

Name: Akshit Arora  
NetID: aeora44

**TCS 487**  
**Cryptography**  
**Practical Project Report**

## **Implementation Description**

The cryptography app offers the following features:

1. Compute a plain cryptographic hash of a given file.
2. Encrypt a given data file symmetrically under a given passphrase.
3. Decrypt a given symmetric cryptogram under a given passphrase.
4. Generate an elliptic key pair from a given passphrase and write the public key to a file.
5. Encrypt a data file under a given elliptic public key file.
6. Encrypt a given elliptic-encrypted file from a given password.
7. Sign a given file from a given password and write the signature to a file.
8. Verify a given data file and its signature file under a given public key file.
9. Compute an authentication tag (MAC) of a given file under a given passphrase. (Bonus question)

The project was divided into two parts, with part 1 implementing the features 1-3 and part 2 implementing the features 4-9

### **Part 1**

Java Classes Implemented in Part 1: Sha3.java, KMACXOF256.java, and InternalFunctions.java

**InternalFunctions.java:** This was the first class I implemented following the Section 2.3 of the NIST Special Publication 800-185. All the Internal Functions mentioned in the paper such as right\_encode, left\_encode, bytepad, substring was implemented in this class using the algorithms described in the paper. Testing of the methods was done according to the examples given in the Section 2.3 of the paper in the main method of the InternalFunctions (not included in the final submission).

**Sha3.java:** The next step of development was to implement the Sha3.java class following the NIST Special Publication 800-185 and Markku-Juhani O. Saarinen's sha3.c implementation. This part was pretty straightforward since sha3.c was self-explanatory. I struggled a bit with converting byte array to long and vice-versa. I referred to stackoverflow for understanding this part (method snippet referred in the java file).

**KMACXOF256.java:** The KMACXOF256 and cSHAKE256 methods were implemented following the Section 3 and Section 4 of the NIST Special Publication 800-185. Both methods were pretty straightforward and were implemented using the definitions in the paper. The methods were tested using the test vectors provided in the project description.

## Part 2

Java Classes Implemented in Part 2: Driver.java and EllipticCurve521.java

I divided the part 2 of the project into 3 parts, first implementing the symmetric cryptography features to the Driver.java, then creating the Edwards curve or E521 implementation of the Elliptic Curve. Lastly, I implemented the features 4-8 into the Driver and implemented some bonus features.

**EllipticCurve521.java:** I started by learning about the BigInteger class, since it was my first time using it. After familiarizing myself with BigInteger, I created a pseudocode for all the methods described in the project description. Three constructors are defined in the EllipticCurve521 class according to the project description. I had some issues figuring out how to get the public generator and hence some of the tests were failing. Once I figured it out, everything went pretty smoothly. The test cases are described in the main method of the EllipticCurve521. I have commented it for now but please feel free to uncomment it and test the functionality of the class. The sqrt method was directly taken from the project description and the multiplyScalar method was inspired by the algorithm described in the appendix.

**Driver.java:** This was the easiest part of the project for me since the implementation of all the features were clearly described in the project description. I chose to create a command-line based application along with JFileChooser for selecting and saving the files. All the features are divided into separate methods and clearly described in the Driver class. The code contains comments describing how I translated the algorithms defined in the project specification to the java code.

## User Instructions

The project was built using Eclipse IDE. Please import the project archive file into Eclipse and run the Driver.java class for using the app.

Once Driver.java is compiled and run, a command line interface would be displayed that looks like the one below:

```
Console [X]
Driver (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/Home/bin/java (Jun 4, 2021, 10:47:25 PM)

----- CRYPTOGRAPHY PROJECT -----
----- By Akshit Arora -----

*****
Compute a plain cryptographic hash of a given file: Enter 1
Encrypt a given data file symmetrically under a given passphrase: Enter 2
Decrypt a given symmetric cryptogram under a given passphrase: Enter 3
Generate an elliptic key pair from a given passphrase and write the public key to a file: Enter 4
Encrypt a data file under a given elliptic public key file: Enter 5
Decrypt a given elliptic-encrypted file from a given password: Enter 6
Sign a given file from a given password and write the signature to a file: Enter 7
Verify a given data file and its signature file under a given public key file: Enter 8
Extra Credit #2, Compute an authentication tag (MAC) of a given file under a given passphrase: Press 9
Quit the App: Enter 10
*****
```

Enter the feature you want to run, here's a demo for running feature 2.

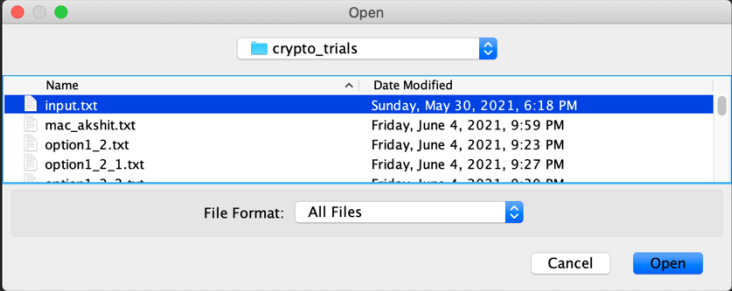
After entering 2, you would get the following result and would be prompted to select the input file using a JFileChooser.

```
Console [X]
Driver (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/Home/bin/java (Jun 4, 2021, 10:47:25 PM)

----- CRYPTOGRAPHY PROJECT -----
----- By Akshit Arora -----

*****
Compute a plain cryptographic hash of a given file: Enter 1
Encrypt a given data file symmetrically under a given passphrase: Enter 2
Decrypt a given symmetric cryptogram under a given passphrase: Enter 3
Generate an elliptic key pair from a given passphrase and write the public key to a file: Enter 4
Encrypt a data file under a given elliptic public key file: Enter 5
Decrypt a given elliptic-encrypted file from a given password: Enter 6
Sign a given file from a given password and write the signature to a file: Enter 7
Verify a given data file and its signature file under a given public key file: Enter 8
Extra Credit #2, Compute an authentication tag (MAC) of a given file under a given passphrase: Press 9
Quit the App: Enter 10
*****
2
Your chose: 2

*****
#2: Encrypt a given data file symmetrically under a given passphrase
Please select the input file.
```



After selecting the input file, you would be prompted to enter the passphrase and create the output file using the JFileChooser. Please note that JFileChooser sometimes gets minimized and you might have to click at the JFileChooser logo in the taskbar.

Here's the output after following all the steps for Option 2

```
----- CRYPTOGRAPHY PROJECT -----
----- By Akshit Arora -----

*****
Compute a plain cryptographic hash of a given file: Enter 1
Encrypt a given data file symmetrically under a given passphrase: Enter 2
Decrypt a given symmetric cryptogram under a given passphrase: Enter 3
Generate an elliptic key pair from a given passphrase and write the public key to a file: Enter 4
Encrypt a data file under a given elliptic public key file: Enter 5
Decrypt a given elliptic-encrypted file from a given password: Enter 6
Sign a given file from a given password and write the signature to a file: Enter 7
Verify a given data file and its signature file under a given public key file: Enter 8
Extra Credit #2, Compute an authentication tag (MAC) of a given file under a given passphrase: Press 9
Quit the App: Enter 10
*****
2
Your chose: 2

*****
-----#2: Encrypt a given data file symmetrically under a given passphrase-----
Please select the input file.
Please enter the passphrase
qazwsxedc123
Please select the output file

Symmetric cryptogram saved successfully at: /Users/stlp/Desktop/crypto_trials/enc_symm_input.txt

*****
Compute a plain cryptographic hash of a given file: Enter 1
Encrypt a given data file symmetrically under a given passphrase: Enter 2
Decrypt a given symmetric cryptogram under a given passphrase: Enter 3
Generate an elliptic key pair from a given passphrase and write the public key to a file: Enter 4
Encrypt a data file under a given elliptic public key file: Enter 5
Decrypt a given elliptic-encrypted file from a given password: Enter 6
Sign a given file from a given password and write the signature to a file: Enter 7
Verify a given data file and its signature file under a given public key file: Enter 8
Extra Credit #2, Compute an authentication tag (MAC) of a given file under a given passphrase: Press 9
Quit the App: Enter 10
*****
```

For quitting the app, simply enter 10.

## Issues Encountered

The app runs smoothly and has no bugs.

I wanted to mention that I tried implanting the other bonus questions as well such as creating the plain hash by directly getting input into the console, but was getting scanner errors and decided not to include it in the final submission.