Duncan Claypool

Arie Slobbe

Noah Miannan

**COMP-123-Ernst: Core Concepts in Computer Science**

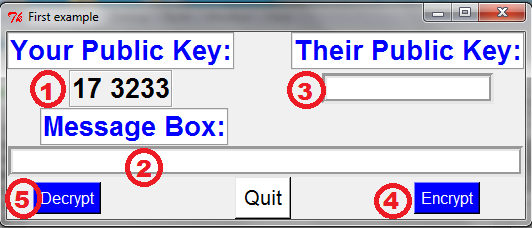
**Final project**

**RSA Encryption Algorithm with GUI**

*User Manual*

This program, a type of RSA encryption and decryption software, is used to send secure coded messages from one person to another. It functions via the creation of keys, one required for encoding, and another required for decoding. If someone is looking for a message to be sent to them, they tell the sender their public key (used to create an encoded message), but not the associated private key (used to decode messages that were created using that public key). The sender then inputs their message and runs the RSA encryption software using the public key of the recipient. The result is a list of integers that can only be decoded with the correct private key associated with that specific public key. The recipient receives this coded list of numbers, inputs it into his user interface and presses the decrypt button. The software then decrypts that message, returning it to its original format. The result is a secure line of communication between individuals using this software.

This specific iteration of this software can be used by following several simple steps. First, open the file name “FileName.py” in the Wingware Python Integrated Development Environment. This will open the source code which contains the software. Next, click the run button (the small green triangle located at the top of the screen) in order to initiate the program. A window will appear displaying (1) your public key, boxes to input (2) a message and (3) recipient public key, as well as buttons to (4) encode or (5) decode the inputted message.



In order to encode a message to be sent to someone else, you must first request from them their public key. This you will input in the text box on the top right hand corner of the window. Note, the key must be typed as two integers separated by a space, else the program will be unable to register it correctly. Then, input your message into the message box and press the *Encode* button. Your coded message will appear outside the window, printed inside the Python Shell, at the bottom right hand corner of the Python script. This allows you to, using your prefered channel of communication, directly copy the message and send it to your recipient. In order to decode an encrypted message that is addressed to you, you simply copy it into the message box and press the *Decode* button. A pop up window will then appear with the message in its original form. This is illustrated below with the following two examples.

The first test example takes the message “Sorry! I’m late…” and walks through the process of encryption and decryption. The first user requested the public key of the second and, once given, inputed that into the *Their Public Key:* input box. Next, they typed their message into the message input box and pressed the *Encrypt* button. The computer printed the following list of values (the encrypted message): 569 995 1707 1467 3381 1621 1467 995 1707 1467 3381 1621 1433 1621 1905 701 1248 995 1621 1467 1905 3179 1707 1621 1467 995 1707 1467 1621 2591 701 2518 2704 1707 2931. This user then copied and sent this message to the second user. This user then pasted this list of integers (the encoded message) into the message input box and pressed the *Decrypt* button. A pop up window then appeared with the original message (Sorry! I’m late…) displayed within. The exact same process was followed for the second example.

The second example uses the imputed message “I hope we’re getting an A..” and undergoes the identical process as the first example. This process started with the first user requesting from the second user their public key, which was subsequently typed into the *Their Public Key:* input box. Next, the user typed their chosen message into the message entry box, and pressed the *Encrypt* button. The result was the following list of values printed into the Python shell: 1433 1621 1905 701 1248 995 1621 2591 995 2433 2518 995 1621 2552 701 2129 2129 1579 1621 2552 995 1467 1621 1579 2129 1621 1253 1621 1528 701 2518 1621 1467 1905 3179 1707 2489. The second user then inserted this, coded, message into the message entry box and pressed the *Decrypt* button. The program then displayed the original message in a pop up window.

*Contents of Program*

This program runs through the updating of global variables and the reliance of one master function upon a series of smaller, supporting, functions. At the start of the script, all pertinent global variables were defined to be zero, creating a frame in which functions can store information via the updating of these variables. The program then consists of a series of short function, each which performs a specific task towards the overall goal of the cipher (decryption, generating a specific key, etc.). Instead of creating a complicated framework of functions calling other functions, these functions operate through the adjusting and calling of global variables. This allows all the subfunctions to remain independent, as well as reliant upon minimal inputs. The program itself is initiated under one master function which, within it, calls to the fore all the previously defined sub functions. This structure simplified and expedited the process of writing and testing such a complex program.

The script was implemented and tested in a step-wise manner, starting with the key generation function and ending with the creation of the user interface. The first step in implementation was to break down the complicated encryption process into smaller, more manageable, components, including: key generation, encryption, decryption and the guided user interface (GUI). With our road map thus sketched out, we proceeded to work through each of these components, in the order shown above, splitting them into sub functions to simplify them. Upon completion of each sub function, we ran a test upon the function using hypothetical inputs, testing both if the function operates correctly by itself, and if it operates correctly within the context of the program as a whole. In so doing, we were able to alleviate errors as we went, allowing us to locate and remedy them with relative ease. The result of this process is a completed program that runs smoothly, with no known bugs.

Overall, with the professors guidance, the creation of the complicated RSA algorithm went better than expected and we were able to code the cipher with fewer hurdles than we had initially anticipated. What we had not expected, however, was the amount of time and effort required to program the supporting GUI. If we had more time, we would add a button to generate a new public key, a host of error messages to prompt the user to proper syntax and an entry allowing the user to input an old public key. Additionally, had we the programing skills, we would add our own channel of communication to directly link the two users communicating with our program. Looking back, the experience was an enjoyable challenge that encourages us to continue our studies in the field of computer science. We feel gratified to have successfully completed a such a useful and complex program.

**Bibliography**

"Algorithm Implementation/Mathematics/Extended Euclidean Algorithm." *- Wikibooks, Open Books for an Open World*. N.p., n.d. Web. 01 May 2014.

“Introduction to Algorithms” T. H. Cormen et al. M.I.T Press 2001