Cryptography Project Report Part 1 - Alex Trinh, Eugene Oh - TCSS 487 Spring 2022

Solution Description:

During the initial phases of doing this project, we encountered a great amount of challenges in regards to implementing Markku-Juhani Saarinen's C implementation of SHA3. Upon first looking at the code, it was incredibly unreadable. Eventually, we decided that it is not the top priority to understand every detail of this implementation but rather just to code in Java as closely as possible. Some functions such as ROTL64 in the C-implementation did not match up to anything in Java, so this required more research into these functions with the addition of implementing these functions in their own methods. Eventually, we were able to implement SHA3 using the C-implementation but without the certainty that this program works as intended. We then moved onto creating the CSHAKE and KMAC functions. For this, we based our implementation of these functions from the pseudocode provided from the NIST document. This also required us to implement additional methods, such as bytesToHex() and concat() in order to fully accommodate the pseudocode. For our last steps of implementation, we moved onto creating separate classes for the required functions as specified in the project requirements, such as encryption and decryption. Following the pseudocode in the NIST document, these classes use the previously implemented CSHAKE and KMAC to generate output. Finally, these classes are then called upon by our main driver class and are used in our command-line interface.

Known Bugs:

Currently, the program does not produce correct output when in comparison to the provided NIST Derived Functions examples. One of the bugs found was in left_encode where part of the outputted byte array was incorrect. This was fixed through using a left-shift operator (<<) on the given BigInteger. However, it was then found that bytepad() was also incorrect when comparing output to the examples. Since this method was derived from Professor Barreto's Presentation, we tried changing the arguments given to bytepad(), changed the implementation of encode_string(), and manipulated the arguments of cShake256() but was not successful. Unfortunately, this bug affects all of our results and there could be more bugs than this, so our output is incorrect. However, all intended operations to fulfill part one requirements (encryption, decryption, generating hashes) works as intended.

Cryptography Project Report Part 2 - Alex Trinh, Eugene Oh - TCSS 487 Spring 2022

Solution Description:

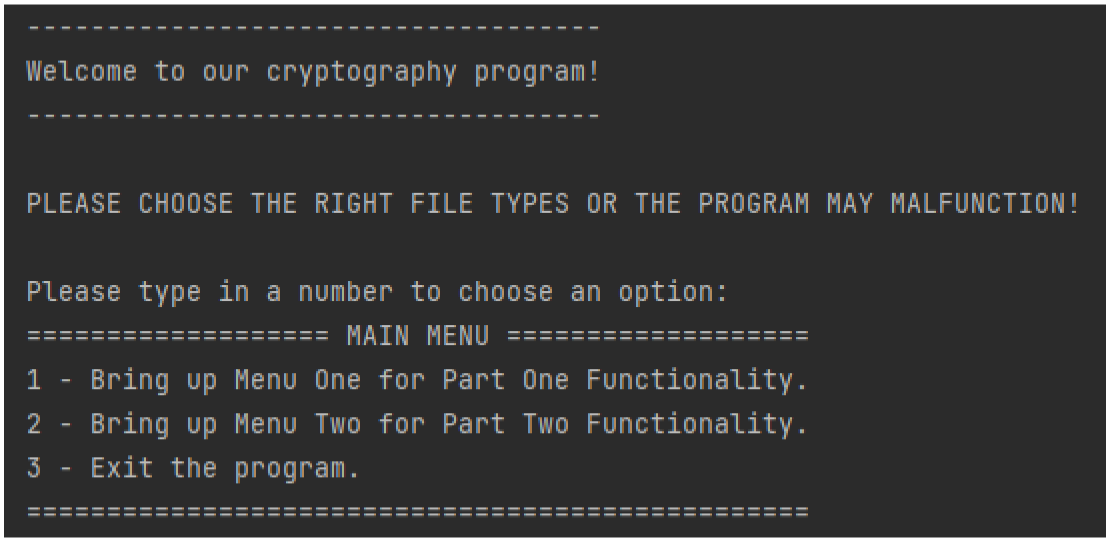
Before starting the second part, we had to go back in attempt to fix the bugs that lingered from the first part. After a couple of hours, we discovered minor errors when comparing our keccak function to the C-Implementation, such as an "i" being a "1" in our code. Some other changes required minor restructure of implementation, but eventually, our program passed the test cases taken from the NIST document. Upon starting the second part of this assignment, we created the Java class representing a point on the elliptic curve. Following the specification sheet, we created a method regarding required constructors and behaviors. When it came to the methods regarding arithmetic such as multiplication and adding two points, we had to refer to the classroom slides and the pseudocode/equations given in the sheet and followed them as closely as possible. With this, we had a functioning point class. For the second half of this part, we implemented the required functions using this point class, such as encryption/decryption and signing/verifying signatures. To follow the pseudocode in the specification sheet, we created another class that represented a separate cryptogram regarding these functionalities. This class contains the variables in the pseudocode, such as "Z", "c", and "t". With this, we were able to fully follow the pseudocode and implemented each required function successfully. Lastly, we finished out the project by adding menu options in our command-line based program to do each of these functions, added comments to each class, and other general code clean-ups.

Known Bugs:

Currently, verifying signatures for the second part of the programming project does not work. The user can enter in any signature or key file for verifying signatures and it will result in unsuccessful outputs. We have tried changing the implementation for verifying and generating signatures but were unsuccessful. However, every other function required should work properly.

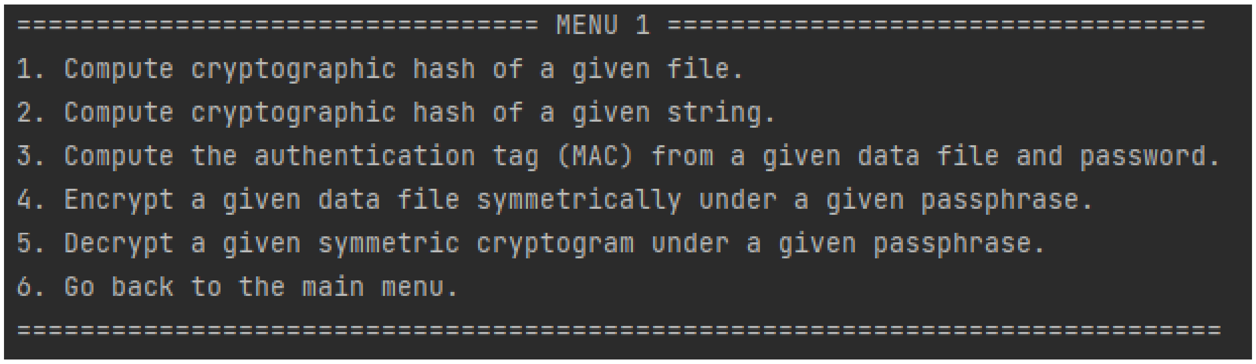
User Instructions

This program takes user commands through the command-line. The class "App.java" must be ran in order to run the program. When ran, the user can select a desired option through typing the number correlated with the option:

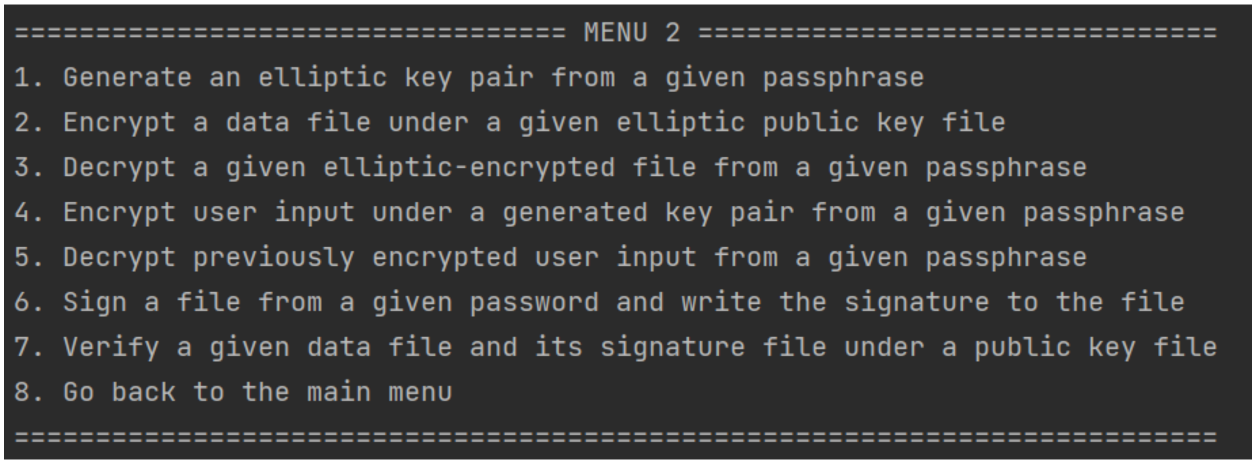


```
------------------------------------
Welcome to our cryptography program!
------------------------------------

PLEASE CHOOSE THE RIGHT FILE TYPES OR THE PROGRAM MAY MALFUNCTION!

Please type in a number to choose an option:
=================== MAIN MENU ===================
1 - Bring up Menu One for Part One Functionality.
2 - Bring up Menu Two for Part Two Functionality.
3 - Exit the program.
=================================================
```

Menus One and Two look as such:

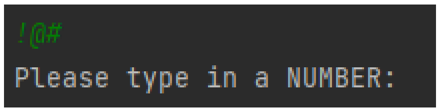


```
================================= MENU 1 ==================================
1. Compute cryptographic hash of a given file.
2. Compute cryptographic hash of a given string.
3. Compute the authentication tag (MAC) from a given data file and password.
4. Encrypt a given data file symmetrically under a given passphrase.
5. Decrypt a given symmetric cryptogram under a given passphrase.
6. Go back to the main menu.
===========================================================================
```

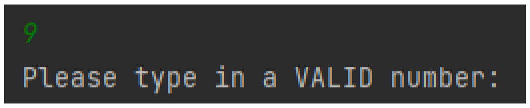


```
================================= MENU 2 =================================
1. Generate an elliptic key pair from a given passphrase
2. Encrypt a data file under a given elliptic public key file
3. Decrypt a given elliptic-encrypted file from a given passphrase
4. Encrypt user input under a generated key pair from a given passphrase
5. Decrypt previously encrypted user input from a given passphrase
6. Sign a file from a given password and write the signature to the file
7. Verify a given data file and its signature file under a public key file
8. Go back to the main menu
==========================================================================
```

The program when traversing menus does not accept any input that is not an integer or is within the range of options, as the following messages will be displayed in such case:

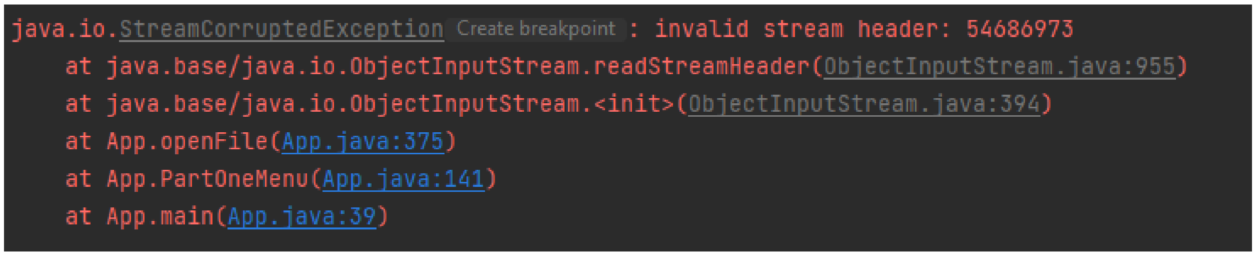


```
!@#
Please type in a NUMBER:
```



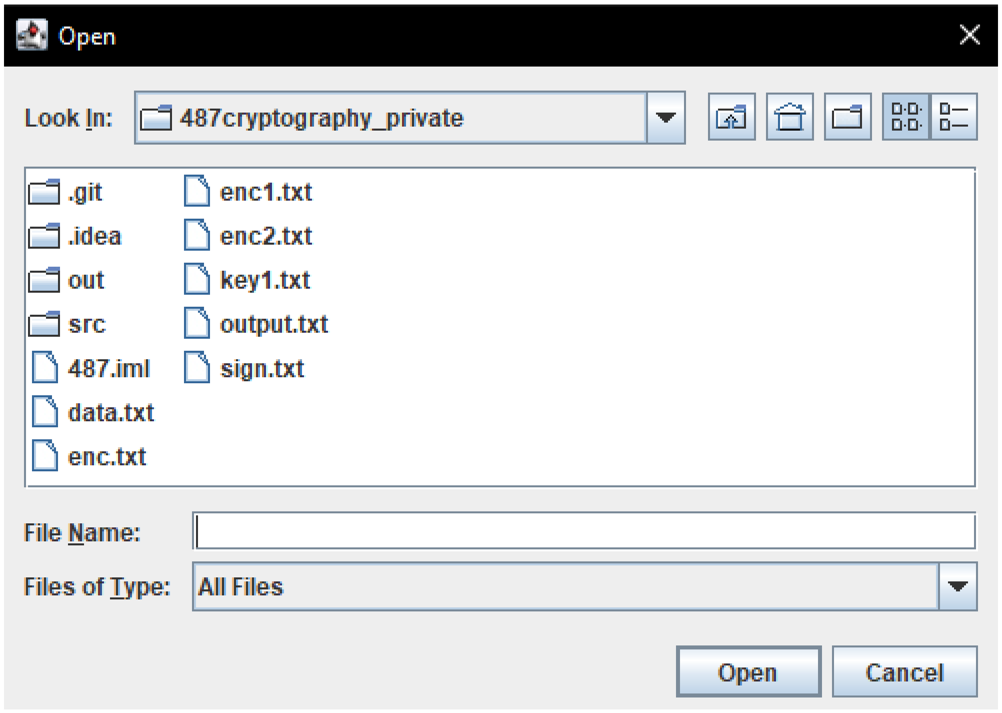
```
9
Please type in a VALID number:
```

This program WILL throw errors if given the wrong file types, such as giving a plain text data file when the program expects an encrypted file. Here is an example of this occurrence:



```
java.io.StreamCorruptedException Create breakpoint : invalid stream header: 54686973
    at java.base/java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:955)
    at java.base/java.io.ObjectInputStream.<init>(ObjectInputStream.java:394)
    at App.openFile(App.java:375)
    at App.PartOneMenu(App.java:141)
    at App.main(App.java:39)
```

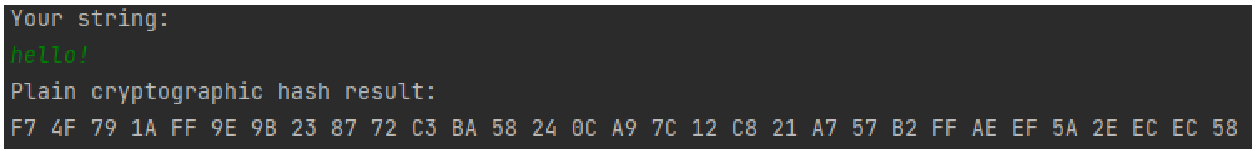
When prompted to enter a password, please enter in single word passphrases or else decryption using these passphrase may not function as intended. Also, some options, such as encryption/decryption, will also display a file menu that looks as such:



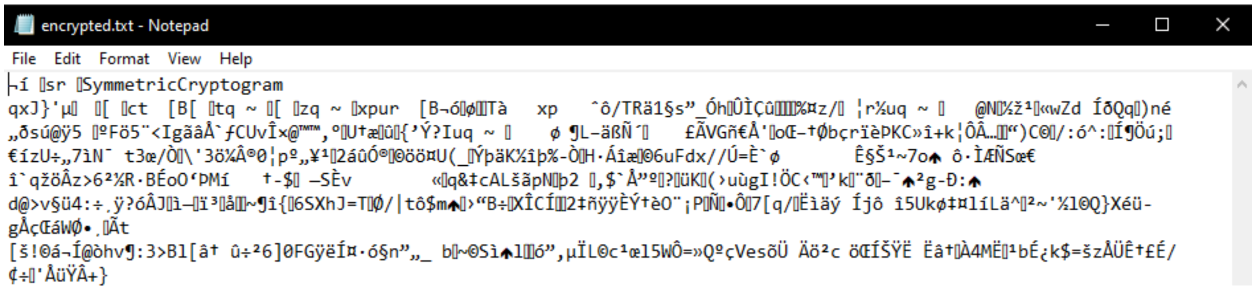
If this menu does not pop up, please restart the program and try again. This may happen on rare occurrences. The user can also press cancel at any time and a message should pop up as such:

```
Please choose a file next time.
```

The user must enter in the ".txt" extension in all output that they created and all input files must be in the same extension or else these files may be inaccessible. In normal circumstances, output can either appear to the console or in an output file:

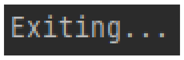


```
Your string:
hello!
Plain cryptographic hash result:
F7 4F 79 1A FF 9E 9B 23 87 72 C3 BA 58 24 0C A9 7C 12 C8 21 A7 57 B2 FF AE EF 5A 2E EC EC 58
```



```
┣í ▯sr ▯SymmetricCryptogram
qxJ}'µ▯  ▯[ ▯ct  [B[ ▯tq ~ ▯[ ▯zq ~ ▯xpur  [B¬ó▯ø▯▯Tà   xp   ˆô/TRä1§s”_Óh▯ÛÌÇû▯▯▯%¤z/▯ ¦r½uq ~ ▯   @N▯½ž¹▯«wZd ÍðQq▯)né
„ðsú@ÿ5 ▯ºFö5¨<IgãâÅ`ƒCUvÎ×@™™,º▯U†æ▯û▯{’Ý?Iuq ~ ▯    ø ¶L–äßÑ´▯    £ÃVGñ€Å'▯oŒ–†ØbçrïèÞKC»î+k¦ÔÂ…▯▯“)C©▯/:ó^:▯Í¶Öú;▯
€ízU÷„7ìN¯ t3œ/Ò▯\'3ö¼Â®0¦pº„¥¹▯2áûÓ®▯©ööᐤU(_▯ÝþäK½îþ%-Ò▯H·Áîæ▯©6uFdx//Ú=È`ø        Ê§Š¹~7o⬆ ô·ÌÆÑSœ€
î`qžöÂz>6²½R·BÉoO‘ÞMí    †-$▯ —SÈv        «▯q&‡cALšãpN▯þ2 ▯,$`Å”º▯?▯üK▯(›ùgI!ÖC‹™▯’k▯¨ð▯–¯⬆²g-Ð:⬆
d@>v§ü4:÷¸ÿ?óÂJ▯ì–▯ï³▯å▯▯~¶î{▯6SXhJ=T▯Ø/|tô$m⬆▯›“B÷▯XÎCÍ▯▯2‡ñÿÿÈÝ†èO¨¡P▯Ñ▯•Ô▯7[q/▯Ëìäý Íjô î5Ukø‡¤1íLä^▯²~'½1©Q}Xéü-
gÅçŒáWØ•¸▯Ãt
[š!©á¬Í@òhv¶:3>Bl[â† û÷²6]0FGÿëÍ¤·ó§n”„_ b▯~©Sì⬆l▯▯ó”,µÏL©c¹œl5WÔ=»QºçVesõÜ Äö²c öŒÍŠŸË Ëâ†▯À4MË▯¹bÉ¿k$=šzÅÜÊ†£É/
¢÷▯'ÅüŸÂ+}
```

The name at the top describes the type of output file that is contained within the text file. Finally, when exiting the program, it will display a simple message as such:



```
Exiting...
```