TCSS 487 Cryptography

Practical project – cryptographic library & app

Group: Vu Nguyen, Rin Pham, Duy Vu

I. Solution

A. Part 1: HA-3 derived function KMACXOF256

- We implement SHA-3 derived function KMACXOF256 using NIST Special Publication 800-185, FIPS PUB 202, and Keccak-reference-3.0 as references.
- We use byte arrays to describe a bit's strings in KMACOF256 supporting functions bytepad, encode_string, left_encode, right_encode.
- But we use byte arrays as bit arrays to describe a bit's strings in all other functions because we feel it is easier for us to follow the description from NIST documents.
- We used the function bytepad provided by the professor.
- Below is the description of the code files and the person who implements them.
- Libraries codes inside the package: edu.uw.tcss487.vrdgroup.main
    - (Vu Nguyen) Keccak1600.java: implement KECCAK[c], SPONGE, KECCAK-p[1600, 24], and permutation state functions: theta, roh, pi, chi, iota.
    - (Vu Nguyen) SHA3.java: Implemented KMACOF256 and its supporting functions including SHAKE256 and cSHAKE256.
    - (Duy Vu) Utils.java: implement utility methods such as xor2Lanes, ROT (rotate a lane from Keccak reference), etc.
- The main features codes inside the package: edu.uw.tcss487.vrdgroup.main.features
    - (Duy Vu) Hash.java: implement hash functions based on the high-level specification.
    - (Vu Nguyen) SymmetricCrypto.java: implement symmetrically encrypt/decrypt, compute authenticate tag based on the high-level specification.
- The test codes inside the package: edu.uw.tcss487.vrdgroup.tests

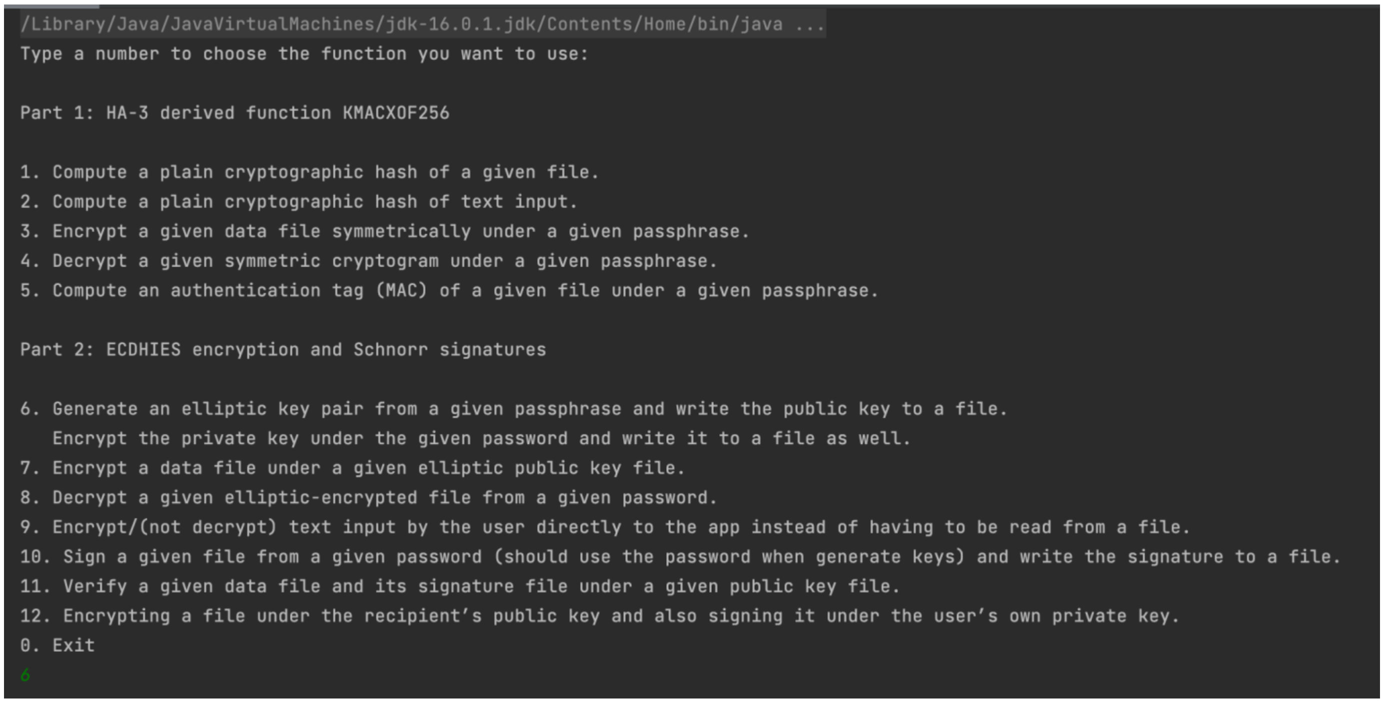
    - (Vu Nguyen) Keccak1600Test.java: implement unit tests for important functions of Keccak such as theta, roh, pi, chi, iota, and states converting.
    - (Vu Nguyen) SHA3Test.java: implement unit tests for KMACOF256 supporting functions and all tests from the high-level specification of the project.
    - (Duy Vu) UtilsTest.java: implement unit tests for important utility functions.

B. Part 2: ECDHIES encryption and Schnorr signatures

- We implement the E521 Elliptic curve based on the description, pseudo-code and the code from the lecture and the project description.
- Below is the description of the code files and the person who implements them.
- Libraries codes inside the package: edu.uw.tcss487.vrdgroup.main
    - (Rin Pham) EdwardsPoint.java: implement the E521 Elliptic curve with all functions described in the project description.
    - (Vu Nguyen) Main.java: implement user interface and combine all features.
- The main features codes inside the package: edu.uw.tcss487.vrdgroup.main.features
    - (Vu Nguyen) EllipticCrypto.java: implement elliptic cryptography features such as generate key/pair, encrypt/decrypt.
    - (Duy Vu) Signature.java: implement signature features such as sign and verify the signature.
- The test codes inside the package: edu.uw.tcss487.vrdgroup.tests
    - (Rin Pham) EdwardsPointTest.java: implement unit tests based on the suggestion from the project description.

II. User Instruction

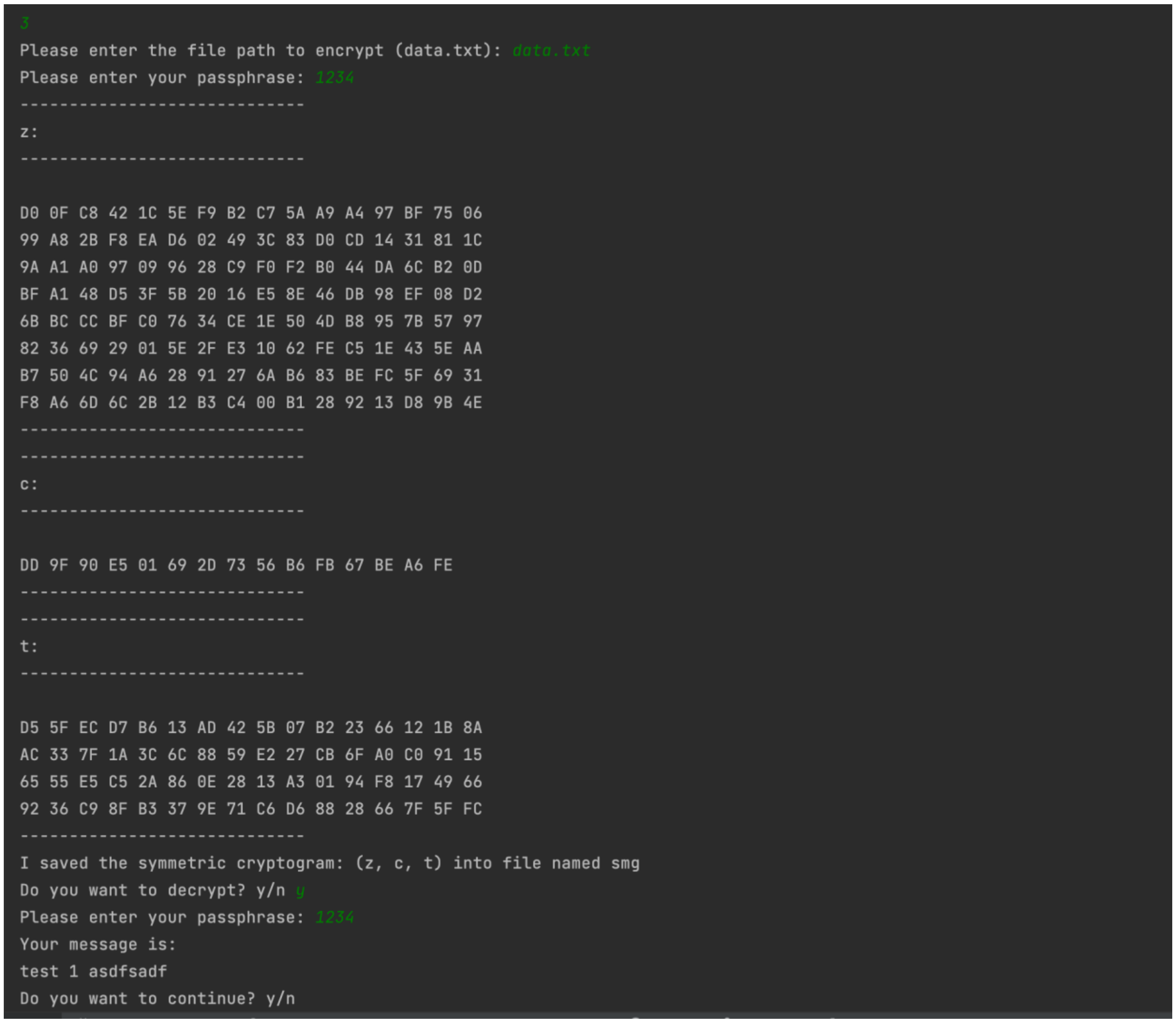
- To run the app, compile the source code and run the Java class edu.uw.tcss487.vrdgroup.main.Main
- This is a command-line application, it provides 12 services as the below screenshot shows, and you can type the number of which service you want to test, and push enter to run the service. The green text in the screenshot is the user input text.



```
/Library/Java/JavaVirtualMachines/jdk-16.0.1.jdk/Contents/Home/bin/java ...
Type a number to choose the function you want to use:

Part 1: HA-3 derived function KMACXOF256

1. Compute a plain cryptographic hash of a given file.
2. Compute a plain cryptographic hash of text input.
3. Encrypt a given data file symmetrically under a given passphrase.
4. Decrypt a given symmetric cryptogram under a given passphrase.
5. Compute an authentication tag (MAC) of a given file under a given passphrase.

Part 2: ECDHIES encryption and Schnorr signatures

6. Generate an elliptic key pair from a given passphrase and write the public key to a file.
   Encrypt the private key under the given password and write it to a file as well.
7. Encrypt a data file under a given elliptic public key file.
8. Decrypt a given elliptic-encrypted file from a given password.
9. Encrypt/(not decrypt) text input by the user directly to the app instead of having to be read from a file.
10. Sign a given file from a given password (should use the password when generate keys) and write the signature to a file.
11. Verify a given data file and its signature file under a given public key file.
12. Encrypting a file under the recipient’s public key and also signing it under the user’s own private key.
0. Exit
6
```

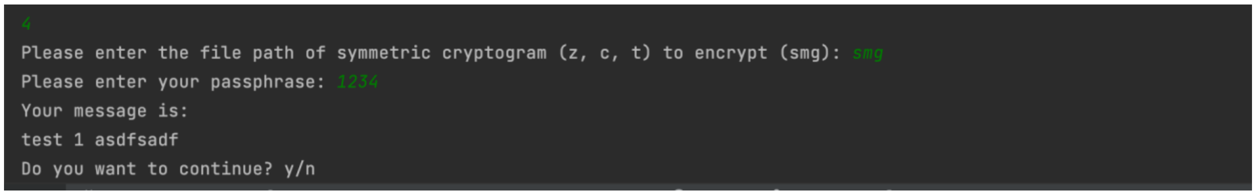
- A service will ask for exactly the information described in the project description, and it will print the result to the screen. The result can be a big integer string and byte array in hex string form. It may notice the name of any file it saved (almost files were created by using the Java Serializable interface). And at the end, it will ask if the user wants to continue running this service again.
- Special, service 3 and 7 will ask if the user wants to decrypt the file for a quick check.
- Below is the screenshot of some services

3. Encrypt a given data file symmetrically under a given passphrase



```
3
Please enter the file path to encrypt (data.txt): data.txt
Please enter your passphrase: 1234
-----------------------------
z:
-----------------------------

D0 0F C8 42 1C 5E F9 B2 C7 5A A9 A4 97 BF 75 06
99 A8 2B F8 EA D6 02 49 3C 83 D0 CD 14 31 81 1C
9A A1 A0 97 09 96 28 C9 F0 F2 B0 44 DA 6C B2 0D
BF A1 48 D5 3F 5B 20 16 E5 8E 46 DB 98 EF 08 D2
6B BC CC BF C0 76 34 CE 1E 50 4D B8 95 7B 57 97
82 36 69 29 01 5E 2F E3 10 62 FE C5 1E 43 5E AA
B7 50 4C 94 A6 28 91 27 6A B6 83 BE FC 5F 69 31
F8 A6 6D 6C 2B 12 B3 C4 00 B1 28 92 13 D8 9B 4E
-----------------------------
-----------------------------
c:
-----------------------------

DD 9F 90 E5 01 69 2D 73 56 B6 FB 67 BE A6 FE
-----------------------------
-----------------------------
t:
-----------------------------

D5 5F EC D7 B6 13 AD 42 5B 07 B2 23 66 12 1B 8A
AC 33 7F 1A 3C 6C 88 59 E2 27 CB 6F A0 C0 91 15
65 55 E5 C5 2A 86 0E 28 13 A3 01 94 F8 17 49 66
92 36 C9 8F B3 37 9E 71 C6 D6 88 28 66 7F 5F FC
-----------------------------
I saved the symmetric cryptogram: (z, c, t) into file named smg
Do you want to decrypt? y/n y
Please enter your passphrase: 1234
Your message is:
test 1 asdfsadf
Do you want to continue? y/n
```

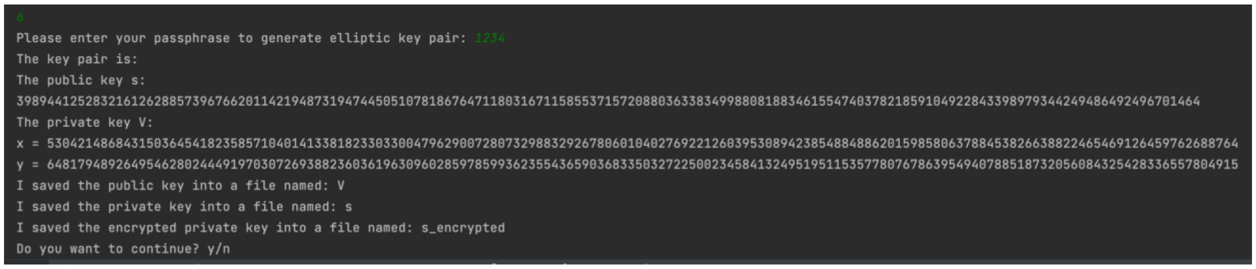
4. Decrypt a given symmetric cryptogram under a given passphrase



```
4
Please enter the file path of symmetric cryptogram (z, c, t) to encrypt (smg): smg
Please enter your passphrase: 1234
Your message is:
test 1 asdfsadf
Do you want to continue? y/n
```

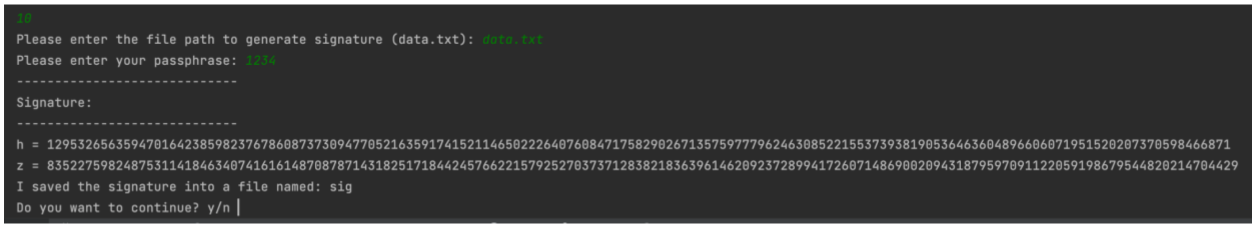
6. Generate an elliptic key pair from a given passphrase and write the public key to a file

Encrypt the private key under the given password and write it to a file as well



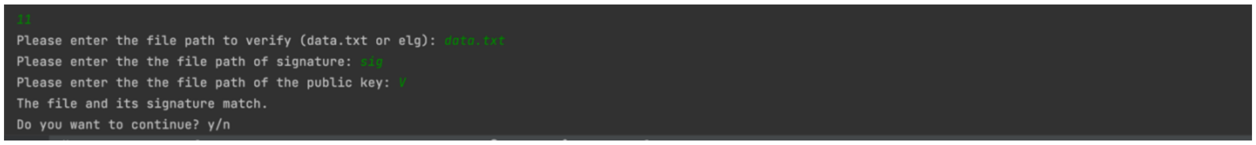
```
6
Please enter your passphrase to generate elliptic key pair: 1234
The key pair is:
The public key s:
398944125283216126288573967662011421948731947445051078186764711803167115855371572088036338349988081883461554740378218591049228433989793442494864924967 01464
The private key V:
x = 530421486843150364541823585710401413381823303300479629007280732988329267806010402769221260395308942385488488620159858063788453826638822465469126459762688764
y = 648179489264954628024449197030726938823603619630960285978599362355436590368335032722500234584132495195115357780767863954940788518732056084325428336557804915
I saved the public key into a file named: V
I saved the private key into a file named: s
I saved the encrypted private key into a file named: s_encrypted
Do you want to continue? y/n
```

10. Sign a given file from a given password (should use the password when generating keys) and write the signature to a file



```
10
Please enter the file path to generate signature (data.txt): data.txt
Please enter your passphrase: 1234
----------------------------
Signature:
----------------------------
h = 12953265635947016423859823767860873730947705216359174152114650222640760847175829026713575977796246308522155373938190536463604896606071951520207370598466871
z = 83522759824875311418463407416161487087871431825171844245766221579252703737128382183639614620923728994172607148690020943187959709112205919867954482021470 4429
I saved the signature into a file named: sig
Do you want to continue? y/n |
```

11. Verify a given data file and its signature file under a given public key file



```
11
Please enter the file path to verify (data.txt or elg): data.txt
Please enter the the file path of signature: sig
Please enter the the file path of the public key: V
The file and its signature match.
Do you want to continue? y/n
```

III. Known Bugs

- Although the application can run all 12 services without issues, we know at least three Keccka functions do not work correctly, they are theta, roh, and iota. They do not pass our unit tests.