Distributed Control Rights Management Signature Verification Program

1. Requirement

Public key cryptography is widely used for the account system of the blockchain and the ownership of the digital assets. Private keys are used to sign messages authorizing crypto-asset transactions.

In the case of bitcoin, the private key is a 256-bit number. Private keys are often stored in Wallet Import Format (WIF), which is obtained by calling the SHA-256 hash function on the private key twice, then adding a checksum to the private key, then converting the private key to base58. The public key is generated by the calling the ECDSA algorithm of the secp256k1 specification on the private key. The bitcoin address is generated by the public key via hash functions (RIPEMD and SHA).

In the process of message signing, the signature is generated by using the hash value of the private key with the message according to the ECDSA algorithm of secp256k1 specification. The signature, message, and public key are spliced together according to a certain structure and broadcast to the whole network. The node first uses the public key and signature to check the message, and then the message is processed after the verification is passed.

Compared with the way the public and private key pairs are generated and used in bitcoin, FUSION's Distributed Control Rights Management technology differs in the following aspects:

1. The private key is generated by many FUSION nodes through distributed computation. Public key generation and the message signature are distributed to the nodes. The generated public key and signature can be broadcast to the target blockchain and verified.
2. The private key or enough data to reconstruct it is never exposed during the generation and use of the private key.

FUSION's Distributed Control Rights Management technology uses distributed computing, homomorphic encryption, zero knowledge proofs and other cryptographic techniques to replace the complete private key with private key fragments. Distributed generation of a valid public key, address and transaction signature in accordance with the targeted blockchain ECDSA algorithm and specification is implemented. With this technology, the management and operation of the crypto-assets on all different blockchains may be realized on FUSION. DCRM is the core technology facilitating the management of assets across heterogeneous blockchains.

2. Description of the verification

2.1 Function of distributed generation

The verification program demonstrates the verification process of the FUSION Distributed Control Rights Management, including:

1. The private key fragments are generated and stored separately by multiple nodes. The public key is generated through distributed computation with the private key fragments.
2. It implements the signature of the designated messages by multiple nodes based on distributed computation.
3. There is no passing of the private key fragment or assembling a complete private

key in this process.

2.2 Verification of public key and signature

The signature generated by Distributed Control Rights Management complies with the digital signature specification of the targeted blockchain. Signature verification remains unchanged. The validation of message signatures involves three input items:

- The public key. In this case, it will be generated by the private key fragments based on distributed computation.
- The message signature. In this case, using private key fragments to generate signature for hash of message based on distributed computation.
- The message itself.

Signature verification algorithm [edit]

For Bob to authenticate Alice's signature, he must have a copy of her public-key curve point Q_A. Bob can verify Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid
2. Check that Q_A lies on the curve
3. Check that $n \times Q_A = O$

After that, Bob follows these steps:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \mathrm{HASH}(m)$, where HASH is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $w = s^{-1} \mod n$.
5. Calculate $u_1 = zw \mod n$ and $u_2 = rw \mod n$.
6. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
7. The signature is valid if $r \equiv x_1 \pmod n$, invalid otherwise.

Note that using Shamir's trick, a sum of two scalar multiplications $u_1 \times G + u_2 \times Q_A$ can be calculated faster than two scalar multiplications done independently.[5]

Specification description of the ECDSA signature verification on Wikipedia

In this verification program, we verify the public key, message and message signature generated by using the method that conforms to the targeted blockchain ECDSA specification. If verification succeeds, we have demonstrated that FUSION Distributed Control Rights Management achieves the status of using the private key fragments, distributed computation, and zero knowledge proofs to generate the public/private key pairing and the message signature for the targeted blockchain.

With Distributed Control Rights Management, the control and management of the target blockchain account system and crypto assets can be realized on FUSION.

2.3 Verification of application scope for Distributed Control Rights Management management

At present, most blockchains use ECDSA algorithm to implement public and private key pairing and message signature.

According to bitcoin and Ethereum's official documentation and code, both adopt the secp256k1 specification. USDT is based on bitcoin implementation. Therefore, all the above three adopt the same ECDSA specification, that is, secp256k1.

The specification of secp256k1 is adopted in this verification program, which shows that the Distributed Control Rights Management of FUSION can generate the public and private key pairs and the message signatures that are valid on bitcoin, Ethereum and USDT.

```
18    static {
19        X9ECParameters params = SECNamedCurves.getByName("secp256k1");
20        CURVE = new ECDomainParameters(params.getCurve(), params.getG(), params.getN(), params.getH());
21        G = CURVE.getG();
22        q = params.getN();
23    }
```

As can be seen in the source code, the program can support the other specification of ECDSA algorithms, such as secp256v1 / secp256r1, so it can adapt to control of the different targeted blockchain account and assets by adjusting the parameter. You can find this parameter in the source code.
(/FUSIONDCRM/src/org.fsn_cfc.util/BitcoinParams.java)

2.4 Additional Notes

Paillier Cryptosystem

The FUSION's Distributed Control Rights Management, adopts the Paillier Cryptosystem to achieve secure computations among nodes without leaking any information. In the scheme design and final implementation, the private key of the Paillier Cryptosystem is generated and stored in a distributed manner, which ensures that no node can decrypt the result alone. There are a lot of mature implementations of Paillier encryption. In this demo verification program, in order to demonstrate Distributed ECDSA Key Generation, we simplify the process of distributed Paillier encryption.

Blockchain Address Generation

The conversion of public key to blockchrain address is standard hash functions. Distributed computation is not needed. So, the demo verification program does not cover any generation or verification of blockchain address.

Some notes about Java Programming

In order to implement and demonstrate verification program more rapidly, we use Java for developing purposes. After this, we will switch to Golang, and finally run Distributed Control Rights Management on the FUSION mainnet.

In anticipation of the Lock-in and management of real assets, Distributed Control Rights Management will be tested in the FUSION testnet to demonstrate and tune, and so as to get the optimal implementation.

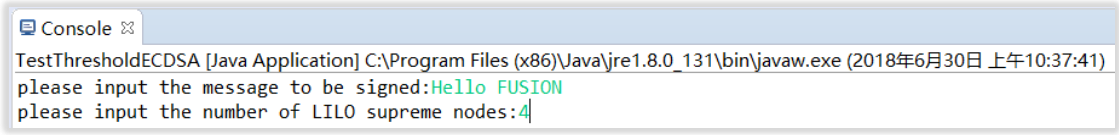
3. Description of the programs

The entrance of the demo verification program is located in the package /FUSIONDCRM/src/org.fsn_cfc.test, which contains two Java files: TestThresholdECDSA.java and BatchThresholdECDSA.java. The two main programs adopt the same distributed process from private key to public key and ECDSA signature. The differences are:

- TestThresholdECDSA.java

By running this Java program, you can input the message to be signed and the number of participants, then get a signature (r, s) and a result(Yes/No) to indicate whether the signature (r, s) is valid.
In the program execution, users can watch the process where every node generates and stores a private key fragment, and generates a public key based on private key fragments, generating ECDSA signature and ECDSA signature verification.
This program demonstrates the implementation of Distributed Control Rights

Management.



TestThresholdECDSA.java takes two inputs, the message to be signed and the number of simulated nodes. These two parameters can be selected by users.

After inputting the parameters, the demo verification program will execute and output the intermediate data and the final verification result.

```
please input the message to be signed:Hello FUSION
please input the number of LILO supreme nodes:4

--Info: User 0 generate Private Key Share
 PrivateKey Share: 257283045385325139462244716346619323342799759094471901597145658649655215996 53

--Info: User 1 generate Private Key Share
 PrivateKey Share: 748842497874717710112239867070683530324816299343918024140396514534254314594 72

--Info: User 2 generate Private Key Share
 PrivateKey Share: 652666087524635926190016499393796019986398386863272549323274390025866620057 12

--Info: User 3 generate Private Key Share
 PrivateKey Share: 464188342044249968029288004233703760061553627430434961321633739070546344345 41
```

Generate the private key fragments by nodes separately

```
--Info: Calculate the Encrypted Private Key
 EncPrivateKey: 628500009802070732212160643839540614840232588921535767425563617184831660368101855136 5

--Info: Calculate the Public Key
 PublicKey: (3044c385346146ae87a492683d0d199a5ccb44824cc0375d0b5d3afac876cfc3,fecc217a9cedba2399f1e8 b
```

Distributed the public key generation based on the private key fragments

```
--Info: User 0 calculate Commitment in round ONE

--Info: User 1 calculate Commitment in round ONE

--Info: User 2 calculate Commitment in round ONE

--Info: User 3 calculate Commitment in round ONE
```

The first signature operation by nodes separately

```
--Info: User 0 calculate Zero-Knowledge in round TWO

--Info: User 1 calculate Zero-Knowledge in round TWO

--Info: User 2 calculate Zero-Knowledge in round TWO

--Info: User 3 calculate Zero-Knowledge in round TWO

--Info: Calculate the Encrypted Inner-Data u
 u: 241054852651001168196297435838275066354636542395723341691873769176595997508632830 9

--Info: Calculate the Encrypted Inner-Data v
 v: 170712365757886187795141496356180119976093179087985803761813624784405467710189719 5
```

The second signature operation by nodes separately, generates the intermediate encrypted values u and v

```
--Info: User 0 calculate Commitment in round THREE

--Info: User 1 calculate Commitment in round THREE

--Info: User 2 calculate Commitment in round THREE

--Info: User 3 calculate Commitment in round THREE
```

The third signature operation by nodes separately

```
--Info: User 0 calculate Zero-Knowledge in round FOUR

--Info: User 1 calculate Zero-Knowledge in round FOUR

--Info: User 2 calculate Zero-Knowledge in round FOUR

--Info: User 3 calculate Zero-Knowledge in round FOUR

--Info: Calculate the Encrypted Inner-Data w
 w: 113407982036782639305120060667509729666404951225553037982122063103207435670909

--Info: Calculate the Encrypted Inner-Data R
 R: (ff93e1d60b9898c3d3aeecabe0bf43f0ce8ca45387c81ef15c55a7980d2221fc,7598ea77fb0a
```

The fourth signature operation by nodes separately, generates the intermediate encrypted values w and R

```
--Info: Calculate the ECDSA Signature in round FIVE
 signature: (r,s)=(81258593151862018334935525083497908169853273955350114693910634991113259145053,1089990447
```

The fifth signature operation by nodes separately, generates the ECDSA signatures for messages

```
--Info: ECDSA Signature Verify Passed!
 (r,s)=(81258593151862018334935525083497908169853273955350114693910634991113259145053,1089990447571
```

Verify this signature with the standard ECDSA signature verification specification

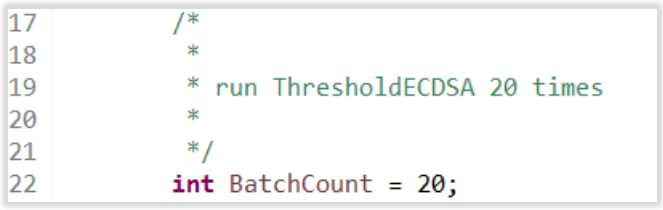
- BatchThresholdECDSA.java

  By running this Java program, you will get BatchCount (default 20) signatures independently and the corresponding running time which varies by the number of participants and key length.

  This program is used for verifying the stability of Distributed Control Rights Management to ensure that the scheme is correct. We have tested 500 times with 100% success rate.

```
--Info: Calculate the Encrypted Inner-Data w
 w: 18204864364150252212018527600931476995042511992096077551544038524274734731936382297775446904882463 69

--Info: Calculate the Encrypted Inner-Data R
 R: (8f7a0ec7ac6dd33cc512abcf82d93b12aecd245d975c3908f2691d90d9f714b5,ec86aa039ca0aa2cfbe14560533181df50

--Info: Calculate the ECDSA Signature in round FIVE
 signature: (r,s)=(11515932839984892068953467943595418964525406978794636973322123930855507872187 6,931641


--Info: ECDSA Signature Verify Passed!
 (r,s)=(11515932839984892068953467943595418964525406978794636973322123930855507872187 6,93164184714528127


Current Count is 20 with running time:19313
```

This program will execute 20 independent processes: generating public key based on the distributed private key and generating ECDSA signature with the default message in the form of 4 simulated nodes. It will return the intermediate data, the executed result of distributed private key and signature verification and time-consuming information.



```
17          /*
18           *
19           * run ThresholdECDSA 20 times
20           *
21           */
22          int BatchCount = 20;
```

Users can modify the parameter about execution time to test the demo verification program.

4. Compilation and test on Windows

4.1 Contents

The compressed file, named “FUSION DCRM verification v1.zip”, includes the source code of verification program, and a dynamic linking library file named gmp.dll.

Gmp.dll comes from the mathematical computing algorithm library of GMP (The GNU Multiple Precision Arithmetic Library, https://gmplib.org/).

4.2 Environment

Windows environment configuration.

1. Download and install the JDK 1.8 32bit for Windows.

Download address:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. And configure system environment variables based on where JDK is actually installed, such as:

And put the gmp.dll in this directory: jre/bin.

```
JAVA_HOME:C:\Program Files (x86)\Java\jdk1.8.0_131\
Path:%JAVA_HOME%/bin ;
CLASSPATH=.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;
```

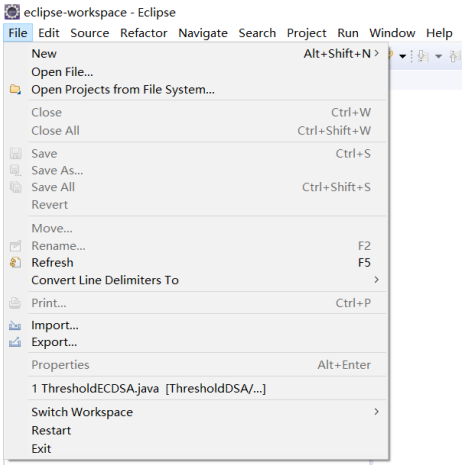
2. Download and install the Eclipse 32 bit for Windows.

Download address:

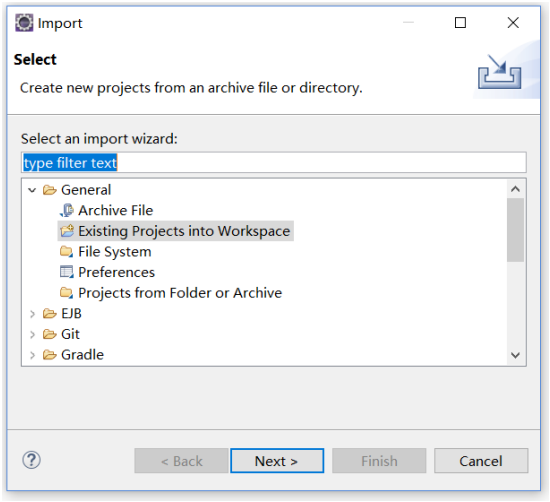
http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/oxygen/3a/eclipse-jee-oxygen-3a-win32.zip

4.3 Compile and execute

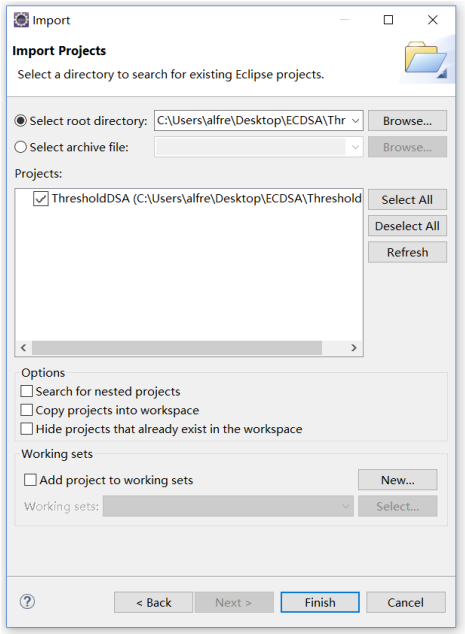
Run the Eclipse, and click the “import” option on the “file” menu.



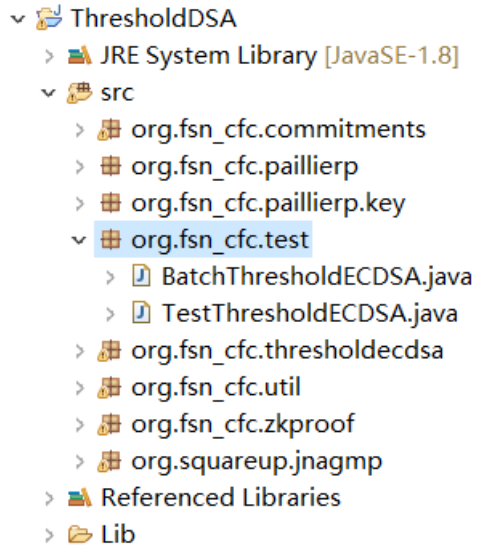
Select the option named “Existing projects into workspace” of “General” item and click “next”.



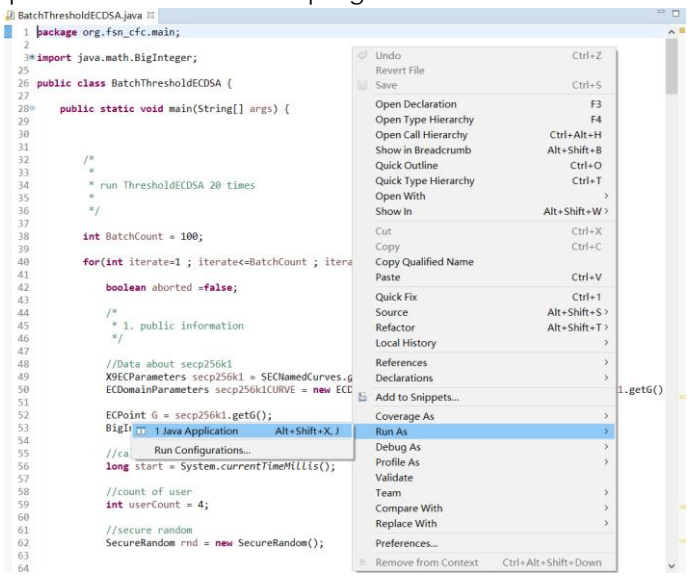
By clicking the button named “browse”, select the folder where the validator program folder resides, and click “finish”.



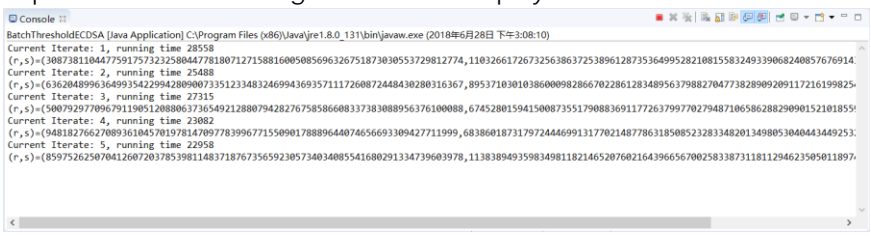
The project directory of FUSIONDCRM is displayed on the left of the Eclipse window. There are two programs: TestThresholdECDSA.java and BatchThresholdECDSA.java in the path of /FUSIONDCRM/src/org.fsn_cfc.test.



Double-click one of them, you can see the source code in the middle of the Eclipse window. Right-click on the code area, select "Run As" from the menu that appears, and click "Java Application" to execute the program.



The output information during execution is displayed in the “Console” window.



5. Compilation and test on Mac OS

5.1 Contents

The compressed file, named “FUSION DCRM verification v1.zip”, includes the source code of verification program.

5.2 Environment

Mac OS environment configuration:

1. Download and install the JDK 10.0. for Mac OS.

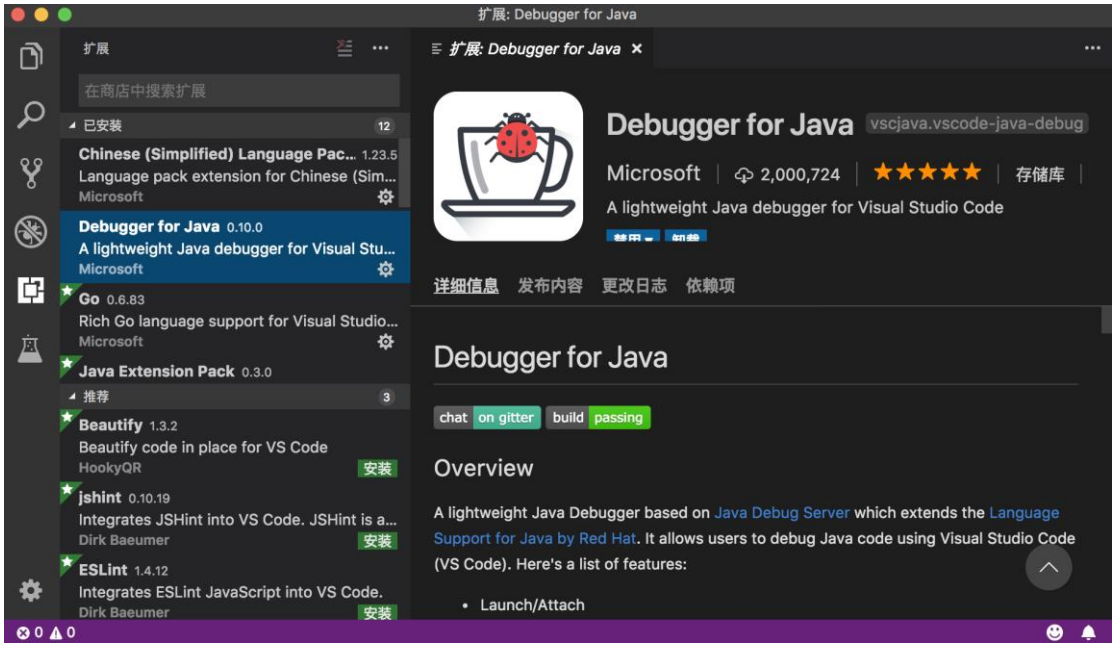
Download address:
http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html

2. Download and install the Visual Studio Code for Mac OS.

Download address:
https://code.visualstudio.com/#alt-downloads

3. Download and install the java debugging plug-in for Visual Studio Code.

Download address:
https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-debug



4. Install the brew and libgmp

Install the brew package management, and enter the command under the terminal as follow.

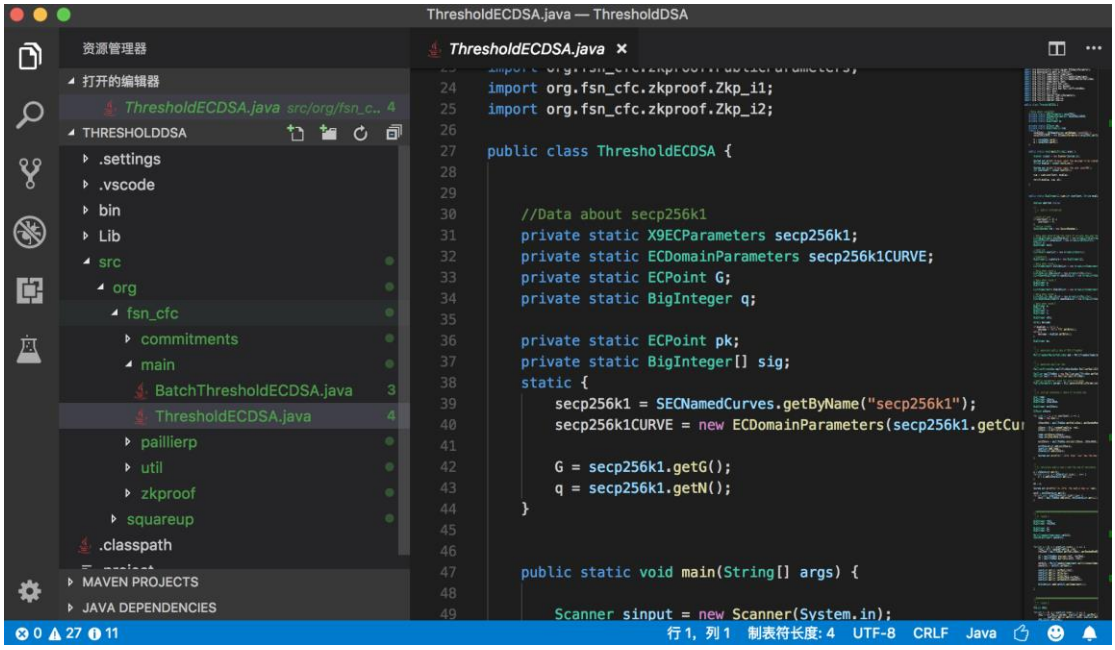
```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

5. Install the libgmp.

```
brew install gmp
```

After successful brew package management installation, enter the command in the terminal as follows.

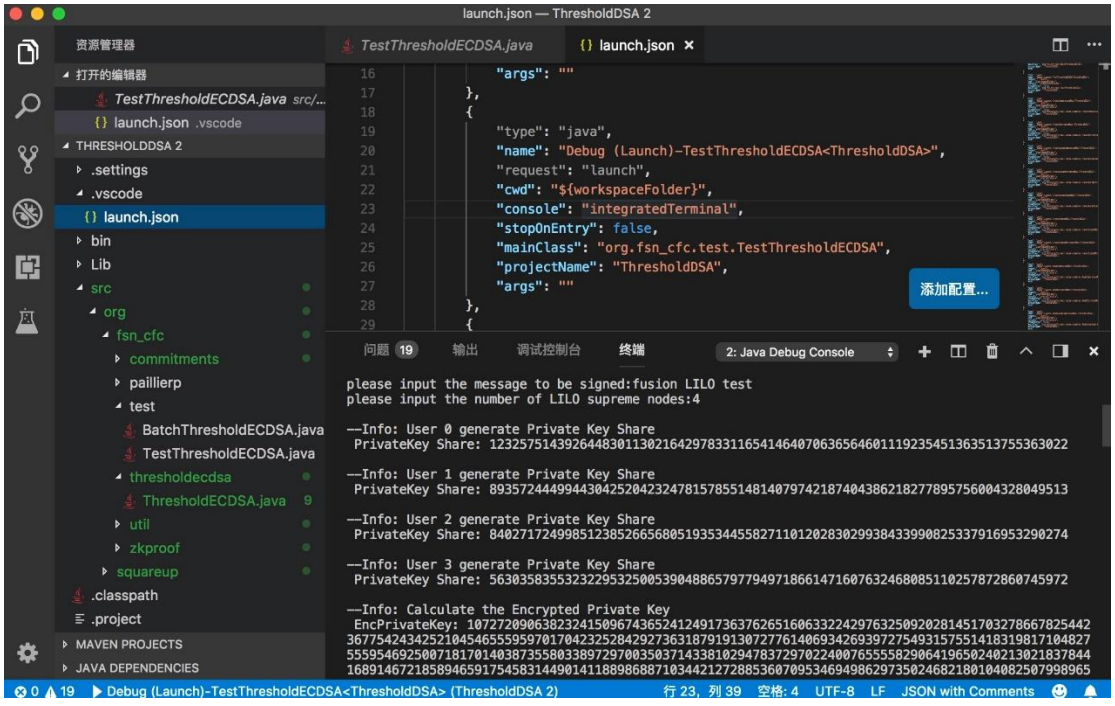
5.3 Compile and execute



1. Open the project folder in the Visual Studio Code.

2. After generating the project configuration file launch.json, press F5, run the main program.

Note: because the default configuration of VSCode does not support accepting input parameters in a debug environment, there will be an error message: Failed to evaluate. Reason: Cannot evaluate because the thread is resumed。



The solution is to change the parameters of the main class in launch.json as follow:

```
"console": "integratedTerminal"
```

3. After inputting the signature message and number of nodes in the debugging terminal window, output the execution state information content and pass the test.

```
please input the message to be signed:fusion LILO test
please input the number of LILO supreme nodes:4
```

6. Compilation and test on Linux

6.1 Contents

The compressed file named “FUSION DCRM verification v1.zip”, includes the source code of verification program.

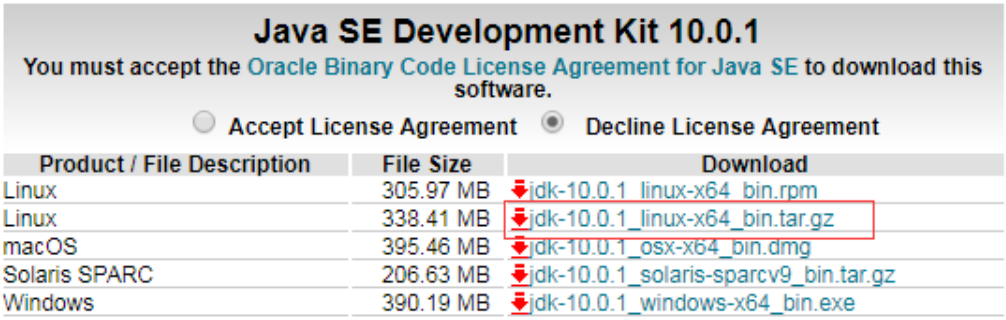
6.2 Environment

Linux environment configuration:

1. Download and install the JDK10.0.1 64bit for Linux.

Download address:

http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html



Java SE Development Kit 10.0.1

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux	305.97 MB	jdk-10.0.1_linux-x64_bin.rpm
Linux	338.41 MB	jdk-10.0.1_linux-x64_bin.tar.gz
macOS	395.46 MB	jdk-10.0.1_osx-x64_bin.dmg
Solaris SPARC	206.63 MB	jdk-10.0.1_solaris-sparcv9_bin.tar.gz
Windows	390.19 MB	jdk-10.0.1_windows-x64_bin.exe

Configure the system environment variables as follow.

```
export JAVA_HOME=/opt/jdk-10.0.1        /*安装目录*/
export JRE_HOME=${JAVA_HOME}/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export  PATH=${JAVA_HOME}/bin:$PATH
```

2. Download and install the Eclipse for Linux.

Download address:

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/oxygen/3a/eclipse-jee-oxygen-3a-linux-gtk-x86_64.tar.gz

6.3 Compile and execute

Run the program in the Eclipse is as same as running on Windows.