KAVE_Suite documentation

§1. Application specifications

The application '**KAVE_Suite**' is an interactive GUI for encrypting and decrypting text data using the AES-128 cryptographic algorithm and sealing/unsealing it in an image using the LSB algorithm. The implementation and execution of the program was done on Windows 11 OS, using PyCharm IDE Community Edition v. 2021.1.3. In addition, all necessary libraries and packages are required to be installed:

- **pip install cryptography**
- **pip install docx2txt**
- **pip install pillow**
- **pip install opencv-python**
- **pip install numpy**

The **cryptography** package is a Python library for encrypting data in various ways, including symmetric encryption, which the KAVE_Suite application uses. More specifically, the Fernet module (cryptography.fernet) is used, which has the ability to generate pseudo-random keys for data encryption. In summary, the following steps are followed within the module to generate the ciphertext:

1. The reference time (timestamp) is recorded, which corresponds to the number of seconds that have passed since January 1, 1970 until the moment our encryption starts.
2. A pseudo-random initialization vector is generated through the module's os.urandom() function using parameters of our operating system and in such

a way that the numbers are as "random" as possible. Numbers are 128 bits long.

3. This is followed by binary processing of the plain text by padding those "incomplete" 128-bit blocks with extra bits to produce full blocks of 128 bits, according to the PKCS #7 algorithm. This process is called "padding" and is done so that the symmetric cryptographic algorithm AES-128 in CBC-mode can be efficiently applied to the data, in accordance with what has already been mentioned. AES-128 encrypts the data with the help of a key, here generated pseudorandomly by the generate_key() function of the fernet module
4. The version number of the fernet module used is stored. This is specific and equal to 128 in the decimal system.
5. As a means of verifying the integrity and correctness of the message, the timestamp, module version, initialization table, and cipher text are given as inputs to a SHA-256 function that produces an HMAC (Hash-based Message Authentication Code) token.
6. All of the above are arranged in order and form the fernet token, i.e. the final encrypted text.

The **doc2txt** package allows editing and converting .doc, .docx text file data to .txt file data. The **pillow** and **opencv** packages offer image editing capabilities. The **numpy** package allows, among other things, processing of multidimensional objects. The images used below for demonstration are .png, and the need to respect this condition is noted.

Tkinter package was used to reproduce the GUI and **PIL** Python library for image reading and manipulation. The code contains comments to fully explain the various functions performed by its sectors. The GUI was designed to be simple and presentable, offering the necessary interaction with the user.

§1. Application functions

Below we will demonstrate the operation of 'KAVE_Suite' for encryption - sealing and decryption - decryption of text data in images.

1] Opening the application, we see the following initial window, where we can choose sealing ('**Encode text**') or unseal ('**Decode image**'). In this first part we will present the option '**Encode text**':

Figure 1.1

Main window of KAVE_Suite

2] choosing '**Encode text**' button appears '**Upload PNG image**' (Figure 1.2) through which we will select the image.**png** which we will use as our message embed file (Figure 1.3):

Figure 1.2

Select 'Encode text'

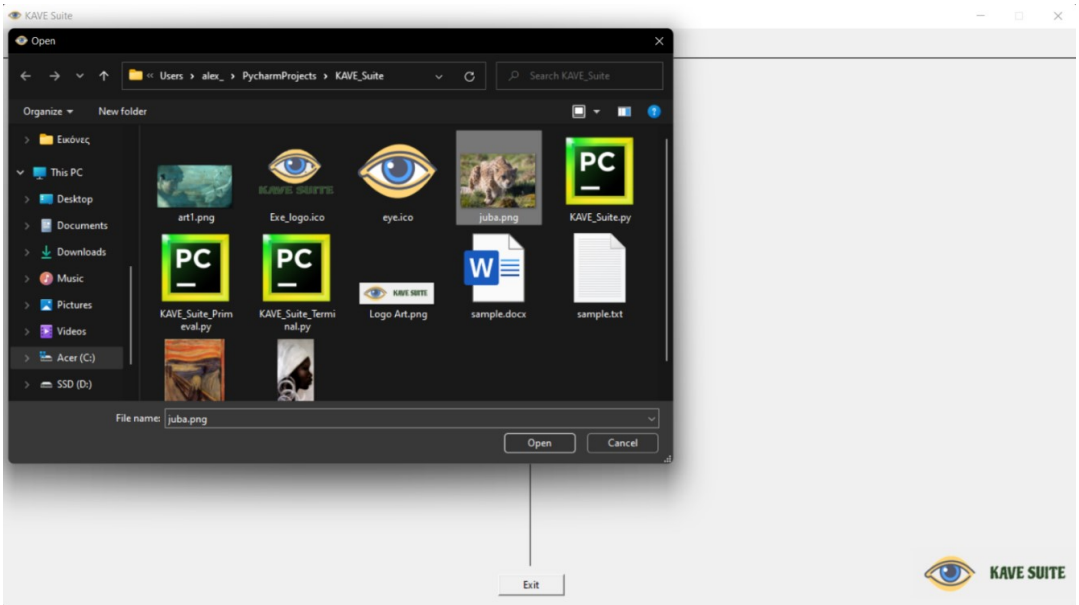

Figure 1.3

Image selection

3] After choosing the image we want, its path and file name appears as a confirmation to the user, in case he chooses wrong or changes his mind about his choice. In addition, the capacity of the image in terms of the encrypted data it can hide within its structure, is presented. The user can repeat the process by selecting again '**Upload PNG image**' without restarting the application, if they wish.

4] 2 options appear (Figure 1.4): the first corresponds to the user entering text via the keyboard ('**Encode text from user input**'). By pressing this button, 2 fields appear: the first accepts the **message** that we want to hide within the image, while the second one gets the **name** of the **new** image that will be produced in the folder where we have the application saved. We must be careful to enter the full name of the new image, including the file extension (.png). If everything is filled in correctly, pressing '**OK**' we

get a success message (Figure 1.5). Figure 1.6 shows the initial contents of our workspace, while Figure 1.7 shows that the new image with the encrypted and watermarked text, but also one key file(.key). As an example, regarding the execution times of the data encryption and decryption process in the image, for data of 1000 characters it took 0.12047 s, while for 10000 characters it took 0.254664 s. Below we will explain the use of the last .key file:

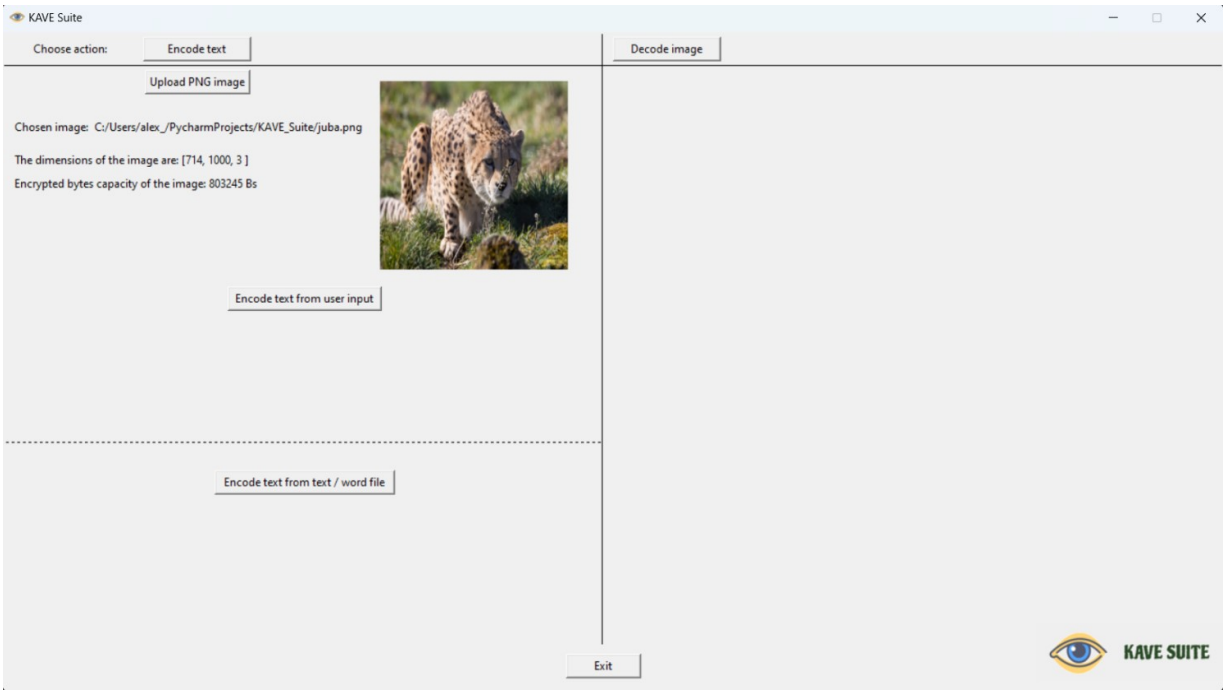

Figure 1.4

Two data options: user input text or .doc, .docx, .txt file data

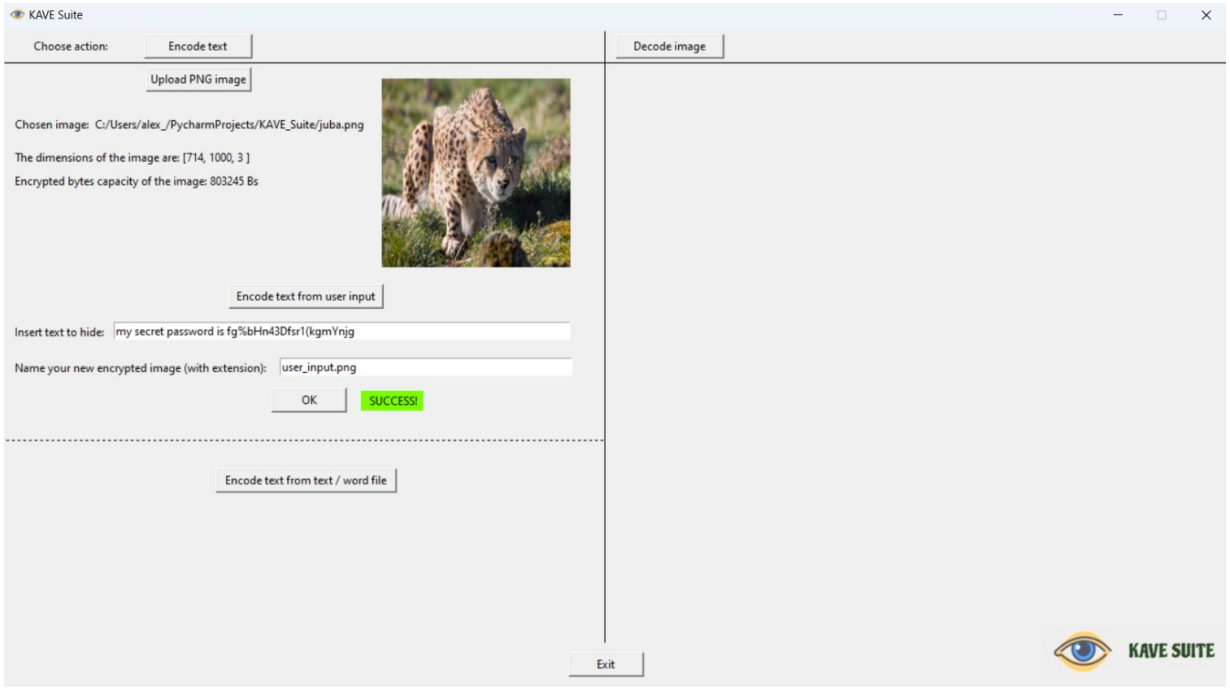

Figure 1.5

Correct completion of fields when entering data from the user's keyboard and message of successful data encryption and sealing

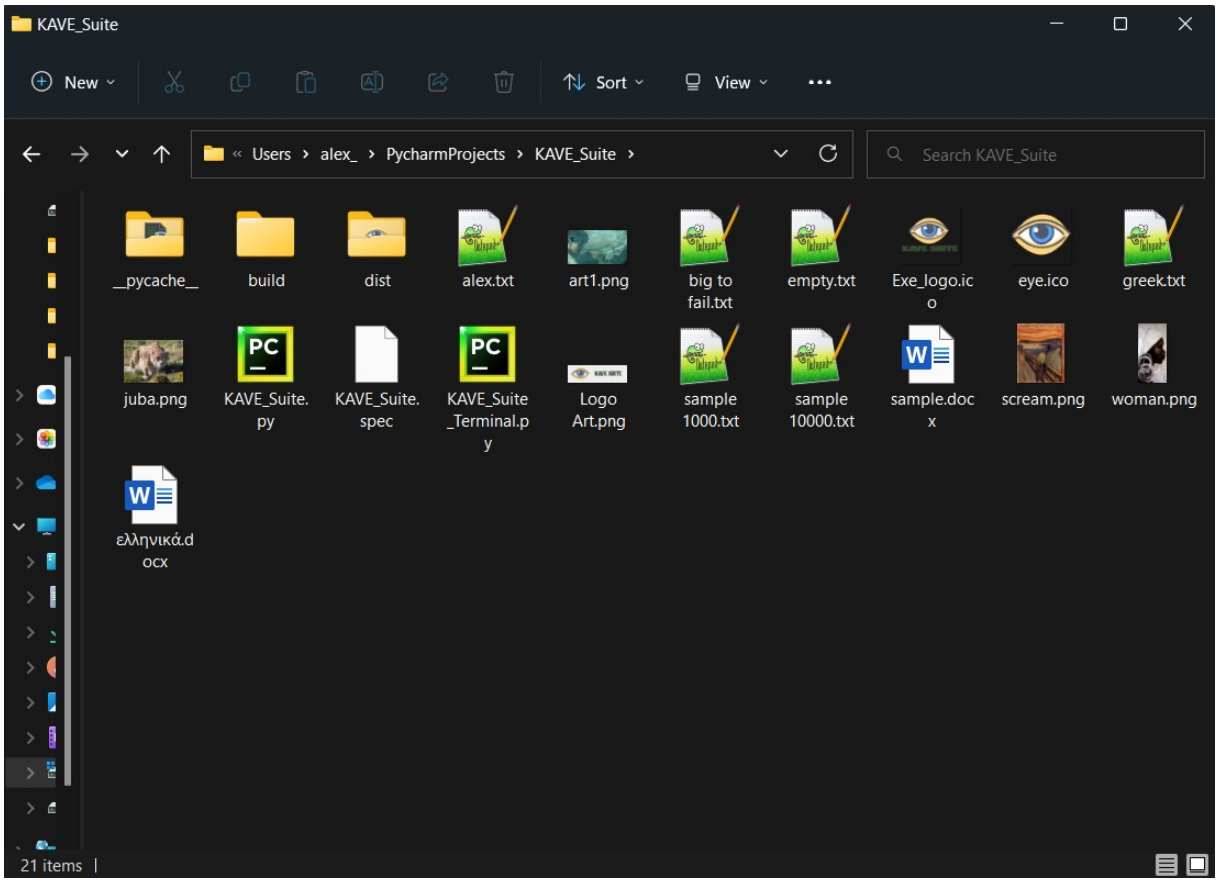

Figure 1.6

Workspace contents before the process of encrypting user text and watermarking it to a new image

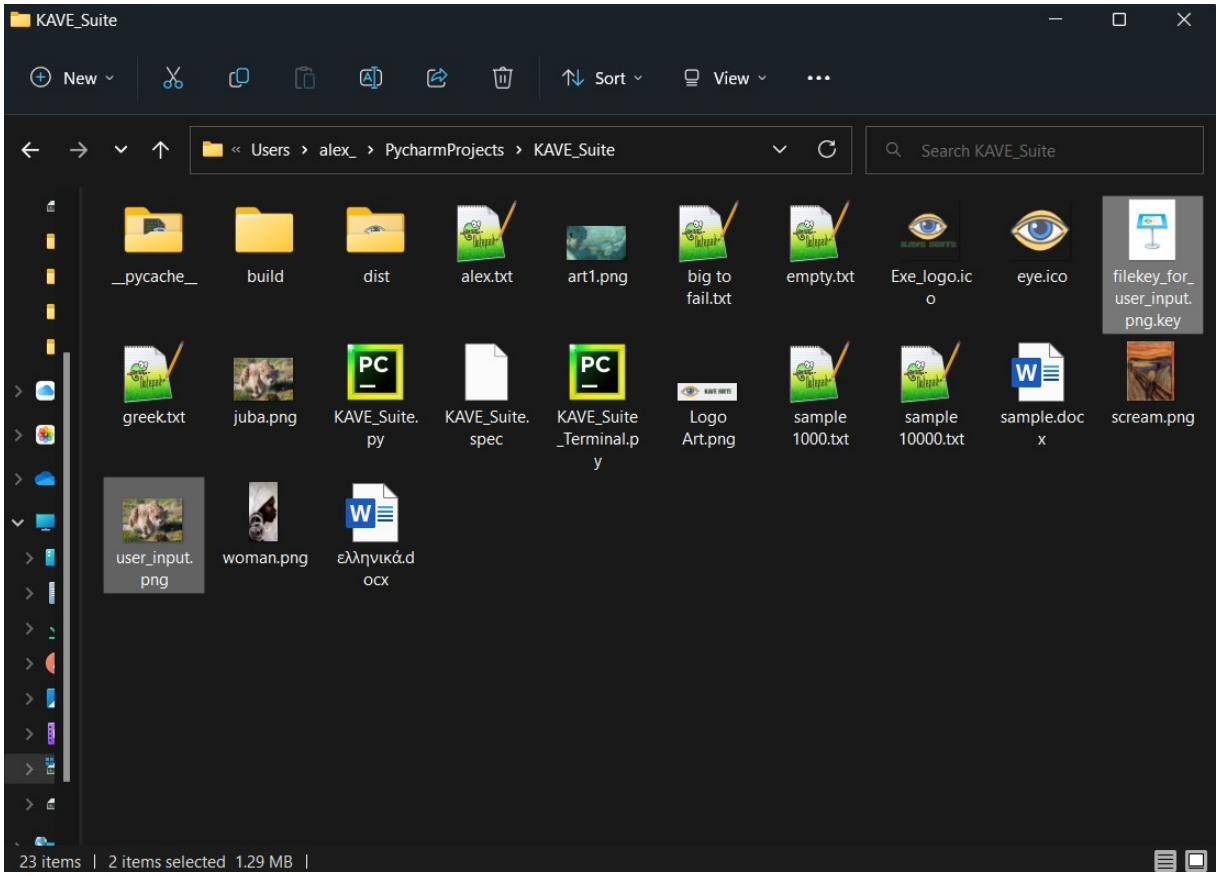

Figure 1.7

Workspace contents after the process of encrypting the user's text and sealing it to a new image

5] In case the user enters a new image name but omits to enter text to be encrypted and sealed, an error message appears and the application waits for text to be entered in the corresponding field (Figure 1.8). Greek, Latin, lowercase and uppercase characters, symbols and numbers are supported (generally reading data in Unicode encoding). The only limitation that exists is the name of the new images, as Greek characters are not supported. It will produce the correct image with the hidden and encrypted data,

however when we try to decrypt it, the application will not be able to read its name and will not display a result. But we can rename the generated image in Latin characters, to import the key file and everything will work normally.

6] The application displays the necessary degree of interaction with the user in case he fails to enter data, such as the name of the new image or the message itself to be encrypted and sealed. For example, in Figure 1.8, if the user enters **designation** for the new image but **not processing data**, the application does not proceed and waits until the process is executed correctly, displaying corresponding error messages. The following Figures show additional diagnostic messages of incorrect actions or omissions that the user may make:

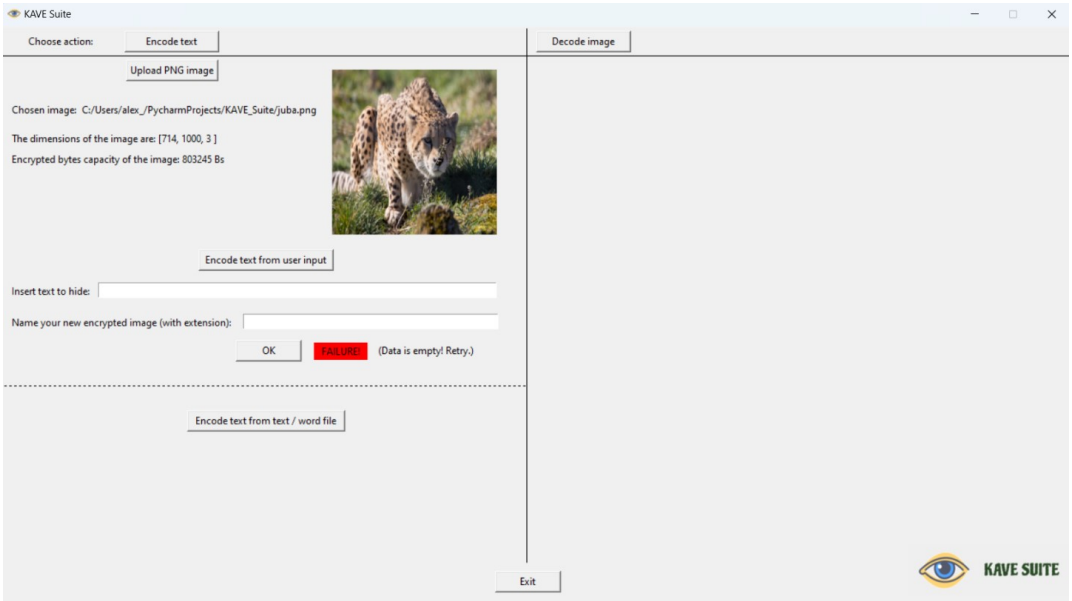

Figure 1.8

Failure message if text field is empty

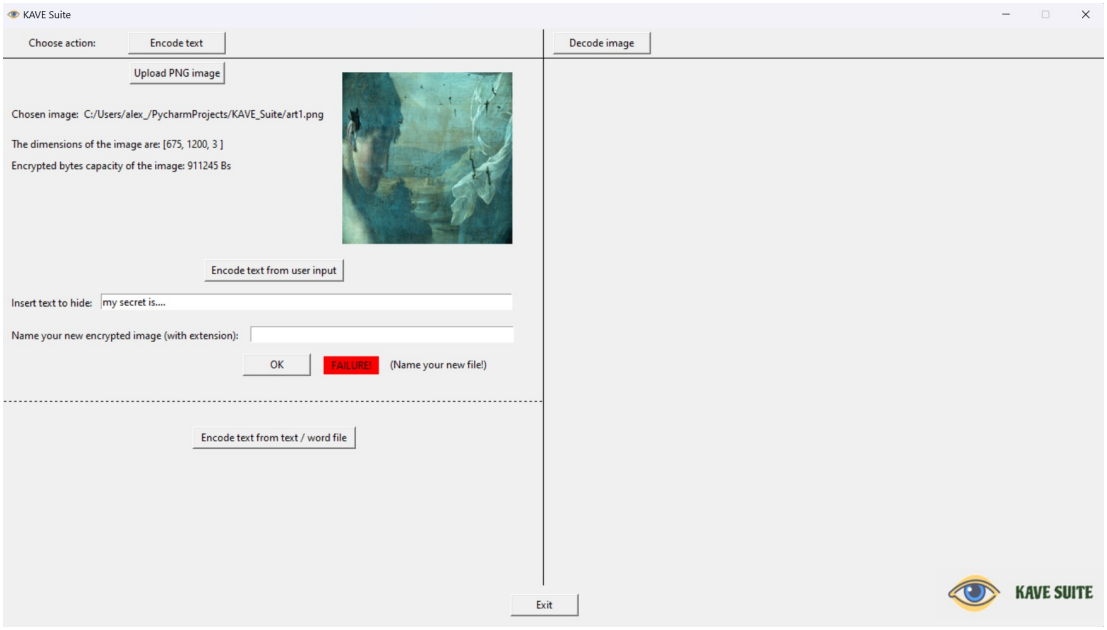

Figure 1.9

Failure message if the new image file name is omitted

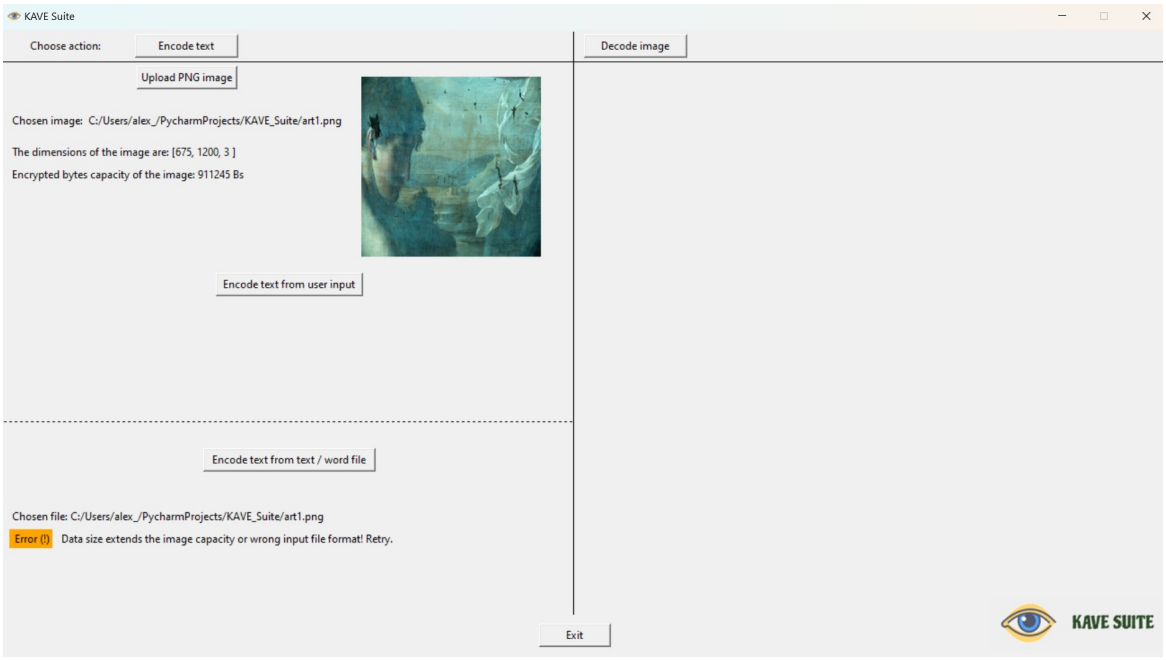

Figure 1.10

Failure message if too large text file or wrong file type selected (eg another image)

7] The application allows the parallel execution of various functions. That is, the user can select an initial image and seal ciphertext produced from data entered by the user, or data from a file.doc,.docxthe.txt. Here we present the results when we select '**Encode text from text/word file**', after we have first selected an initial image from our files (Figure 1.11). First, a window will appear to select the file from which we want to extract, encrypt and seal data (Figure 1.12). Figure 1.13 shows the contents of the .txt file we selected. Greek, Latin, lowercase and uppercase characters, symbols and numbers are supported (generally reading data in Unicode encoding). The only limitation that exists is the name of the new images, as Greek characters are not supported. It will produce the correct image with the hidden and encrypted data, however when we try to decrypt it, the application will not be able to read its name and

will not display a result. But we can rename the generated image in Latin characters, to import the key file and everything will work normally.

Figure 1.11

Select image in initial window and select 'Encode text from text/word file'

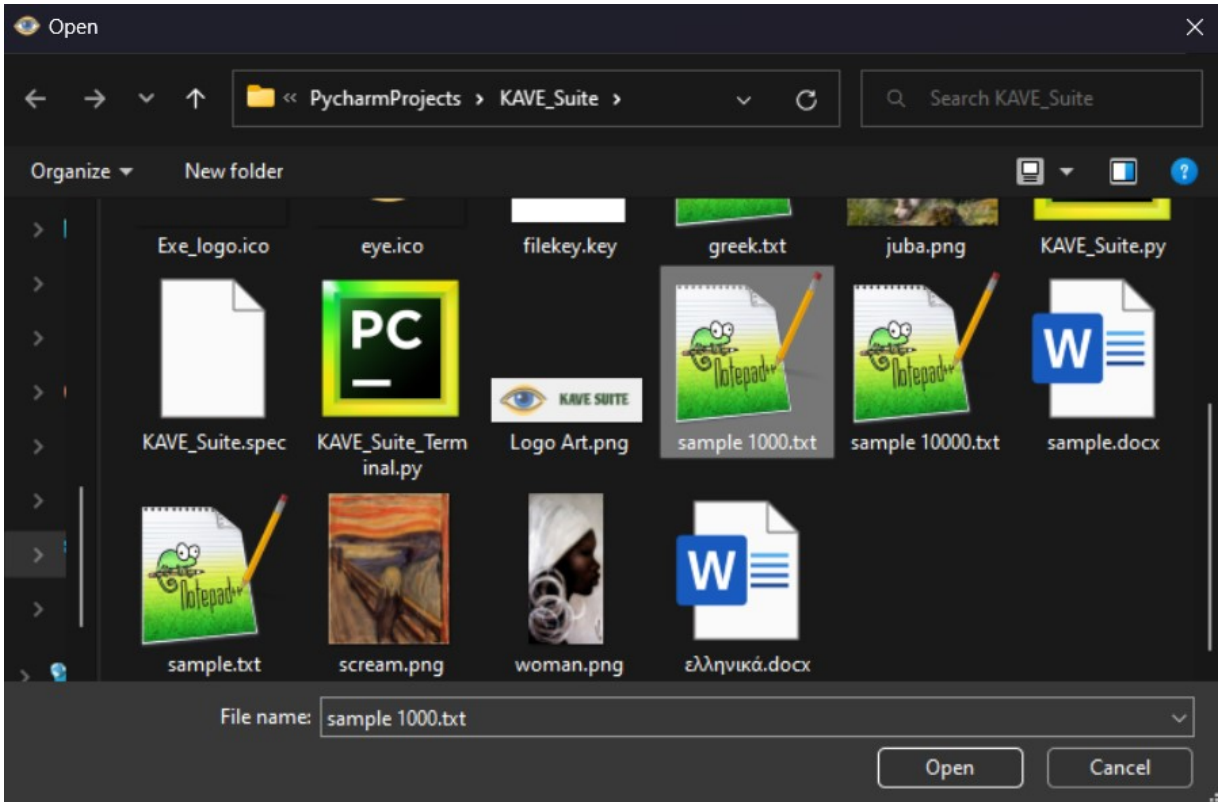

Figure 1.12

Select .txt file

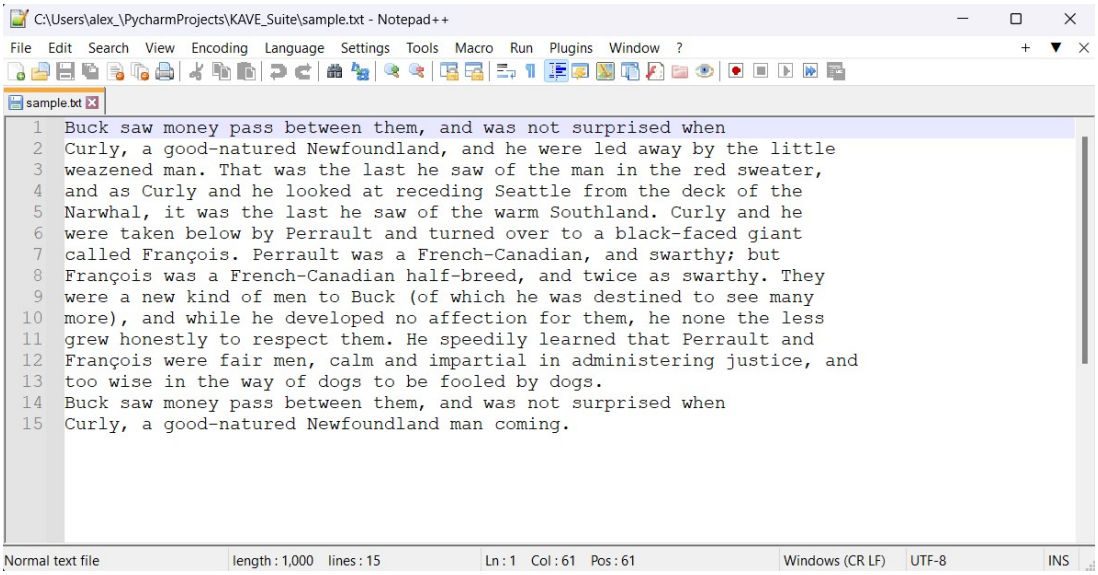

Figure 1.13

Contents of selected .txt file

8] After selecting the data file we want, we name the new image that will be produced. Then, pressing '**Okay**' we receive a success message 'SUCCESS' (Figure 1.14). In Figure 1.15 we see the new files created in our workspace, these being the new image and the .key file:

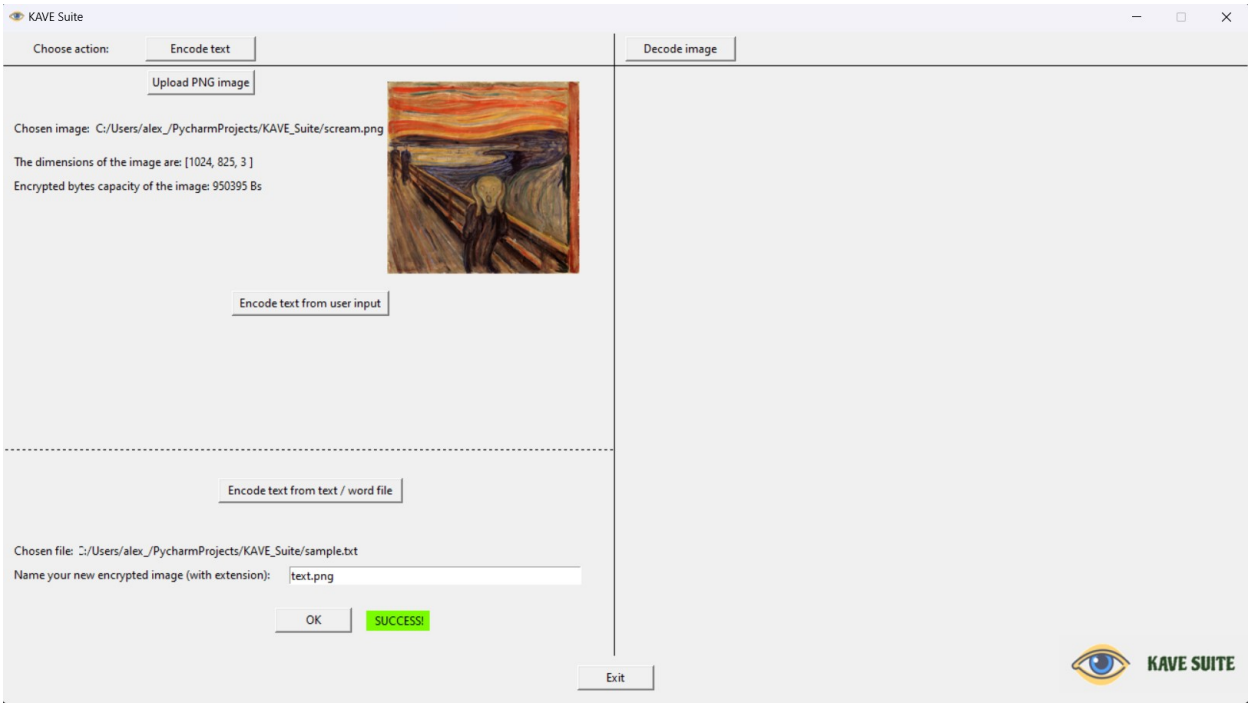

Figure 1.14

Successfully encrypt and seal data from the .txt file

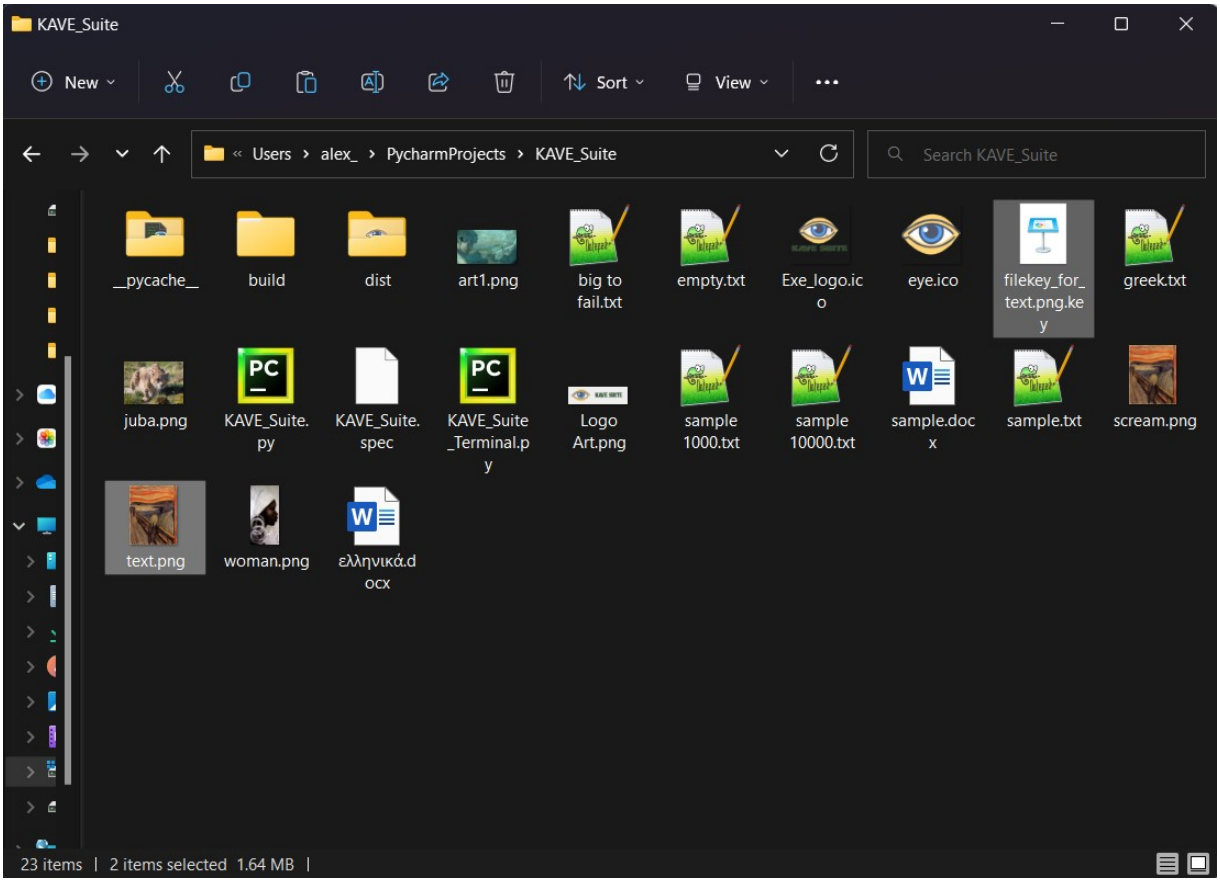

Figure 1.15

New files in our workspace

9] If instead of a .txt file we choose a .doc or .docx, the results are similar. In Figure 1.16 we select the file from which we will extract the text data. Greek, Latin, lowercase and uppercase characters, symbols and numbers are supported (generally reading data in Unicode encoding). The only limitation that exists is the name of the new images, as Greek characters are not supported. It will produce the correct image with the hidden and encrypted data, however when we try to decrypt it, the application will not be able to read its name and will not display a result. But we can rename the generated image in

Latin characters, to import the key file and everything will work normally. In our example, the file 'sample.docx' contains the same data as the .txt file we selected earlier. In Figure 1.17 we see the message for successful data encryption and sealing. In Figure 1.18 we see the new files in our workspace.

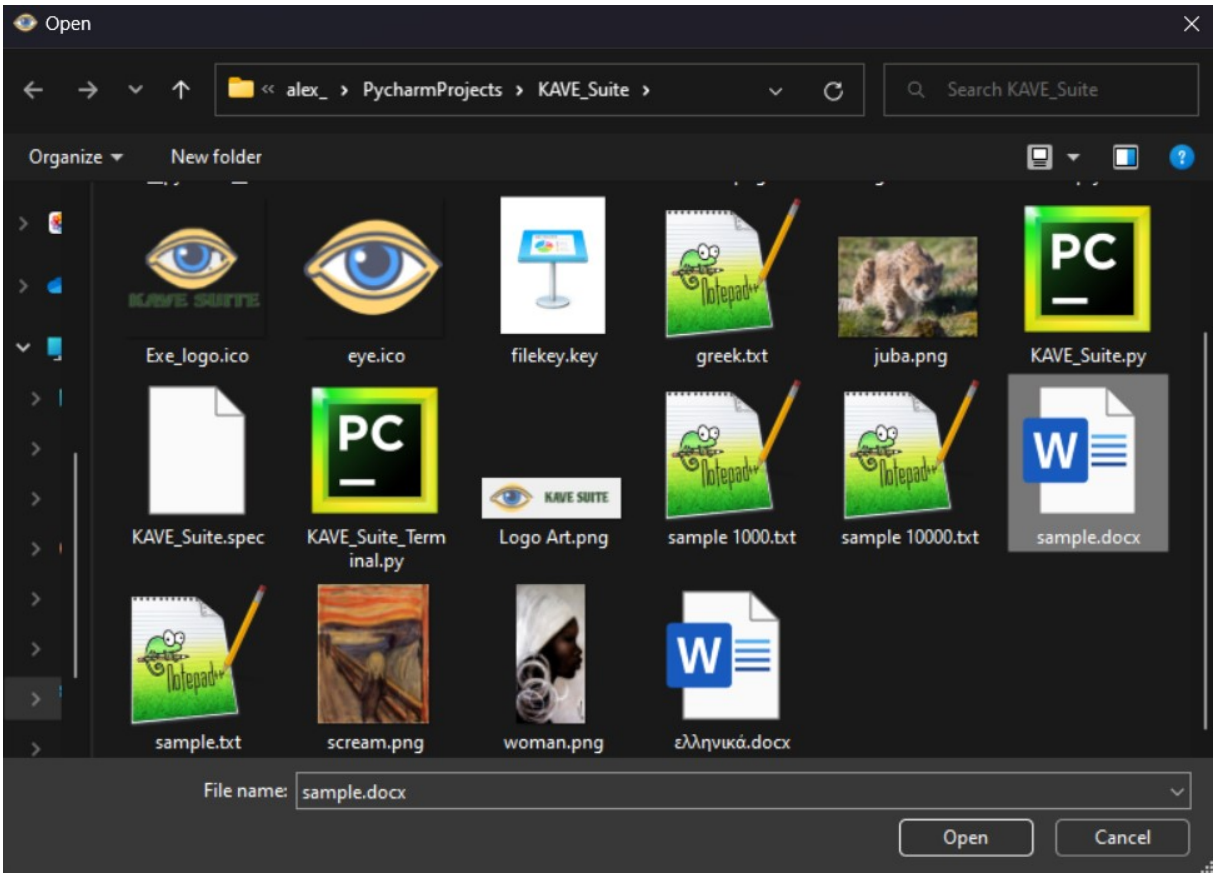

Figure 1.16

Select .doc, .docx file

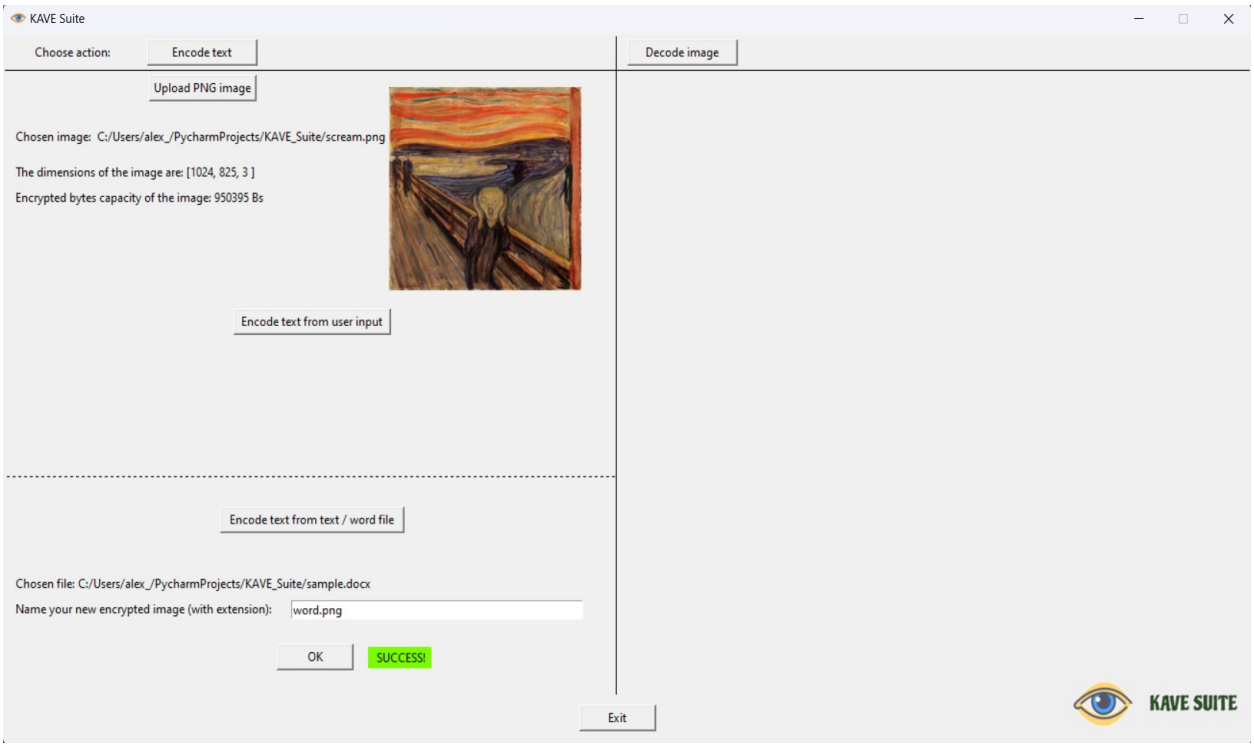

Figure 1.17

Successfully encrypt and seal data from .doc/.docx file

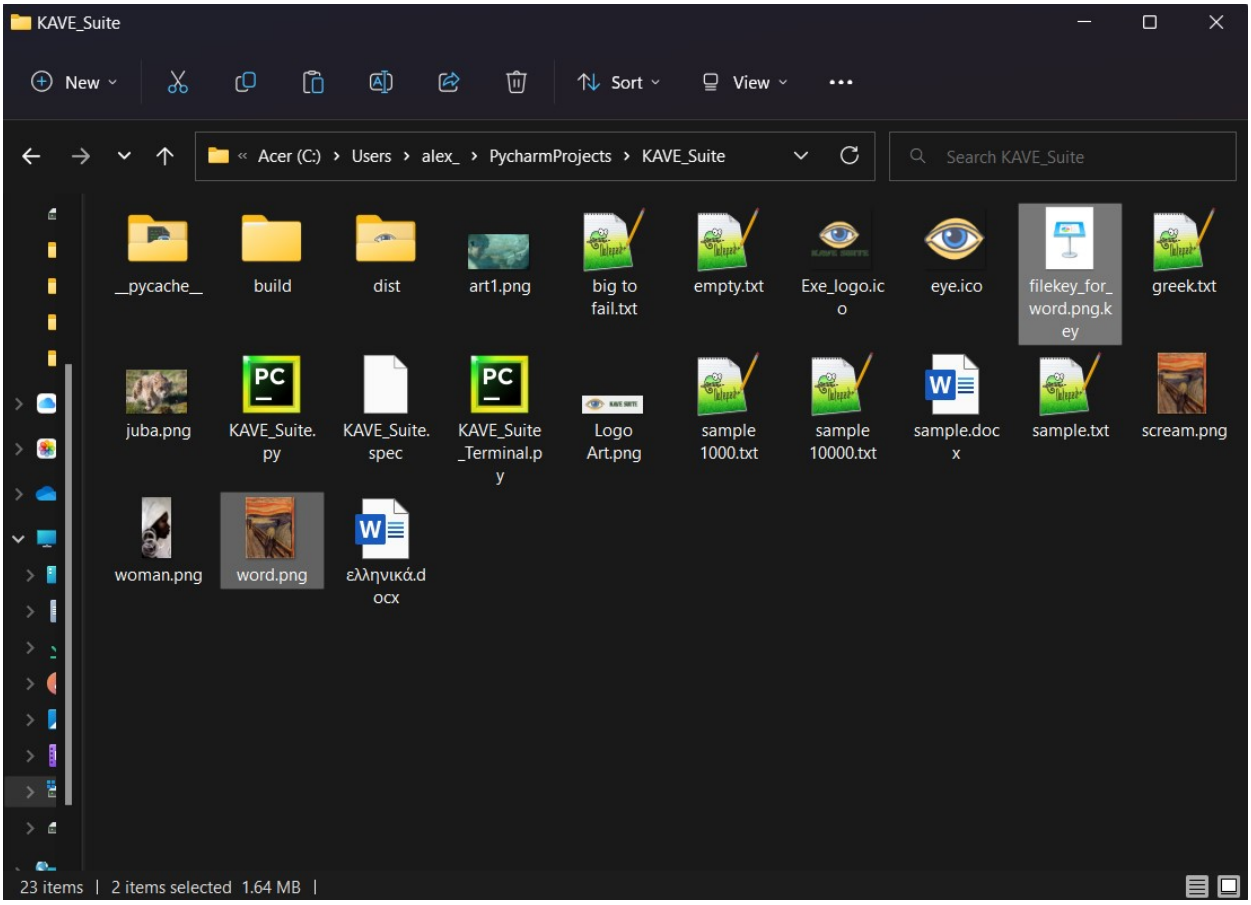

Figure 1.18

New files in our workspace

10] If the user does not enter a name for the new image, the application displays an error message and expects correct data entry (Figure 1.19):

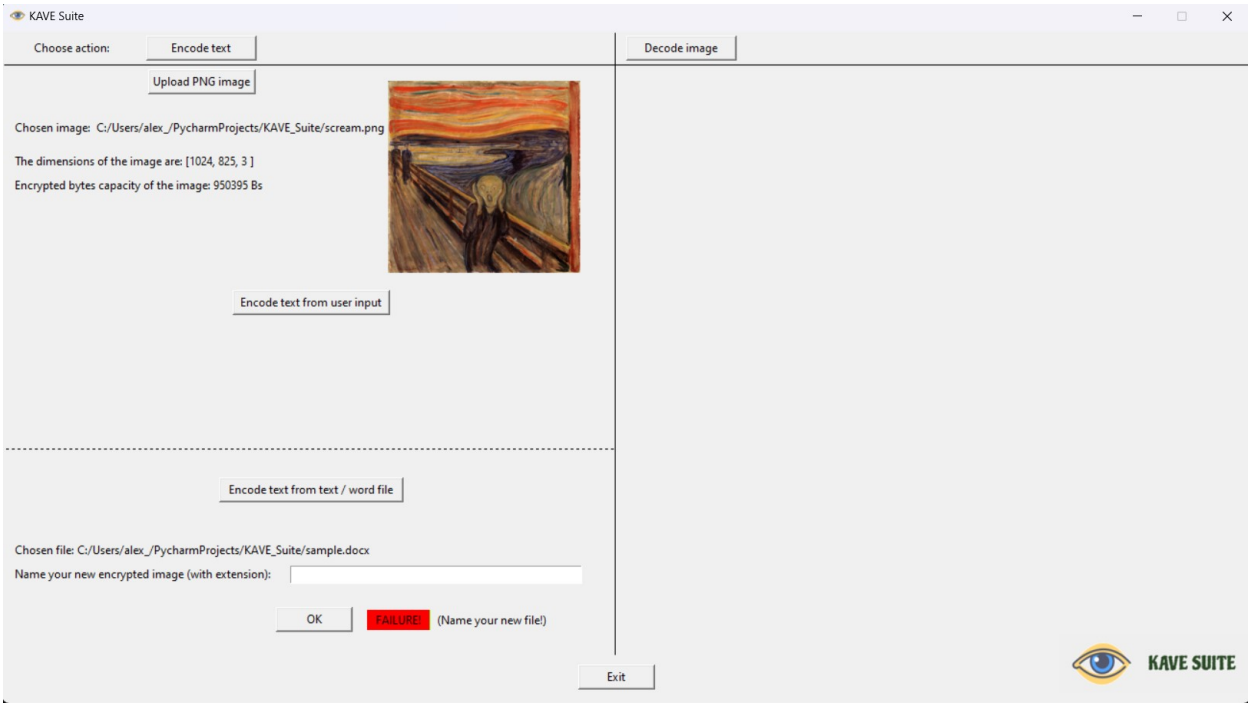

Figure 1.19

Error message when no name is entered for the new image

11] If we want to decrypt and subsequently decrypt data from an image that is in our files (and possibly received via e-mail or a file storage device, etc.) we choose '**Decode image**' and then '**Upload PNG image**' to upload the specific image (Figure 1.20):

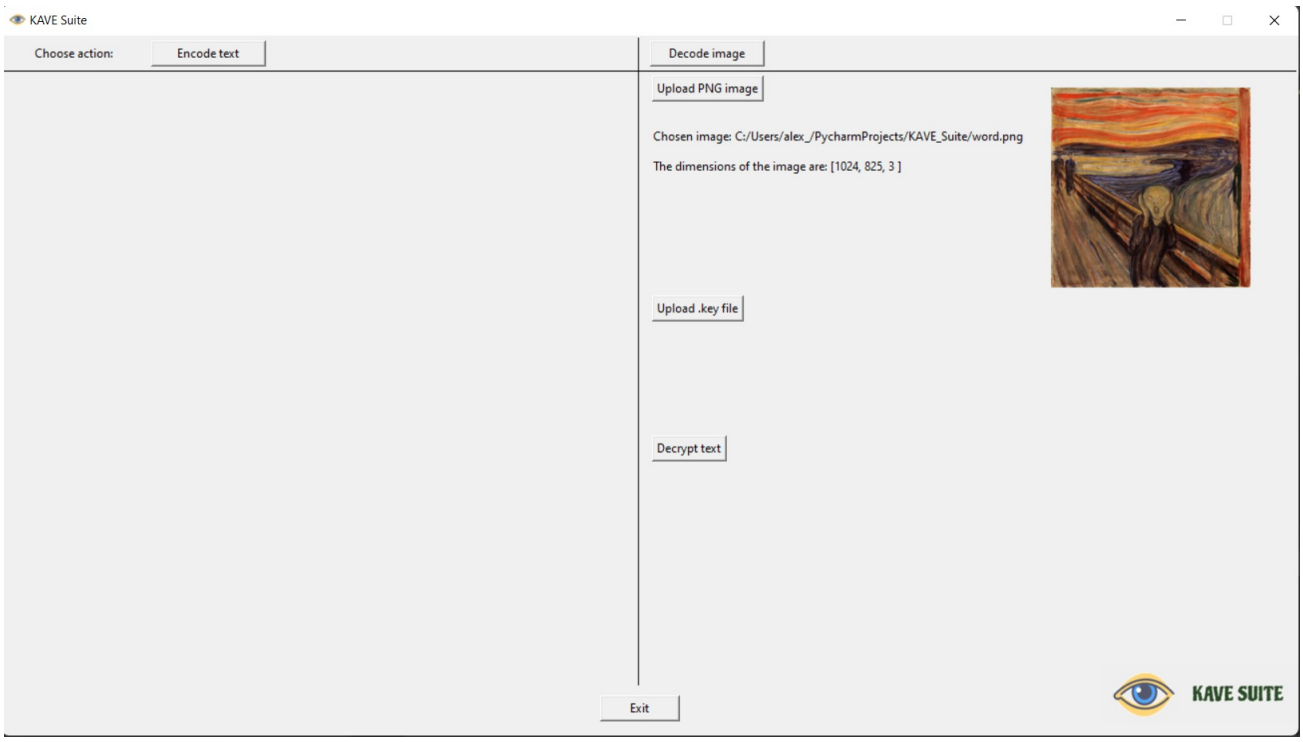

Figure 1.20

Format window after selecting image to decrypt and decrypt its data

12] Each key is unique and different from the others. Without the key corresponding to an image, which contains some encrypted text hidden, a third party can only extract the encrypted data, without being able to read the raw data. The key file can be exchanged between two communication parties either through a secure communication channel or via a storage medium etc. For the convenience of the recipient but also to better organize the generated files each time, the application adds to the file name of each key and the name of the image whose data is intended to be decrypted. For example, if we create an image with a name X.png, then the generated key will have a name filekey_for_X.png.key. In Figure 1.21 we select '**Upload .key file**' and select the correct file from the File Explorer. Then selecting '**Decrypt text**', a new file is created

'result_of_image_<image name>.txt' in our workspace (Figure 1.22). Selecting an incorrect key will cause the application to simply do nothing and wait until we select the correct key.

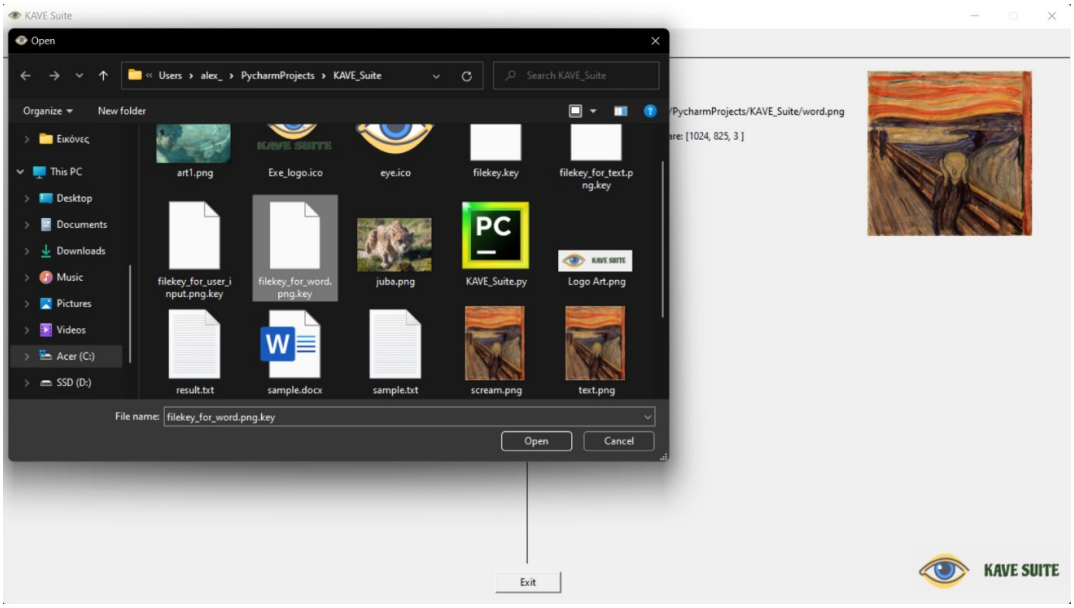

Figure 1.21

Selecting the correct .key file

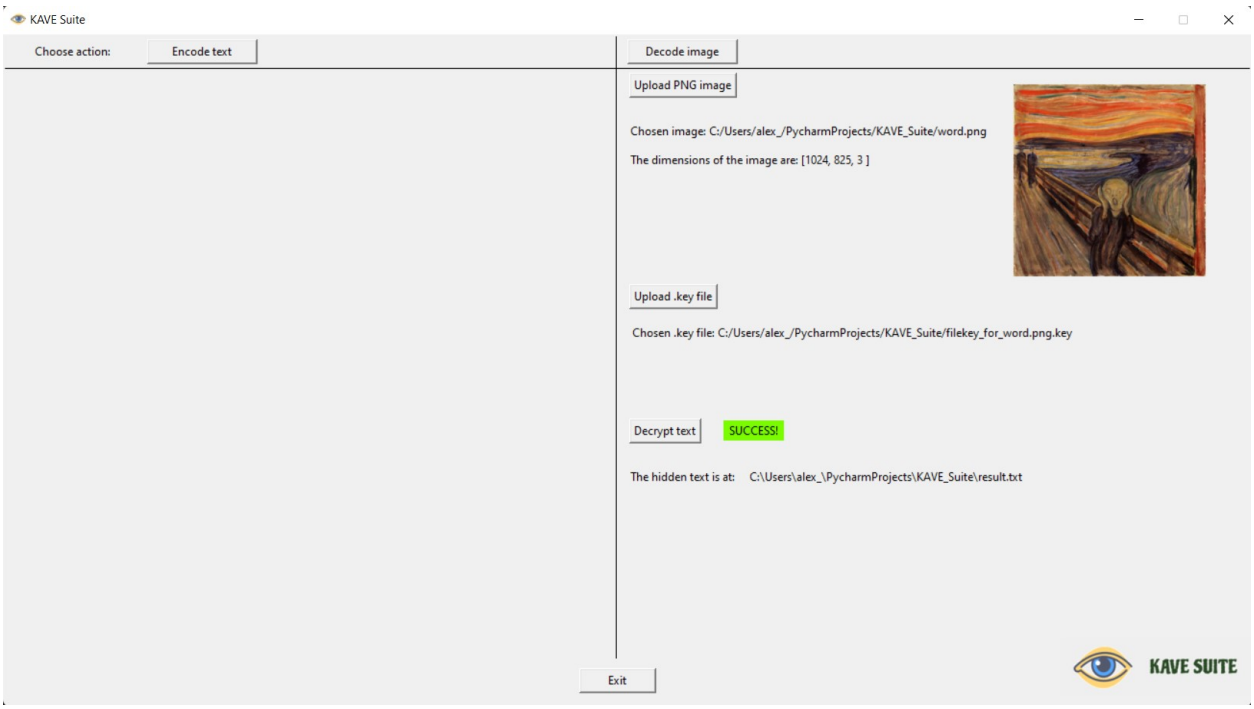

Figure 1.22

Message of successful decryption and decryption of data in file 'result_of_image_<image name>.txt'