Cryptography Project Report

Lam Mai, David Shcerbina, Daniel Jiang

General

User Instructions

Our program/library is a console application where the user can iterate through menus and choose whether they would like to do symmetric cryptography (choice `1`) or elliptic curve arithmetic (choice `2`). Furthermore, within the symmetric cryptography menu, the user is given choices **`a`** to **`e`**, each of which have a short description and within the elliptic curve arithmetic menu, the user is given choices **`f`** to **`m`**, each of which also have a short description.

Notes

- While we noticed in the NIST documents for cSHAKE examples have the hex string space-separated, our encryption and decryption would not work properly so for the sake of them working, our hex string outputs to the console or files will have no space-separation nor should the input file hex strings be space-separated.
- All input files that we are reading from should be in the `files/input` folder and all output files should go to the `files/output` folder
- Examples utilized in Part 2 - Elliptic Curve Arithmetic utilizes the phrase “secret passphrase” for password, passphrase, key generation, etc.

Known Bugs

- In part 2/elliptic curve arithmetic, part `j` may throw an `IllegalArgumentException` error for the `Arrays.copyOfRange()` method during an attempted decryption
  - The input public key should be in hex format and have a length of 132 hex characters
- Invalid user inputs might cause the program to experience a crash.

Part 1: Symmetric Cryptography (SHA-3 derived function KMACXOF256)

Solution Description

a) Compute plain cryptographic hash of a given file

- To get the cryptographic hash, we apply the following KMACXOF256 parameters

```
return KMACXOF256( key: "", m, outBitLen: 512, divS: "D");
```

- The format of the inputs in the file are lines of plain text where each line is consider one input
  - Example content of an input text file:

```
testing line 1
testing line 2
testing line 3
```

- The format of the output is a hex string where each line is the cryptographic hash of the corresponding plain text
  - Example console outputs of the above inputs:

```
Result:
F392C45D13AF96A1FF9C771AA2A9B0AAEAB60340F67E431DE118B57E9C65AA78574B29BD504A35E6CAEF1B3DCA86EFAAAEDE17C10D41FF8F54536493D7F151FD
CF7C8B8EB9AE2903331DC3390A99379948F9E90C84B60C598B1D71656652EE2C45ED7CFF256A76A82169C6BE06A11ACB0333071213EEAB91F1BEBFF78331CBAA
F0035207F7A84008ADB47B232175D4A5B7B288EB7628B01E27F5F4628C4B19E118AB5EB5DF1103F7E9774EC912AF01634AB01775C4918BEE3DD333DC066289F7
```

b) Compute plain cryptographic hash of text input [BONUS]

- We apply the same KMACXOF256 parameters as part a) to get the cryptographic hash
- The input can be any plain text message
- The format of the output is a hex string, which will be printed in the console:

```
Result:
5D1326841E932DC7A27508B059D371EC711971E7916F46EB93450257692EFF23FC458F00DD28C5BC44404F4C1CBB22CF7AB562A098AD831773C7B31C80BE78B8
```

c) Encrypt a given data file under a given passphrase

- Following the project specification for encrypting data symmetrically, we were able to obtain the encryption for our data

```
//z ← Random(512)
SecureRandom random = new SecureRandom();
byte[] z = new byte[64]; // 512 bits
random.nextBytes(z);
byte[] pwBytes = pw.getBytes();

// (ke || ka) ← KMACXOF256(z || pw, "", 1024, "S")
byte[] keyGen = KMACXOF256(byteArrayToString(byteConcat(z, pwBytes)), new byte[]{}, outBitLen: 1024, divS: "S");
byte[] ke = Arrays.copyOfRange(keyGen, from: 0, to: 64);
byte[] ka = Arrays.copyOfRange(keyGen, from: 64, to: 128);

// c ← KMACXOF256(ke, "", |m|, "SKE") xor m
byte[] toXor = KMACXOF256(byteArrayToString(ke), new byte[]{}, outBitLen: m.length * 8, divS: "SKE");
byte[] c = xorBytes(toXor, m);

// t ← KMACXOF256(ka, m, 512, "SKA")
byte[] t = KMACXOF256(byteArrayToString(ka), m, outBitLen: 512, divS: "SKA");

// symmetric cryptogram (z, c, t)
// (z || c || t)
return byteConcat(byteConcat(z, c), t);
```

- The input text file contain plain text messages like the following:

```
testing line 1
testing line 2
testing line 3
```

- The output will be written to a separate text file, containing the encrypted data for each line of input
  - We used the passphrase “mypass” to encrypt the messages above:

```
037746B876E349C4889A2810AACD7E528844DC9E0F04C2D5E3CD2978F77D02B21307DB0AE86CB1CB3C069338377D9556C8FC46FE6FD2A91946E90DADD0A1F655C00E992F48396A549864FED90D29DF244D190E
1130E67940E3ECEAACBE92AB736D3773612F40EF6351BD086250D8A04A6C3953EA767C6D91E17D3F38FADBD4E4BF8D63B498299105ECD956B93823BFEC5CD573ADA765930B4A06FEAF66D81F371385B3C192CF
DF0EB1F3A2EBE83A7778F6A8F26522334484CABDFC3F1A4853C00FFD36A8726BC3C4F13AD073299A3A18305B9CE6A88D57B2961214758C06316B8CF2180BF9E7BC048EEAC9AC3203A5BA8A94312BDB2F4A9D9F
```

d) Decrypt a given cryptogram under a given passphrase

- Following the project specification for decrypting a symmetric cryptogram, we are able to recover the plain text of the encrypted messages

```
// Taking z, c, t apart
byte[] z = Arrays.copyOfRange(zct, from: 0 , to: 64);
byte[] c = Arrays.copyOfRange(zct, from: 64, to: zct.length - 64);
byte[] t = Arrays.copyOfRange(zct, from: zct.length - 64, zct.length);

// (ke || ka) ← KMACXOF256(z || pw, "", 1024, "S")
byte[] keyGen = KMACXOF256(byteArrayToString(byteConcat(z, pw.getBytes())), new byte[]{}, outBitLen: 1024, divS: "S");
byte[] ke = Arrays.copyOfRange(keyGen, from: 0, to: 64);
byte[] ka = Arrays.copyOfRange(keyGen, from: 64, to: 128);

// m ← KMACXOF256(ke, "", |c|, "SKE") xor c
byte[] toXor = KMACXOF256(byteArrayToString(ke), new byte[]{}, outBitLen: c.length * 8, divS: "SKE");
byte[] m = xorBytes(toXor, c);

// t' ← KMACXOF256(ka, m, 512, "SKA")
byte[] tPrime = KMACXOF256(byteArrayToString(ka), m, outBitLen: 512, divS: "SKA");

// accept if, and only if, t' = t
// if t = tPrime returns decrypted message, else return cryptogram
return Arrays.equals(t, tPrime) ? m : c;
```

- The input text file contains the encrypted messages which are the output of part c)
- When entered the correct passphrase to decrypt, the output are the original plain text messages
- When entered the wrong passphrase, decryption will result in strings of unreadable characters:

```
SO /H9jT d
)
e VTJACK f US7DC3
EOT 12ETX 1+
```

e) Compute an authentication tag (MAC) of a given file under a given passphrase

- Following the project specifications, we apply the following KMACXOF256 parameters to obtain the authentication tag of a given file

```
return KMACXOF256(pw, m, outBitLen: 512, divS: "T");
```

- The input is a text file containing plain text messages:

```
testing line 1
testing line 2
testing line 3
```

- Using a passphrase of “mypass”, the resulting authentication tags are printed in the console for the corresponding input:

```
CC7ED4BF4F170F3C8A1CDE76EF51AF52EE8420BBBE598CF6012643FE92012235C7DCB8044D99DAD94A3BD7BC79123BDF389E5A9720A9EE07EF0BBE2BE4E4C1C2
2B017BE6C809AC93FDBE8ADBFA67B0CD73865618CA9D03CF810784DE711E266A5AB2895E32414ECA5EC6DF945ABFB54575701C178202B11160EDD37D289C5A9A
796EA713D04A44C1EFCE9B746612B3CB4641FFD48118585528ECDFD75A24B95DE04FF9B414AF7BB4FF1ED74BE573C734E71C8DFFA390E68BBA52BACAEAE7E18E
```

User Instructions

a) **Compute plain cryptographic hash of a given file**
   - The program prompts for an input file and reads from it. This file should contain plain text messages and is a file with ".txt" extension
   - The program then prints the cryptographic hash to the console output

b) **Compute plain cryptographic hash of text input [BONUS]**
   - The program prompts for a text input and prints the cryptographic hash to the console output

c) **Encrypt a given data file under a given passphrase**
   - The program prompts for an input file. This file should contain plain text messages and is a file with ".txt" extension
   - The program then prompts for a passphrase, prompts for an output file name, reads from the input file, and writes the encryption of the input file to the output file

d) **Decrypt a given cryptogram under a given passphrase**
   - The program prompts for an input file. This file should contain cryptographic hashes that was created by encrypting plain text data symmetrically. It should also be a file with ".txt" extension
   - The program then prompts for a passphrase, prompts for an output file name, reads from the input file, and writes the decryption of the input file to the output file

e) **Compute an authentication tag (MAC) of a given file under a given passphrase**
   - The program prompts for an input file. This file should contain plain text messages and is a file with ".txt" extension
   - The program then prompts for a passphrase, reads from the input file, and then prints out the authentication tag (MAC)

Part 2: Elliptic Curve Arithmetic (Edwards curve/ECDHIES encryption and Schnorr signatures)

Solution Description

f) Generate an elliptic key pair from a given passphrase and write the public key to a file

- In order to generate an elliptic key pair, we first gather a plain text passphrase from the user through the terminal.
- After which a *Schnorr/ECDHIES* key pair is generated after the keyphrase is converted into bytes. We are able to generate this key pair through KMAXOF256 and further manipulation as shown in the code below.

```
// s <- KMACXOF256(pw, "", 512, "K");
byte[] tempS = Symmetric.KMACXOF256(Symmetric.byteArrayToString(pw), new byte[]{}, outBitLen: 512, divS: "K");
byte[] sArr = new byte[65];
System.arraycopy(tempS, srcPos: 0, sArr, destPos: 1, tempS.length);

// s <- 4s
s_bigInt = new BigInteger(sArr).multiply(BigInteger.valueOf(4L));

// V <- s*G
// Note: s*G is multiplication of the scalar factor s by curve point G
s = s_bigInt.toByteArray();
V = G.multiply(s_bigInt);
// key pair: (s, V)
```

Hex results obtained from following ***part of*** user instructions.

```
00E6E94AA6D8D29D612A6AA195751A5775DD6A956975EA5BFAE994396110032532A74025D2BE8701B2CC71150054DE6CE6D4F4FDAF6B6A8953B30343984FF1CD5C9500EE3FF113AEE2AFF939B
```

g) Encrypt the private key under the given password and write it to a file [BONUS]

- While encrypting a private key under a passphrase and afterward writing it to a file, we will be using the same phrase from ***part f*** in the user instructions.
- As implied in the ***part f***, an *Schnorr/ECDHIES* will be generated and afterwards is multiplied by G producing this file result.

  Hex results obtained from following ***part g.***

```
308F763411752F2378A0BDB352D0EC8B1301CC864FD9E9640BD711CAED1947851AEF4CE9064F3FD15B68597F691E555A808D59BF5161F831A953D3A46FC8B4908
```

h) Encrypt a data file under a given elliptic public key file

- Following the project specifications this operation will encrypt the data file given an input to generate an elliptic public key. This is done with the following after converting data into a byte array.

```
//  k <- Random(512);
SecureRandom random = new SecureRandom();
byte[] randBytes = new byte[65];
random.nextBytes(randBytes);
randBytes[0] = 0;
BigInteger k = new BigInteger(randBytes);
//  k <- 4k
k = k.multiply(BigInteger.valueOf(4L));

//  W <- k*V;
//  Z <- k*G
E521 W = V.multiply(k);
E521 Z = ECKey.G.multiply(k);

//  (ke || ka) <- KMACXOF256(W_x, "", 1024, "P")
byte[] keyGen = Symmetric.KMACXOF256(
        Symmetric.byteArrayToString(W.getX().toByteArray()),
        new byte[]{},
         outBitLen: 1024,
         divS: "P");
byte[] ke = Arrays.copyOfRange(keyGen,  from: 0,  to: 64);
byte[] ka = Arrays.copyOfRange(keyGen,  from: 64,  to: 128);

//  c <- KMACXOF256(ke, "", |m|, "PKE") xor m
byte[] toXor = Symmetric.KMACXOF256(Symmetric.byteArrayToString(ke), new byte[]{},  outBitLen: m.length * 8,  divS: "PKE");
byte[] c = Symmetric.xorBytes(toXor, m);

//  t <- KMACXOF256(ka, m, 512, "PKA")
byte[] t = Symmetric.KMACXOF256(Symmetric.byteArrayToString(ka), m,  outBitLen: 512,  divS: "PKA");

//  cryptogram: (Z, c, t)
return Symmetric.byteConcat(Symmetric.byteConcat(Z.getBytes(), c), t);
```

- An input file of "***test.txt***" is provided in the input folder and will be used to further our test. The data file will be converted to a byte array and manipulated as shown in the snippet above.
- Once a given file name has been given the operation insert the filename "***partfPublicKey.txt***" when prompted for a public key file (*included in the input folder*).
- Once an output name is specified the program will save the new encrypted file.
  Hex results obtained from following ***part h.***

```
003D79C2166535E4CE7D7D2148B8687D47F7C828451FB0D13B130E9E2E5F97CF71C97DAC4B044985638D6060ABE1673B5C2F9E6E189011154EEF8E4D2DCD9964C5EA00850624ABB7FDEE15A0C03(  15
007FFA6AD23EF1DC17E12116921B001F79CD1513CE04D8D80551D61367E92BF1F1ECDE21B334B7C844717CFCCCFA9441343A1CB4125DC1D74D185BDDC4F1A01657C201A49C01DE8AC19608241CBCF6552FF722
01C00E0ECBC88AC1D69EB2429A423181E87BCE98D063BE9249A02454EBDB5B79875A8A635421C3AA6F671DB90456702EF5D4E8CCBED518DD92F406B0ADC5401158030110OFAC61B4BDAA123585DC6CF803E964
```

i) Decrypt a given elliptic-encrypted file from a given password

- Following the specifications from the document provided for decrypting an elliptic-encrypted file, we were able to decrypt the given file with its corresponding passphrase.

```
// s <- KMACXOF256(pw, "", 512, "K"); s <- 4s
byte[] tempS = Symmetric.KMACXOF256(pw, new byte[]{}, outBitLen: 512, divS: "K");
byte[] sArr = new byte[65];
System.arraycopy(tempS, srcPos: 0, sArr, destPos: 1, tempS.length);
BigInteger s = new BigInteger(sArr).multiply(BigInteger.valueOf(4L));

// W <- s*Z
E521 W = z.multiply(s);

// (ke || ka) <- KMACXOF256(W_x, "", 1024, "P")
byte[] keyGen = Symmetric.KMACXOF256(
        Symmetric.byteArrayToString(W.getX().toByteArray()),
        new byte[]{},
         outBitLen: 1024,
         divS: "P");
byte[] ke = Arrays.copyOfRange(keyGen, from: 0, to: 64);
byte[] ka = Arrays.copyOfRange(keyGen, from: 64, to: 128);

// m <- KMACXOF256(ke, "", |c|, "PKE") xor c
byte[] toXor = Symmetric.KMACXOF256(Symmetric.byteArrayToString(ke), new byte[]{}, outBitLen: c.length * 8, divS: "PKE");
byte[] m = Symmetric.xorBytes(toXor, c);

// t' <- KMACXOF256(ka, m, 512, "PKA")
byte[] tPrime = Symmetric.KMACXOF256(Symmetric.byteArrayToString(ka), m, outBitLen: 512, divS: "PKA");

// accept if, and only if, t' = t
return Arrays.equals(t, tPrime) ? m : c;
```

- Following ***part h*** we encrypted a datafile given a passphrase, the same encrypted file and passphrase will be used to decrypt the file, however the input folder contains the encrypted file for your convenience.
- Input the example data file name "***parthEllipticEncrypt.txt***" into the file to decrypt.
- Using the entered passphrase from **part f** will be used to decrypt the encrypted file to its original contents.

  File contents results obtained from following ***part i.***

```
testing line 1
testing line 2
testing line 3
```

j) Encrypt/decrypt text input [BONUS]

- Prompts the user to either encrypt or decrypt a text input, provided a public key (*hex values [length 132]*).
- The public key can be obtained from **part f**, where once provided the output will result in a hex value during encryption or the original plain text contents if the correct key is provided while in decrypting mode.

k) Sign a given file from a given password and write the signature to a file

- Users will be able to sign any file of their choosing when a password is given as long as the file extension is of a "***.txt***" extension.
- Following the project specifications we are able to generate the file signature.

```
// s <- KMACXOF256(pw, "", 512, "K"); s <- 4s
byte[] tempS = Symmetric.KMACXOF256(pw, new byte[]{}, outBitLen: 512, divS: "K");
byte[] sArr = new byte[65];
System.arraycopy(tempS, srcPos: 0, sArr, destPos: 1, tempS.length);
BigInteger s = new BigInteger(sArr).multiply(BigInteger.valueOf(4L));

// k <- KMACXOF256(s, m, 512, "N"); k <- 4k
byte[] tempK = Symmetric.KMACXOF256(s.toString(), m, outBitLen: 512, divS: "N");
byte[] kArr = new byte[65];
System.arraycopy(tempK, srcPos: 0, kArr, destPos: 1, tempK.length);
BigInteger k = new BigInteger(kArr).multiply(BigInteger.valueOf(4L));

// U <- k*G
E521 U = ECKey.G.multiply(k);

// h <- KMACXOF256(U_x, m, 512, "T"); z <- (k - hs) mod r
byte[] tempH = Symmetric.KMACXOF256(Symmetric.byteArrayToString(U.getX().toByteArray()), m, outBitLen: 512, divS: "T");
byte[] hArr = new byte[65];
System.arraycopy(tempH, srcPos: 0, hArr, destPos: 1, tempH.length);
BigInteger h = new BigInteger(hArr);
  BigInteger h = new BigInteger(hArr).multiply(BigInteger.valueOf(4L));

BigInteger z = k.subtract(h.multiply(s)).mod(E521.R);

// signature: (h, z)
return new BigInteger[]{h,z};
```

- With both the file and password numeric numbers serve in place as the digital file signature.
- Given multiple lines in a file, output will reflect the amount.
  Hex results obtained from following ***part h*** by using test.txt and the passphrase "***secret passphrase***".

```
78565125138400036071184439103676772616252280315365956267050244651168792672850153658081867500155813472182272810771053175609623988446922415323673405579266537719975 ✓
58383114765347801951837018348036711987176107354953454958784427397625420128224756824035099579373685973563520317935972501638707597730619993901245172674299868513043638421236064810184952416546408584296024331920270161067229722598858343452484705540711984554934856240288085936217398570241273099346344229962643917813653323077166512812287061
```

l) Verify a given data file and its signature file under a given public key file

- Through providing the original contents of the data file, output of the digital signature, and a public key from part F, we are able to verify the integrity of the file contents.
- The following implementation was obtained following project specifications.

```
BigInteger h = hz[0];
BigInteger z = hz[1];

// U <- z*G + h*V
E521 U = ECKey.G.multiply(z).add( V.multiply(h) );

// accept if, and only if, KMACXOF256(Ux, m, 512, "T") = h
byte[] tempH = Symmetric.KMACXOF256(Symmetric.byteArrayToString(U.getX().toByteArray()), m, outBitLen: 512, divS: "T");
byte[] hArr = new byte[65];
System.arraycopy(tempH, srcPos: 0, hArr, destPos: 1, tempH.length);
BigInteger myH = new BigInteger(hArr);
```

- If successful the message "This is verified" will appear or else the user will be prompted with the message "This is NOT verified".

m) Offer the possibility of encrypting a file under the recipient's public key and also signing it under the user's own private key [BONUS]

- We as a group chose not to implement this feature

User Instructions

f) Generate an elliptic key pair from a given passphrase and write the public key to a file

- The program prompts for a passphrase and generates a public key which is written to an output file
- As example we will be using the passphrase "***secret passphrase***".

```
Enter a passphrase for the generated elliptic key pair public key:
secret passphrase
```

- When the operations have been computed the user will be prompted for a file name in order to save their results. In this instance we will be using "***partf***" as our file name. When the file name is submitted, our program will append "***PublicKey.txt***" to the name thus resulting in the official output file of "***partfPublicKey.txt***". When this file is opened a key pair will appear, thus saving and ending the keypair operation.

g) Encrypt the private key under the given password and write it to a file [BONUS]

- The program prompts for a passphrase and then generate a private key from the passphrase and encrypts it and writes the encryption to an output file
- As an example we will be using the phrase "***secret passphrase***".

```
Enter a passphrase to encrypt the generated elliptic key pair private key:
secret passphrase
```

- Once the results have been computed the user will be asked to provide a filename to save the output. In this instance we will be using "***partg***" as our filename, once completed we will be left with the the filename "***partgPrivateKey.txt***".
  Our output file will result in the contents shown below.

```
308F763411752F2378A0BDB352D0EC8B1301CC864FD9E9640BD711CAED1947851AEF4CE9064F3FD15B68597F691E555A808D59BF5161F831A953D3A46FC8B4908
```

h) Encrypt a data file under a given elliptic public key file

- The program prompts for a data file and a public key file and writes the encrypted data file to an output file
- As an example we will be using the file "***test.txt***" (*provided in input file*) to generate our encrypted file.

```
Enter a file to encrypt under a given elliptic public key file:
test.txt
```

- In order to encrypt the date we need a public key of which we will use the public key "***partfPublicKey.txt***" from ***part f*** (*provided in input file*).
- After inputting both example data file and public key provide a name for your output data. In our case we will call ours "***parth***" which will be outputted as "***parthEllipticEncrypt.txt***".

The result of our example usages will appear as shown below.

```
00EF3CF3B7CEE76BCAEEC800EFD26F4F9D4E89FC93275D00C9203A0CEAA1D00D1F0DC355EC50F65AD9E3E45C670719C56F9C347188CDA0BEA32611E8A2DFFADD78250152E3E412C47E2640640BE
00A98B859A8B2E7396F69AC2FD13D797AFCE99CE179A5AF2150776EF34A817DB59D205DE5046F31EB48C6D2B1152F3CFB8A38EE4E259D972D640CB9B13331A66E9E401CF07EF92ED1EDA820A70FCA4F24E9FAE
003CEDA70B58339C2FB5B6FA0BD7060C1071BACC5D1037316C1B5C3F5C94CAC42A879E86FC0988D1B35CA60B5CC824B931164E62F95B5D7CFB6A5481E23EF1F5D3360012B9CD1981CB39FDE2F62DB28E66A9A6
```

i) Decrypt a given elliptic-encrypted file from a given password

- The program prompts for an encrypted data file and the passphrase used when generating a public key to encrypt the file, then the program writes the decrypted data file to an output file
- As an example, we will use the encrypted file "***parthEllipticEncrypt.txt***" and the passphrase "***secret passphrase***" which was used to generate a public key and encrypt the file. For convenience every file is always provided in the input folder.

```
Enter a file to decrypt under a given passphrase:
parthEllipticEncrypt.txt
Enter a passphrase:
secret passphrase
```

The result of our example usages will appear as shown below.

```
testing line 1
testing line 2
testing line 3
```

j) Encrypt/decrypt text input [BONUS]

- The program prompts for the user to either do an encryption or decryption
  - The encryption asks for text input and a public key hex and outputs the encrypted text to the console output
  - The decryption asks for an encrypted text input and passphrase and outputs the decrypted text to the console output
- As an example, we will use "***Hello world***" to encrypt the message with our public key "***partfPublicKey.txt***". The user however will need to copy and paste the contents of the public key (*length of 132*) into the terminal.
  The result of our example usages will appear as shown below.

```
Result:
016A2A0C4B48FE260E4DF5F9625667FE411475C98E8CEF2B4F7F9987F65D2
```

- Decrypting messages is easy too. Using our previously encrypted message "***016A2A0C4B48FE260E4DF5F96256…***" we can combine that with the passphrase we used generate the public key and to encrypt our message "***secret passphrase***", we can decrypt the message back to its original state.

```
Enter a message to decrypt:
016A2A0C4B48FE260E4DF5F9625667FE411475C98E8CEF2B4F7F9987F65D2
Enter the passphrase:
secret passphrase
```

The result of our example usages will appear as shown below.

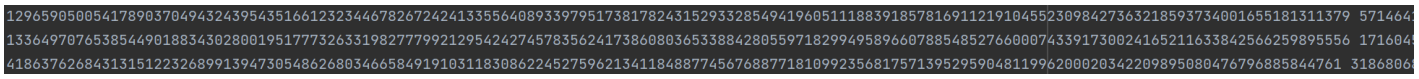
```
Result:
Hello world
```

k) Sign a given file from a given password and write the signature to a file

- The program prompts the user for an data file and a passphrase and writes the signature to and output file
- As an example we will be using the data file "***text.txt***" and the passphrase "***secret passphrase***" to be used to generate a unique signature.

```
Enter a file to sign:
test.txt
Enter a passphrase:
secret passphrase
```

- After which you are able to state an output for the generated digital, in our case we will be calling our output "***partk***", this will later be converted to be called "***partkSignature.txt***".
  The result of our example usages will appear as shown below.



l) Verify a given data file and its signature file under a given public key file

- The program prompts for a data file, a signature file, and a public key file and outputs to the console output whether the given data file and its signature file is verified or not
- As our final example on the program, we will be using the data file "test.txt", our digital signature "part
- kSignature.txt", and the public key used to generate our signature "partfPublicKey.txt" , in order to verify the authenticity of the sender. These have been obtained from previous **parts k** and **f**.

```
Enter a data file:
test.txt
Enter a signature file:
partkSignature.txt
Enter a public key file:
partfPublicKey.txt
```

- When the authenticity can be proven a message of "***This is verified***" is posted, else the message is "***This is NOT verified***".
  The result of our example usages will appear as shown below.

```
This is verified
```

m) Offer the possibility of encrypting a file under the recipient's public key and also signing it under the user's own private key [BONUS]

- We as a group chose not to implement this feature

Material Citations

- NIST Documentation: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf
- Elliptic Curve Slides (canvas) from Prof. Paulo Barreto: https://canvas.uw.edu/courses/1555586/files/91632405?wrap=1
- Guidance on elliptical generator and arithmetic https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc
- Used to obtain baseline values for Edwards Elliptical curve https://eprint.iacr.org/2013/647.pdf
- Used to guidance and referencing for elliptical point addition. https://fse.studenttheses.ub.rug.nl/10478/1/Marion_Dam_2012_WB_1.pdf
- Tiny_sha3 for referencing when implementing Keccak functionality: https://github.com/mjosaarinen/tiny_sha3