk1 = new BigInteger("432398415306986194693973996870836079581453988813");
    pk1 = new BigInteger("493360183248080935347335488404117524857260585278296306689674805688547564165674962162949190519101486861
}
```

Verifying the Given Parameters

Before jumping into the implementation with the given parameters, the validity of each will be tested.

To test the validity, the following logic was used:

```
public static Boolean SystemParametersCheck() {
    Boolean result = true;
    BigInteger pMinus1 = p.subtract(BigInteger.ONE);
    BigInteger gOrder = g.modPow(q, p);

    if(!isPrime(p) || !isPrime(q)) {
        result = false;
    }

    if(p.bitLength() < 512 || p.bitLength() > 1024 || p.bitLength() % 64 != 0) {
        result = false;
    }

    if(pMinus1.mod(q).compareTo(BigInteger.ZERO) != 0 || q.bitLength() != 160) {
        result = false;
    }

    if(g.compareTo(BigInteger.ZERO) == -1 || g.compareTo(pMinus1) == 1 || gOrder.compareTo(BigInteger.ONE) != 0) {
        result = false;
    }

    return result;
}

public static Boolean isPrime(BigInteger num) {
    // with 99.22% certainty
    if(!num.isProbablePrime(7)) {
        return false;
    }
    return true;
}
```

Firstly, primality of p and q are checked by using the **isProbablyPrime(int certainty)** property of the **BigInteger** class in **math.java**. The function input is a measure of the uncertainty that the caller is willing to tolerate: if the call returns true the probability that this BigInteger is prime exceeds $(1 - 1/2^{certainty})$. The chosen value of 7 will determine if p or q is a prime with a certainty of 99.22%.

Then, the bit length of p is checked if it lays between 512 and 1024 bits, as well as if the bit length is a multiple of 64 bits.

The bit length of q is checked to be 160 bits, as well as if it's a prime factor of p. We only look to check its divisibility by p because the primality of q has been previously determined.

Lastly, we check to make sure g is an element of GF(p) (is in the range [0 <= g <= p – 1]) with order q.

If any of these conditions fail, the system parameter check returns false and throws an **IllegalArgumentException** in **main()**.

The given pair of sk_1, pk_1 was checked using this function:

```
public static Boolean ExistingKeysCheck(BigInteger sk, BigInteger pk) {
    Boolean result = true;
    BigInteger qMinus1 = q.subtract(new BigInteger("1"));

    if(sk.compareTo(BigInteger.ONE) == -1 || sk.compareTo(qMinus1) == 1) {
        result = false;
    }

    if(pk.compareTo(g.modPow(sk, p)) != 0) {
        result = false;
    }

    return result;
}
```

The **ExistingKeysCheck** function takes in two BigInteger keys and determines if sk (secret key) is larger than 0 but smaller than q (which is why it is chosen to be 159 bits). The pk (public key) is then checked to confirm that pk = g^{sk} mod p.

If any of these conditions fail, the function returns false and throws an **IllegalArgumentException** in **main()**.

The main flow of the parameter checking portion is as follows:

```
LoadData();
if(SystemParametersCheck()) {
    System.out.println("Valid system parameters (p, g, q)");
} else {
    throw new IllegalArgumentException("Invalid system parameters (p, g, q)");
}
if(ExistingKeysCheck(sk1, pk1)) {
    System.out.println("Valid keys (sk1, pk1)");
} else {
    throw new IllegalArgumentException("Invalid system keys (sk1, pk1)");
}
```

Generating New Keys

After confirmation that the initial system parameters are valid, the program goes on to generate two new sk_i-pk_i key pairs for i = 2,3 by making this call from **main()**:

```
GenerateNewKeys(new BigInteger("3"), new BigInteger("5"));
```

Which invokes this function:

```
public static void GenerateNewKeys(BigInteger seed1, BigInteger seed2) {
    sk2 = seed1;
    sk3 = seed2;
    pk2 = g.modPow(sk2, p);
    pk3 = g.modPow(sk3, p);
}
```

Note that the seed values passed into **GenerateNewKeys** are the actual secret key (sk) key values (in this case $sk_2 = 3$, $sk_3 = 5$). The corresponding public key (pk) is quickly calculated as g^{sk} mod p. The sk, pk values do not need to be returned as they are static and global variables in the program scope.

The **ExistingKeysCheck** function used to check the validity of the initial sk_1, pk_1 pair was also used to check the validity of the newly generated keys.

```
GenerateNewKeys(new BigInteger("3"), new BigInteger("5"));

System.out.println("Generated new ski-pki key-pairs");

if(ExistingKeysCheck(sk2, pk2)) {
    System.out.println("Valid keys (sk2, pk2)");
} else {
    throw new IllegalArgumentException("Invalid system keys (sk2, pk2)");
}

if(ExistingKeysCheck(sk3, pk3)) {
    System.out.println("Valid keys (sk3, pk3)");
} else {
    throw new IllegalArgumentException("Invalid system keys (sk3, pk3)");
}
```

Next, the keys k_i used to generate $Sig_{ski}(m_i)$ in DSS are picked. We pick two keys at random:

```
BigInteger key1 = new BigInteger("71");
BigInteger key2 = new BigInteger("31");
```

And determine their validity by calling the **CheckKey** function:

```
if(CheckKey(key1, q, p)) {
    System.out.println("Key1 OK");
} else {
    throw new IllegalArgumentException("Invalid DSS key k1");
}
if(CheckKey(key2, q, p)) {
    System.out.println("Key2 OK");
} else {
    throw new IllegalArgumentException("Invalid DSS key k2");
}
```

The criteria on k is that it falls in the range [0, q] and it has an inverse modulo q, which leads to the following logic:

```
public static Boolean CheckKey(BigInteger key, BigInteger q) {
    Boolean result = true;
    try {
        key.modInverse(q);
        if(key.compareTo(BigInteger.ONE) == -1 || key.compareTo(q) != -1) {
            result = false;
        }
    } catch(ArithmeticException ex) {
        result = false;
        System.out.println("Key does not have an inverse modulo q");
    }
    return result;
}
```

If an inverse for k module q does not exist, **BigInteger.modInverse(BigInteger modulo)** will throw an **ArithmeticException**.

Generating Messages

From the main program, the messages are generated by making these calls:

```
BigInteger message1 = GenerateMessage(pk1, pk2, BigInteger.ONE, "FULL");
BigInteger message2 = GenerateMessage(pk2, pk3, BigInteger.ONE, "FULL");
BigInteger simpleMessage1 = GenerateMessage(pk1, pk2, BigInteger.ONE, "SIMPLE");
BigInteger simpleMessage2 = GenerateMessage(pk2, pk3, BigInteger.ONE, "SIMPLE");
```

Given the same parameters, there are two types of messages that are generated and used for this DSS implementation: **FULL** and **SHORT**.

The **FULL** type of message just performs a full bit-wise concatenation of the two pk's and an 1-bit integer amount value (in this case, 1):

```
public static BigInteger GenerateMessage(BigInteger pk1, BigInteger pk2, BigInteger amt, String type) {
    try {
        byte[] bytePK1 = pk1.toByteArray();
        byte[] bytePK2 = pk2.toByteArray();
        byte[] byteAmt = amt.toByteArray();

        if(type.equals("FULL")) {
            byte[] concatByte = new byte[bytePK1.length + bytePK2.length + byteAmt.length];
            System.arraycopy(bytePK1, 0, concatByte, 0, bytePK1.length);
            System.arraycopy(bytePK2, 0, concatByte, bytePK1.length, bytePK2.length);
            System.arraycopy(byteAmt, 0, concatByte, bytePK1.length + bytePK2.length, byteAmt.length);

            return new BigInteger(1, concatByte);
```

The **BigInteger** values are converted to their byte arrays, and then the code performs regular **arraycopy** operations to create the concatenated byte array and returns it upon completion.

The **SHORT** type of message concatenates the 399 MSBs from pk_1 and pk_2 together, extends the amount value to 2 bits (still maintains integer value of 1) and appends the 2-bit amount value to the end of the concatenated byte array and returns the 800-bit result.

```
} else if (type.equals("SIMPLE")) {
    byte[] concatByte = new byte[100];

    byte lastBytePK1 = bytePK1[49];
    byte firstBytePK2 = bytePK2[0];
    byte firstBitPK2 = (byte) ((firstBytePK2 >> 7) & 1);

    if(firstBitPK2 == 1) {
        lastBytePK1 = (byte) (lastBytePK1 | (1));
    } else {
        lastBytePK1 = (byte) (lastBytePK1 & ~(1));
    }
    bytePK1[49] = lastBytePK1;

    BigInteger shiftedPK2 = pk2.shiftLeft(1);
    byte[] shiftedBytePK2 = shiftedPK2.toByteArray();

    byte lastBytePK2 = shiftedBytePK2[49];
    lastBytePK2 = (byte) (lastBytePK2 | (1));
    lastBytePK2 = (byte) (lastBytePK2 & ~(1 << 1 ));
    shiftedBytePK2[49] = lastBytePK2;

    System.arraycopy(bytePK1, 0, concatByte, 0, 50);
    System.arraycopy(shiftedBytePK2, 0, concatByte, 50, 50);

    return new BigInteger(1, concatByte);
```

The logic in this section requires more explanation. Note that **BigInteger.toByteArray()** returns byte in Big-Endian, and bit positions start at 1 in this discussion.

First, we take the most significant 400 bits (first 50 bytes) from pk_1. Then, we grab the first (most significant) byte of pk_2 and determine if the MSB of pk_2 is a 1 or 0 by bit-shifting the MSB to the LSB position in the byte and performing an **AND** operation with 1 (binary 0000 0001).

If the MSB of pk_2 is a 1, the 400^{th} bit of pk_1 becomes a 1, and if the MSB of pk_2 is a 0, the 400^{th} bit of pk_1 becomes a 0. This is because in the concatenated byte array, the 399^{th} bit is the last bit from pk_1 and the 400^{th} bit is the first but from pk_2.

The whole pk_2 is bit-wise shifted left, as the previous MSB has been included in the (last bit of byte 50) 400^{th} bit position of pk_1 and so all the successive bits should follow. This time, the 399^{th} bit of pk_2 is set to "0" and the 400^{th} bit of pk_2 is set to "1" to represent the 2-bit representation of the amount of 1 (end of the concatenated array).

Finally, the two arrays are concatenated together and returned as an 800-bit shortened message. The use of these shortened messages will come later when calculating the transactions.

Generating Digital Signatures and Verification

In the main program, signing digital signatures and verification is handled through the following calls:

```
BigInteger[] Sig1 = dssModule.Sign(sk1, message1, p, q, g, key1);
System.out.println("Sig1 created -> (" + Sig1[0] + ", " + Sig1[1] + ")");

Boolean verified = dssModule.Verify(pk1, message1, p, q, g, Sig1);
```

The signature for the second transaction is generated and verified similarly:

```
BigInteger[] Sig2 = dssModule.Sign(sk2, message2, p, q, g, key2);
System.out.println("Sig2 created -> (" + Sig2[0] + ", " + Sig2[1] + ")");

verified = dssModule.Verify(pk2, message2, p, q, g, Sig2);
```

The signing and verification calls require the use of the **DSSmodule.java** class, which simply contains two methods: **Sign** and **Verify.**

```
public class DSSmodule {
    public BigInteger[] Sign(BigInteger sk, BigInteger message, BigInteger p, BigInteger q, BigInteger g, BigInteger key) {

    public Boolean Verify(BigInteger pk, BigInteger message, BigInteger p, BigInteger q, BigInteger g, BigInteger[] digitalSignature) {
}
```

First, we discuss the **Sign** method:

```
public BigInteger[] Sign(BigInteger sk, BigInteger message, BigInteger p, BigInteger q, BigInteger g, BigInteger key) {
    BigInteger r;
    BigInteger s;

    r = g.modPow(key, p).mod(q);
    //System.out.println("r -> " + r);

    SHA3.DigestSHA3 digestSHA3 = new SHA3.Digest224();
    byte[] digestedMessage = digestSHA3.digest(message.toByteArray());
    BigInteger intDigestedMessage = new BigInteger(1, digestedMessage).mod(q);

    s = key.modInverse(q).multiply(intDigestedMessage.subtract(sk.multiply(r))).mod(q);
    //System.out.println("s -> " + s);

    BigInteger[] digitalSignature = new BigInteger[]{r, s};
    return digitalSignature;
}
```

Right away, we can calculate the value of r due to the build-in operatons provided by the **BigInteger** class. The g value is raised to the value of the key, followed by a modulo p and then modulo q. After calculating the r value, the open source java library **BouncyCastle** was used as the provider for the SHA3-224 hashing function. The message is converted to a byte array and digested, and the digested message is converted back to **BigInteger** modulo q. The signing equation (figured below) is rearranged and s is isolated, giving the equation for s in the code.

$$h(m) \equiv xr + ks \bmod q$$

After s is calculated, the digital signature pair (r, s) is returned.

Secondly, we discuss the **Verify** method:

```
public Boolean Verify(BigInteger pk, BigInteger message, BigInteger p, BigInteger q, BigInteger g, BigInteger[] digitalSignature) {
    Boolean result = true;
    BigInteger u;
    BigInteger v;
    BigInteger w;

    if(digitalSignature[0].compareTo(BigInteger.ZERO) != 1 || digitalSignature[0].compareTo(q) != -1) {
        result = false;
    }

    if(digitalSignature[1].compareTo(BigInteger.ZERO) != 1 || digitalSignature[0].compareTo(q) != -1) {
        result = false;
    }

    SHA3.DigestSHA3 digestSHA3 = new SHA3.Digest224();
    byte[] digestedMessage = digestSHA3.digest(message.toByteArray());
    BigInteger intDigestedMessage = new BigInteger(1, digestedMessage).mod(q);

    BigInteger inverseS = digitalSignature[1].modInverse(q);

    u = intDigestedMessage.multiply(inverseS).mod(q);
    v = digitalSignature[0].negate().multiply(inverseS).mod(q);
    w = g.modPow(u, p).multiply(pk.modPow(v, p)).mod(p).mod(q);
    //System.out.println("w -> " + w);

    if(w.compareTo(digitalSignature[0]) != 0) {
        return false;
    }

    return result;
}
```

The first thing that takes place is to check that r and s are within the range [0, q], and reject if not satisfied. Then, the message is digested using the **BouncyCastle** SHA3-224 hashing function to give us h(m). The method then takes the inverse of the s given and uses these values to calculate for u, v, and finally w using these relationships:

$$u = h(m)s^{-1} \bmod q, \text{ and } v = -rs^{-1} \bmod q$$

$$w = g^u y^v$$

At the end of the method, it checks to see if the calculated w is equivalent to the r value held within the digital signature pair. If they are equivalent, then the digital signature is verified and the function returns true. If they are not equivalent the verification fails and the function returns false.

Generating Hash-Chain Based Transactions

In the scope of **main()**, transactions are generated after the digital signature has been verified by calling the **Transact** method from the **TransactionModule.java** class.

```
BigInteger[] Sig1 = dssModule.Sign(sk1, message1, p, q, g, key1);
System.out.println("Sig1 created -> (" + Sig1[0] + ", " + Sig1[1] + ")");

Boolean verified = dssModule.Verify(pk1, message1, p, q, g, Sig1);
if(verified) {
    System.out.println("Sig1 verified");
    BigInteger T1 = transactModule.Transact(BigInteger.ONE, simpleMessage1, BigInteger.ONE);
    if(T1.compareTo(BigInteger.ZERO) == 0) {
        System.out.println("No preimage found for T1...");
    }
}
```

The second transaction is generated similarly:

```
BigInteger[] Sig2 = dssModule.Sign(sk2, message2, p, q, g, key2);
System.out.println("Sig2 created -> (" + Sig2[0] + ", " + Sig2[1] + ")");

verified = dssModule.Verify(pk2, message2, p, q, g, Sig2);
if(verified) {
    System.out.println("Sig2 verified");
    BigInteger T2 = transactModule.Transact(message1, simpleMessage2, BigInteger.ONE);
    if(T2.compareTo(BigInteger.ZERO) == 0) {
        System.out.println("No preimage found for T2...");
    }
}
```

The **TransactionModule.java** class contains two methods: **Transact** and **CheckPreImage**:

```
public class TransactionModule {
    public BigInteger Transact(BigInteger hashMessage, BigInteger simpleMessage,  BigInteger amt) {

    public Boolean CheckPreImage(byte[] digestedM1) {
}
```

First, let's look at **Transact**. **Transact** is the method which will construct the transaction hash following the relationship:

$$T_1 = h(m_1'), m_1' = h(amt_0)||m_1||nonce_1$$

$$m_1 = pk_1||pk_2||amt_1$$

For T_i where i = 2, 3, …, amt_0 is replaced by m_{i-1}.

The simple messages generated earlier will now be passed into **Transact** as **simpleMessage** and is 800 bits. The **hashMessage** is the message that is to be hashed (into 224 bits) and then concatenated with the shortened message and the nonce (128 bits) to produce a perfectly digestible 1152-bit block for the SHA3-244 hash.

```
public BigInteger Transact(BigInteger hashMessage, BigInteger simpleMessage,  BigInteger amt) {

    SHA3.DigestSHA3 digestSHA3 = new SHA3.Digest224();
    byte[] digestedMessage = digestSHA3.digest(hashMessage.toByteArray());

    byte[] byteSimpleMessage = simpleMessage.toByteArray();
    byte[] byteSimpleMessagePadded = new byte[100];

    for(int b = byteSimpleMessage.length - 1; b >= 0; b--) {
        byteSimpleMessagePadded[byteSimpleMessagePadded.length - 1 - b] = byteSimpleMessage[byteSimpleMessage.length - 1 - b];
    }

    BigInteger nonce;
    BigInteger T1 = BigInteger.ZERO;

    byte[] byteNoncePadded = new byte[16];
    for(int i = 0; i < Math.pow(2, 24); i++) {
        nonce = new BigInteger(Integer.toString(i));
        byte[] byteNonce = nonce.toByteArray();
        for(int j = byteNonce.length - 1; j >= 0; j--) {
            byteNoncePadded[byteNoncePadded.length - 1 - j] = byteNonce[byteNonce.length - 1 - j];
        }
        byte[] m1 = new byte[1152];

        System.arraycopy(digestedMessage, 0, m1, 0, digestedMessage.length);
        System.arraycopy(byteSimpleMessagePadded, 0, m1, digestedMessage.length, byteSimpleMessagePadded.length);
        System.arraycopy(byteNoncePadded, 0, m1, digestedMessage.length +  byteSimpleMessagePadded.length, byteNoncePadded.length);

        byte[] digestedM1 = digestSHA3.digest(m1);

        if(CheckPreImage(digestedM1)) {
            T1 = new BigInteger(1, digestedM1);
            break;
        }
    }
    return T1;
}
```

It should also be noted that because the original byte sizes of the digested SHA3-224 messages were lost when converting to **BigInteger** (removes leading 0's in byte array during conversion), the padding had to be manually added before concatenation.

Now let's look at **CheckPreImage**. **CheckPreImage** intakes a transaction in byte array format and determines if the pre-image is valid.

```
public Boolean CheckPreImage(byte[] digestedM1) {
    for(int i = 0; i < 3; i++) {
        if(digestedM1[i] != 0) {
            return false;
        }
    }
    System.out.println("FOUND A PREIMAGE! -> " + Hex.toHexString(digestedM1));
    return true;
}
```

The requirements for the Proof-of-Work is to find a pre-image of h (SHA3-224) such that

$$T_1 = \underbrace{00\cdots0}_{32}\underbrace{**\cdots*}_{192}$$
$$T_2 = \underbrace{00\cdots0}_{32}\underbrace{**\cdots*}_{192}$$

where * means any value. However, for Java, it is only required to find a pre-image with only 24 bits of leading 0's in the transaction hash. Thus, the function iterates over the first 3 (most significant) bytes of

the transaction hash and determines if they are all equivalent to 0. If they are equal, the pre-image has been found and the function returns true, returns false otherwise.

The function loops iterates through from 0 to 2^{24} - 1 possible nonces to try and generate a transaction with the correct pre-image instead of looping through 2^{128}. This is because if all other 1128 bits of the 1152-bit message remain constant (104 bits of which belong to the nonce and are all 0's), one of the 2^{24} nonce values should map to a transaction hash with 24 leading 0's. Thus, the time complexity for the search of a valid pre-image is 2^{24}.

Final Results

Valid system parameters (p, g, q)

Valid keys (sk1, pk1)

Generated new ski-pki key-pairs

Valid keys (sk2, pk2)

sk2 -> 3

pk2 ->
730339767816565250247034519050512331059942773272647736415501065418864001586564229352879883464129819759496063442977337576166827891042746642630709501525211851961948959665002983412830232794027569079362823786545627690461669486276904451920458414546759532909491886400278723872563569251771479208833292193756816744 5

Valid keys (sk3, pk3)

sk3 -> 5

pk3 ->
163818460596722533180091987452979921608224681959175850464820032241124518312973428029398467132069237067184250931481147837323105554944679787289206815738226901711800714426579244849853877189315707643183775178485422288323281910386602778780422958203658009093510525532489841427342810905732009751935166961733819835818

Key1 OK -> 71

Key2 OK -> 31

message1 created ->
227049014896509248103748652299139718229851100902067111745652401759948694456334481066963572957087665470374360368386667957632956459731199455524017096556428262363106513444477609625124159571238643847475863076462946987193426125156520448701371679227972280400743099275205071422176246751701156223967038298213301668934278093315371658369741032945900679549530918088213291647217946151900077700953045689481371623071969130042997501450115004678461362876138901596916927448323145527783844372597611820789879324589515861644333383388995441699302611841663341531894986332006844670472551783158147860164390040744525870008945936067496925030528 1

message2 created ->
860439698945328494456102725784583476225662645200774821002860137234002841527707515725086855690272182487254069906184357511947256084051460432652623230419286282474048959571900852453585921833290838386725854092369246450608822955691876383588940584414574060949067034495657790738994790423417638485887809381844212429904931422078246627214015240515976683852749028527085864908232411342823828831512618663003959068145430469873032013139424431815018731282919323680055891339321412760016385737306062257596702497497301

125395880064332352479769364923890643097428290347514112248219787958075584626381802427244072135257917210290740014 01386986121729

simpleMessage1 created ->
182997462620805512192134781759659308766615703119063892013784864035309079219377404900699253306111923827728107732389075398328424734583765557203028424757054012306283064743799615516025133411925942794792134254711551629385571694005970451951398532 9

simpleMessage2 created ->
270898075887682707334004652861403724879997598527472247209325788643286886040555471741189380994013116720602317814511700572507642485024126925678825986292931616932363404742276647954498168312052677323390595389430756943549502616012208114750322845

r -> 269105770324817675424681515877209589459040358127

s -> 248220774819800795052551757812571154930403145514

Sig1 created -> (269105770324817675424681515877209589459040358127, 248220774819800795052551757812571154930403145514)

u -> 261801281528416355269182114176435617254747139 87

v -> 306719378675836314956942304507162090988288505439

w -> 269105770324817675424681515877209589459040358127

Sig1 verified

Transacting T1...

FOUND A PREIMAGE! -> 000000206958bfa41d3a27817773af7ee9d8aa25d8b4368daee99745

valid nonce found -> 00000000000000000000000000a280dd

r -> 561474052182132145133696462779627378633253300757

s -> 821674223850197685893089939876292785695495986272

Sig2 created -> (561474052182132145133696462779627378633253300757, 821674223850197685893089939876292785695495986272)

u -> 171358809608680385552810448769269715861942677 51

v -> 633923147330011460031862048797651750904604487974

w -> 561474052182132145133696462779627378633253300757

Sig2 verified

Transacting T2...

FOUND A PREIMAGE! -> 000000cf4243b3d4f79bd037b8dbe6b85f383b75e8424f3606da7661

valid nonce found -> 00000000000000000000000002eb63b

Full Source Code

The following pages contain the full source code for the DSS length-3 blockchain implementation. The first class **BlockchainDSS.java** handles the setup, parameter checking, and main execution of the program. The **DSSmodule.java** class is a standalone module which generates a digital signature for a transaction and also verifies a digital signature for a transaction. **TransactionModule.java** is the module that deals with finding a pre-image of h in computing transactions T_1 and T_2 such that the most 3 significant bytes are all 0's (24 bits for Java implementation).

Some print statements have been commented out; uncomment those to see full results during execution.

```
import java.math.BigInteger;

import org.bouncycastle.util.encoders.Hex;

public class BlockchainDSS {
	static BigInteger p;
	static BigInteger q;
	static BigInteger g;
	static BigInteger sk1;
	static BigInteger pk1;
	static BigInteger sk2;
	static BigInteger pk2;
	static BigInteger sk3;
	static BigInteger pk3;
	static DSSmodule dssModule = new DSSmodule();
	static TransactionModule transactModule = new TransactionModule();

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		try {
			loadData();
			if(SystemParametersCheck()) {
				System.out.println("Valid system parameters (p, g, q)");
			} else {
				throw new IllegalArgumentException("Invalid system parameters (p, g, q)");
			}
			if(ExistingKeysCheck(sk1, pk1)) {
				System.out.println("Valid keys (sk1, pk1)");
			} else {
				throw new IllegalArgumentException("Invalid system keys (sk1, pk1)");
			}

			GenerateNewKeys(new BigInteger("3"), new BigInteger("5"));

			System.out.println("Generated new ski-pki key-pairs");

			if(ExistingKeysCheck(sk2, pk2)) {
				System.out.println("Valid keys (sk2, pk2)");
			} else {
				throw new IllegalArgumentException("Invalid system keys (sk2, pk2)");
			}

			if(ExistingKeysCheck(sk3, pk3)) {
				System.out.println("Valid keys (sk3, pk3)");
			} else {
				throw new IllegalArgumentException("Invalid system keys (sk3, pk3)");
			}

			/*System.out.println("sk2 -> " + sk2);
			System.out.println("pk2 -> " + pk2);
			System.out.println("sk3 -> " + sk3);
			System.out.println("pk3 -> " + pk3);*/

			BigInteger key1 = new BigInteger("71");
			BigInteger key2 = new BigInteger("31");
```

```
                if(CheckKey(key1, q)) {
                        System.out.println("Key1 OK");
                } else {
                        throw new IllegalArgumentException("Invalid DSS key k1");
                }
                if(CheckKey(key2, q)) {
                        System.out.println("Key2 OK");
                } else {
                        throw new IllegalArgumentException("Invalid DSS key k2");
                }

                BigInteger message1 = GenerateMessage(pk1, pk2, BigInteger.ONE, "FULL");
                BigInteger message2 = GenerateMessage(pk2, pk3, BigInteger.ONE, "FULL");
                BigInteger simpleMessage1 = GenerateMessage(pk1, pk2, BigInteger.ONE, "SIMPLE");
                BigInteger simpleMessage2 = GenerateMessage(pk2, pk3, BigInteger.ONE, "SIMPLE");

                BigInteger[] Sig1 = dssModule.Sign(sk1, message1, p, q, g, key1);
                System.out.println("Sig1 created -> (" + Sig1[0] + ", " + Sig1[1] + ")");

                Boolean verified = dssModule.Verify(pk1, message1, p, q, g, Sig1);
                if(verified) {
                        System.out.println("Sig1 verified");
                        BigInteger T1 = transactModule.Transact(BigInteger.ONE, simpleMessage1,
                                BigInteger.ONE);
                        if(T1.compareTo(BigInteger.ZERO) == 0) {
                                System.out.println("No preimage found for T1...");
                        }
                }

                BigInteger[] Sig2 = dssModule.Sign(sk2, message2, p, q, g, key2);
                System.out.println("Sig2 created -> (" + Sig2[0] + ", " + Sig2[1] + ")");

                verified = dssModule.Verify(pk2, message2, p, q, g, Sig2);
                if(verified) {
                        System.out.println("Sig2 verified");
                        BigInteger T2 = transactModule.Transact(message1, simpleMessage2,
                                BigInteger.ONE);
                        if(T2.compareTo(BigInteger.ZERO) == 0) {
                                System.out.println("No preimage found for T2...");
                        }
                }

        } catch (IllegalArgumentException ex) {
                System.out.println(ex.toString());
        }
}

public static Boolean CheckKey(BigInteger key, BigInteger q) {
        Boolean result = true;
        try {
                key.modInverse(q);
                if(key.compareTo(BigInteger.ONE) == -1 || key.compareTo(q) != -1) {
                        result = false;
                }
        } catch(ArithmeticException ex) {
                result = false;
```

```
                System.out.println("Key does not have an inverse modulo q");
        }
        return result;
    }

    public static BigInteger GenerateMessage(BigInteger pk1, BigInteger pk2, BigInteger amt, String
    type) {
        try {
                byte[] bytePK1 = pk1.toByteArray();
                byte[] bytePK2 = pk2.toByteArray();
                byte[] byteAmt = amt.toByteArray();

                if(type.equals("FULL")) {
                        byte[] concatByte = new byte[bytePK1.length + bytePK2.length +
                                byteAmt.length];
                        System.arraycopy(bytePK1, 0, concatByte, 0, bytePK1.length);
                        System.arraycopy(bytePK2, 0, concatByte, bytePK1.length, bytePK2.length);
                        System.arraycopy(byteAmt, 0, concatByte, bytePK1.length + bytePK2.length,
                                byteAmt.length);

                        return new BigInteger(1, concatByte);
                } else if (type.equals("SIMPLE")) {
                        byte[] concatByte = new byte[100];

                        byte lastBytePK1 = bytePK1[49];
                        byte firstBytePK2 = bytePK2[0];
                        byte firstBitPK2 = (byte) ((firstBytePK2 >> 7) & 1);

                        if(firstBitPK2 == 1) {
                                lastBytePK1 = (byte) (lastBytePK1 | (1));
                        } else {
                                lastBytePK1 = (byte) (lastBytePK1 & ~(1));
                        }
                        bytePK1[49] = lastBytePK1;

                        BigInteger shiftedPK2 = pk2.shiftLeft(1);
                        byte[] shiftedBytePK2 = shiftedPK2.toByteArray();

                        byte lastBytePK2 = shiftedBytePK2[49];
                        lastBytePK2 = (byte) (lastBytePK2 | (1));
                        lastBytePK2 = (byte) (lastBytePK2 & ~(1 << 1 ));
                        shiftedBytePK2[49] = lastBytePK2;

                        System.arraycopy(bytePK1, 0, concatByte, 0, 50);
                        System.arraycopy(shiftedBytePK2, 0, concatByte, 50, 50);

                        return new BigInteger(1, concatByte);
                } else {
                        throw new IllegalArgumentException("Invalid message type for
                                GenerateMessage.");
                }
        } catch(IllegalArgumentException ex) {
                System.out.println(ex.toString());
                return new BigInteger("-1");
        }
    }
```

```
public static void GenerateNewKeys(BigInteger seed1, BigInteger seed2) {
	sk2 = seed1;
	sk3 = seed2;
	pk2 = g.modPow(sk2, p);
	pk3 = g.modPow(sk3, p);
}

public static Boolean ExistingKeysCheck(BigInteger sk, BigInteger pk) {
	Boolean result = true;
	BigInteger qMinus1 = q.subtract(new BigInteger("1"));

	if(sk.compareTo(BigInteger.ONE) == -1 || sk.compareTo(qMinus1) == 1) {
		result = false;
	}

	if(pk.compareTo(g.modPow(sk, p)) != 0) {
		result = false;
	}

	return result;
}

public static Boolean SystemParametersCheck() {
	Boolean result = true;
	BigInteger pMinus1 = p.subtract(BigInteger.ONE);
	BigInteger gOrder = g.modPow(q, p);

	if(!isPrime(p) || !isPrime(q)) {
		result = false;
	}

	if(p.bitLength() < 512 || p.bitLength() > 1024 || p.bitLength() % 64 != 0) {
		result = false;
	}

	if(pMinus1.mod(q).compareTo(BigInteger.ZERO) != 0 || q.bitLength() != 160) {
		result = false;
	}

	if(g.compareTo(BigInteger.ZERO) == -1 || g.compareTo(pMinus1) == 1 ||
		gOrder.compareTo(BigInteger.ONE) != 0) {
		result = false;
	}

	return result;
}

public static Boolean isPrime(BigInteger num) {
	// with 99.22% certainty
	if(!num.isProbablePrime(7)) {
		return false;
	}
	return true;
}
```

```
        public static void loadData() {
                p = new
                        BigInteger("168199388701209853920129085113302407023173962717160229197318545484823101
                        018386724351964316301278642143567435810448472465887143222934545154943005714265124445
                        244247988777471773193847131514083030740407543233616696550197643519458134465700691569
                        680905568000063025830089599260400096259430726498683087138415465107499");
                q = new BigInteger("95945266147545120932543359563494111215000386582 1");
                g = new
                        BigInteger("943891927763273985898453269803498145264338690934127823454309460592065688
                        040051816008558259061429672718725483758777389498758125404332234449684613507894613850
                        437750299639006381231834351335372621529733554984329953645051389125697558596236498663
                        751353531793626707987717707118474306269548642698889883711135675028 52");
                sk1 = new BigInteger("432398415306986194693973996870836079581453988813");
                pk1 = new
                        BigInteger("493360183248080935347335488404117524857260585278296306689674805688547564
                        165674962162949190519101486861866227068697023216644650947032473686465068210152903024
                        809904501302806169292269172462551470632923017242976806834012586361821855991241311700
                        775484507542940837288850755169851449449849200101384928972720692571 60");
        }
}
```

```
import java.math.BigInteger;
import org.bouncycastle.jcajce.provider.digest.SHA3;
import org.bouncycastle.util.encoders.Hex;

public class DSSmodule {
	public BigInteger[] Sign(BigInteger sk, BigInteger message, BigInteger p, BigInteger q, BigInteger
	g, BigInteger key) {
		BigInteger r;
		BigInteger s;

		r = g.modPow(key, p).mod(q);
		//System.out.println("r -> " + r);

		SHA3.DigestSHA3 digestSHA3 = new SHA3.Digest224();
		byte[] digestedMessage = digestSHA3.digest(message.toByteArray());
		BigInteger intDigestedMessage = new BigInteger(1, digestedMessage).mod(q);

		s = key.modInverse(q).multiply(intDigestedMessage.subtract(sk.multiply(r))).mod(q);
		//System.out.println("s -> " + s);

		BigInteger[] digitalSignature = new BigInteger[]{r, s};
		return digitalSignature;
	}

	public Boolean Verify(BigInteger pk, BigInteger message, BigInteger p, BigInteger q, BigInteger g,
	BigInteger[] digitalSignature) {
		Boolean result = true;
		BigInteger u;
		BigInteger v;
		BigInteger w;

		if(digitalSignature[0].compareTo(BigInteger.ZERO) != 1 || digitalSignature[0].compareTo(q)
		!= -1) {
			result = false;
		}
		if(digitalSignature[1].compareTo(BigInteger.ZERO) != 1 || digitalSignature[0].compareTo(q)
		!= -1) {
			result = false;
		}

		SHA3.DigestSHA3 digestSHA3 = new SHA3.Digest224();
		byte[] digestedMessage = digestSHA3.digest(message.toByteArray());
		BigInteger intDigestedMessage = new BigInteger(1, digestedMessage).mod(q);
		BigInteger inverseS = digitalSignature[1].modInverse(q);

		u = intDigestedMessage.multiply(inverseS).mod(q);
		v = digitalSignature[0].negate().multiply(inverseS).mod(q);
		w = g.modPow(u, p).multiply(pk.modPow(v, p)).mod(p).mod(q);
		//System.out.println("w -> " + w);

		if(w.compareTo(digitalSignature[0]) != 0) {
			result = false;
		}
		return result;
	}
}
```

```
import java.math.BigInteger;
import org.bouncycastle.jcajce.provider.digest.SHA3;
import org.bouncycastle.util.encoders.Hex;

public class TransactionModule {
	public BigInteger Transact(BigInteger hashMessage, BigInteger simpleMessage,  BigInteger amt) {
		SHA3.DigestSHA3 digestSHA3 = new SHA3.Digest224();
		byte[] digestedMessage = digestSHA3.digest(hashMessage.toByteArray());

		byte[] byteSimpleMessage = simpleMessage.toByteArray();
		byte[] byteSimpleMessagePadded = new byte[100];

		for(int b = byteSimpleMessage.length - 1; b >= 0; b--) {
			byteSimpleMessagePadded[byteSimpleMessagePadded.length - 1 - b] =
				byteSimpleMessage[byteSimpleMessage.length - 1 - b];
		}

		BigInteger nonce;
		BigInteger T1 = BigInteger.ZERO;

		byte[] byteNoncePadded = new byte[16];
		for(int i = 0; i < Math.pow(2, 24); i++) {
			nonce = new BigInteger(Integer.toString(i));
			byte[] byteNonce = nonce.toByteArray();
			for(int j = byteNonce.length - 1; j >= 0; j--) {
				byteNoncePadded[byteNoncePadded.length - 1 - j] = byteNonce[byteNonce.length
					- 1 - j];
			}
			byte[] m1 = new byte[1152];

			System.arraycopy(digestedMessage, 0, m1, 0, digestedMessage.length);
			System.arraycopy(byteSimpleMessagePadded, 0, m1, digestedMessage.length,
				byteSimpleMessagePadded.length);
			System.arraycopy(byteNoncePadded, 0, m1, digestedMessage.length +
				byteSimpleMessagePadded.length, byteNoncePadded.length);

			byte[] digestedM1 = digestSHA3.digest(m1);

			if(CheckPreImage(digestedM1)) {
				T1 = new BigInteger(1, digestedM1);
				break;
			}
		}
		return T1;
	}

	public Boolean CheckPreImage(byte[] digestedM1) {
		for(int i = 0; i < 3; i++) {
			if(digestedM1[i] != 0) {
				return false;
			}
		}
		System.out.println("FOUND A PREIMAGE! -> " + Hex.toHexString(digestedM1));
		return true;
	}
}
```