Tampereen yliopisto

Viljami Irri

PROGRAMMING PROJECT
CryptoCanvas

May 2025

TABLE OF CONTENTS

ABBREVIATIONS

AES	Advanced Encryption Standard
CPU	Central Processing Unit
GCM	Galois/Counter Mode
GUI	Graphical User Interface
LSB	Least Significant Bit
MSE	Mean Square Error
OWASP	Open Worldwide Application Security Project
PSNR	Peak Signal-to-Noise Ratio
RAM	Random-Access Memory

1. INTRODUCTION

CryptoCanvas is a program that allows users to encrypt and decrypt images, and steganographically hide and reveal images and text within other images. The program has a local database where users can store image files. The users are authenticated with combinations of email and password. The users are also stored into the local database where their passwords are stored as hashes. The main points of secure programming in the application are hashing of the passwords with Argon2id, and encryption and decryption of images with AES-256-GCM utilizing key derivation with Argon2id. While steganography is not cryptography, it aids in concealing information through obscurity – only the sender and receiver are aware of the data's existence.

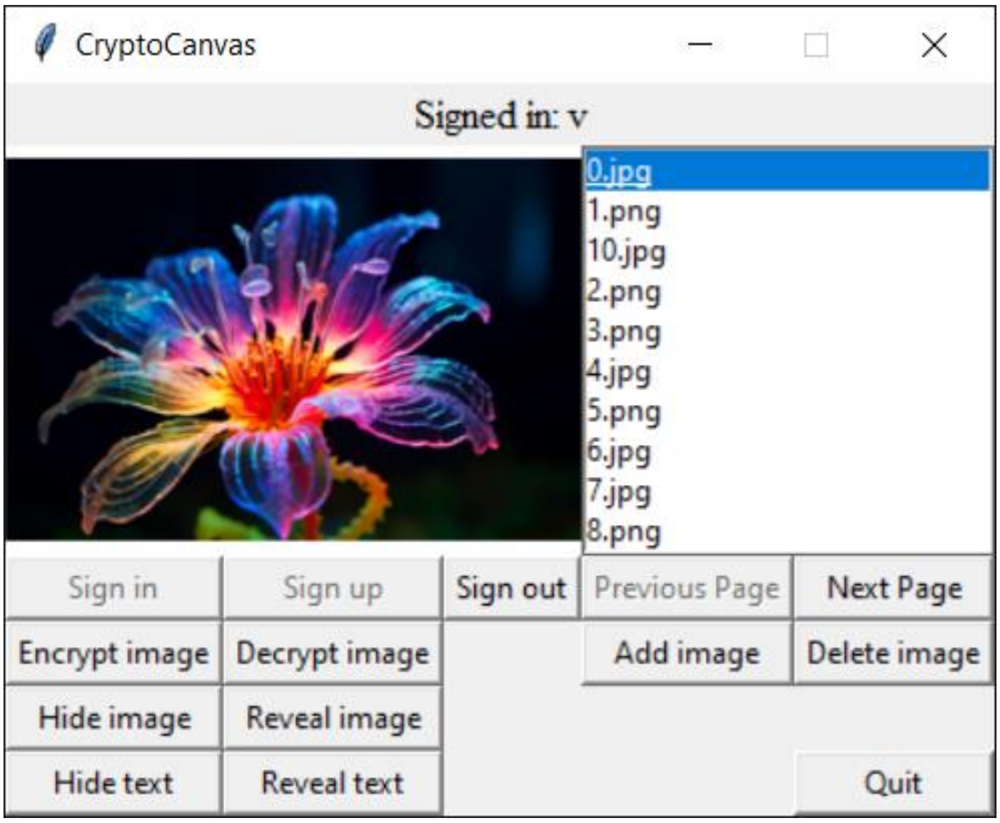

Figure 1. *GUI of the program.*

In figure 1, a user has signed in to the application and has added images to the database.

1.1 How to install and use

The program was tested on Python 3.7.9 on Windows 10. The required libraries are:

- cryptography (*pip install stegano*)
- PIL (*pip install pillow*)
- stegano (*pip install stegano*)
- argon2 (*pip install argon2-cffi*).

Other libraries used should be included in Python’s standard library and do not require explicit installations.

To run the program, navigate to the *src* directory in console and run *python main.py* (or *py main.py*).

Signing in grants access to the database feature. The user can store images in the database and use them for the operations in the program. The database can be navigated with arrow keys or mouse.

For the image operations, simply choose an image file from the database or the device for the given operation. The *Hide image* -operation requires two images to be chosen; the first image will hide the second image within itself. For text operations, after choosing an image to hide the text, the text can be supplied in a .txt file from the device, or by inputting text manually.

All files created by the program are saved on the user's device and images can be added to the database manually. The user is prompted to give the resulting file’s name in every occasion.

2. PROGRAM STRUCTURE

The class diagram of the program is shown in figure 2.

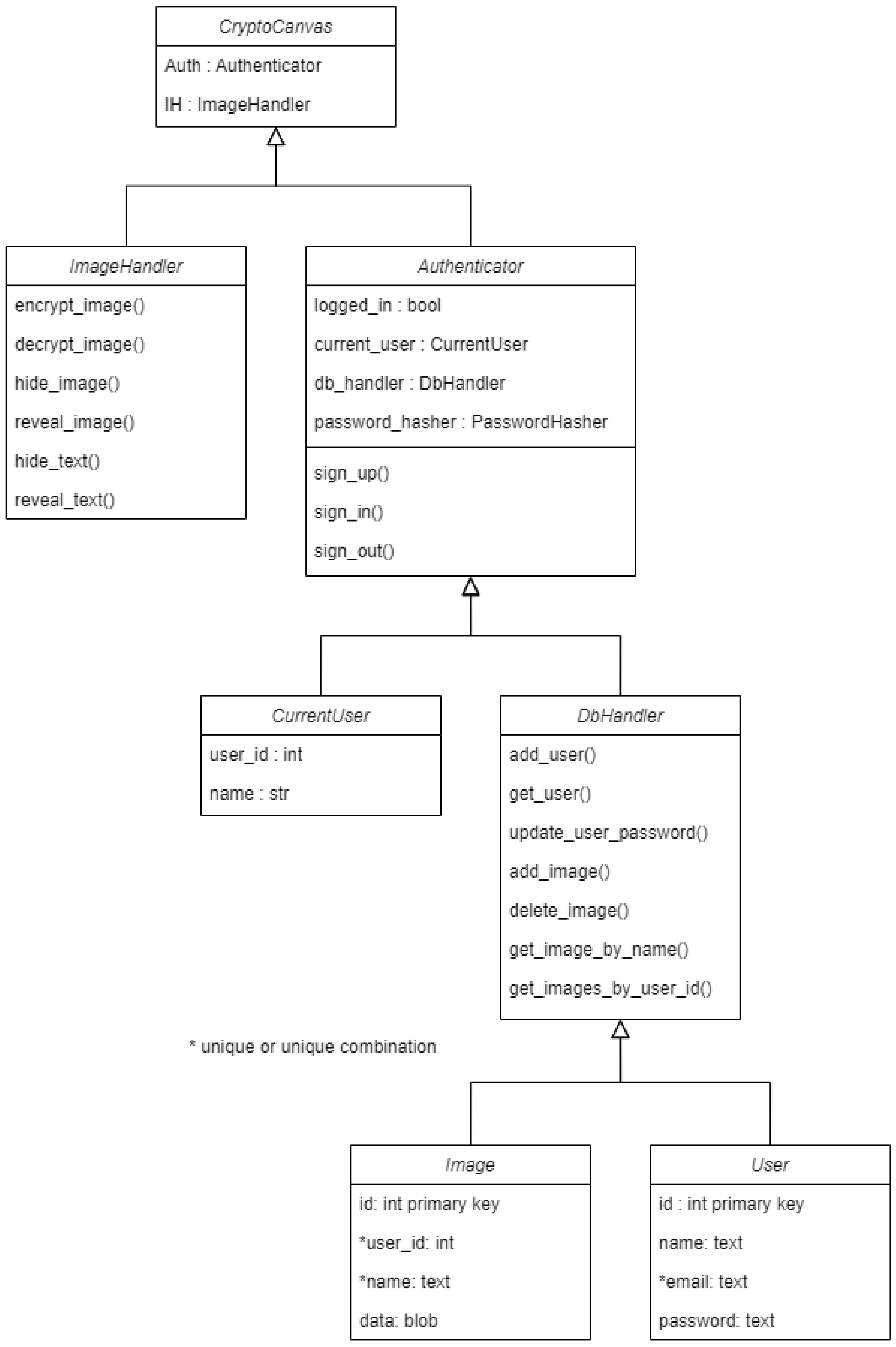

Figure 2. *Class diagram of the program.*

The class diagram does not list all methods and members of the classes, only the crucial ones for understanding the structure of the whole. The *CryptoCanvas* class is the main class of the program including GUI. It uses the *ImageHandler* and *Authenticator* classes in performing image handling and user authentication operations respectively. Because only signed up users of the program have access to the database feature, the *Authenticator* class has a *DbHandler* as a member which is responsible for all database-related operations. The database stores *Users* and *Images* in separate tables.

3. SECURE PROGRAMMING SOLUTIONS

3.1 General solutions

Exceptions in the program are handled using try-except blocks. In error cases a message box pops up informing the user of the error. Similarly, when the user cancels an operation, a message box informs them of this. When signing up to the program, the email is checked for uniqueness, and regular expressions are used for emails and passwords. If an image handling operation requires a file path instead of binary image data as an argument, and the user chooses an image from the database, a temporary file of the chosen image is created with the *tempfile* module's function *NamedTemporaryFile()*. This temporary file is deleted when the operation has been completed successfully, or when the operation is canceled.

3.2 Encryption

Encryption of the images is done with AES-256-GCM. AES is a symmetric encryption block cipher algorithm in this case with a key that is 256 bits long. GCM is a cipher mode that adds authentication to the encryption, providing integrity, authenticity, and confidentiality.

The encryption requires three things: a nonce or initialization vector, a file encryption key, and password. In this program the file encryption key is derived from the given password using salt with Argon2id. The nonce and salt are generated with the *secrets* module's function *token_bytes()*, which is a cryptographically secure pseudo-random number generator. The password for deriving the file encryption key is supplied by the user. The AES-256-GCM algorithm and the *secrets* module are recommended by OWASP [1].

For Python, the *cryptography* module has an implementation of AES-256-GCM which was used in the program. The nonce generated in the encryption process is prepended to the ciphertext. Similarly, the salt is appended to the ciphertext. When decrypting, the user supplies the password again, and the file encryption key is derived from it by retrieving the salt from the ciphertext. The nonce is also retrieved from the ciphertext and the encrypted file can then be decrypted successfully if the user provided the correct password. Therefore, remembering the password for each encrypted file is essential in ensuring that the file can be decrypted later.

There is a security vulnerability with this approach, which is when the same password is used more than once for encrypting a file. If an attacker is able to discover the correct password for a file, multiple files may be compromised. The program does not enforce any rules regarding the passwords for encryption such as length or special characters. This is partly because AES-256-GCM does not require a password for encryption, and I wanted to give this freedom to the user. However, I am not entirely sure what the Argon2 library does when the password to be hashed is an empty string, but the images seem to be encrypted regardless. I did not find anything in the documentation of the library about this. I have verified that the resulting key is a bytes-object. A security conscious user should know not to use the same password many times and be able to come up with strong passwords for sensitive files they might want to encrypt.

3.3 Hashing

User passwords are hashed using the Argon2id key derivation function which is a hybrid version of the Argon2d and Argon2i modes. The two modes together resist GPU cracking attacks, time-memory trade-off attacks, and side-channel attacks. All three modes have three parameters that control execution time, required memory, and degree of parallelism. The configurations affect CPU and RAM usage in different ways. The algorithm is recommended by OWASP with different configurations. [2]

For Python, the *argon2-cffi* library provides an implementation of all the Argon2 modes with possibility to configure the parameters. The implementation always uses a random salt for hashing by default. The same library was used for encryption key derivation.

3.4 LSB-steganography

Hiding and revealing images and text within or from images is achieved by utilizing LSB-steganography. It entails swapping out the least significant bits of the carrier image with those from the hidden message [3].

For Python, the *stegano* library provides an implementation for LSB-steganography. In this program, if the secret message is an image, its binary data is converted to a hex string and then hidden into the carrier image. Text can be hidden as it is. The sieve of Eratosthenes is used to select the pixels in which the data is hidden. It is an algorithm for finding prime numbers, and the library provides an implementation for this and several other algorithms for selecting the pixels to hide data in.

One of the weaknesses of LSB-steganography are the limitations imposed by the size of the carrier. Since each pixel of the carrier image can only hold one bit of secret data, the secret message must be small enough to embed into the carrier.

The performance of stego images can be evaluated using metrics such as PSNR, and MSE to analyze similarity of the carrier image and the resulting stego image. Randomization in pixel selection enhances the resistance against steganalysis attacks. [3]

3.5 Testing

The program was tested manually on Python v3.7.9. The program had lots of issues concerning usage of libraries, file selection logic, uncaught exceptions, and GUI-related problems.

The usage of AES-256-GCM was mostly straightforward but I had trouble finding a way to store the file encryption keys securely. In my initial implementation I made storing the key securely a responsibility of the user. However, convenience problems like this usually indirectly result into insecure practices, so I was not happy with it. After a lot of pondering, I realized that the encryption key could be derived from the password given for the encryption. Hashing of the password only required storing the salt which could be appended to the ciphertext. Now the user must remember just the password for each encrypted file, rather than also having to remember or store the file encryption key as well. This change improved the usability and hence the security, too.

Retrieving the hashed password from the output of the *argon2-cffi* library's *PasswordHasher* class's output was particularly difficult since the documentation of the library was really lacking for this. I discovered that the output was split into different sections with '$' characters, and the generated hash was encoded in unpadded base64, which I then had to decode. On the contrary, hashing of the user passwords was a much easier task than the encryption key derivation, because there was no need to extract only the hash from the output. In encryption it was necessary since AES-256-GCM requires a 256-bit key which did not match the output of the *PasswordHasher* class.

When testing the steganographic functions, I noticed that the *stegano* library required file paths for hiding and revealing operations. This did not cause problems until I decided to add the local database to the program. Retrieved images from the database did not have a file path, so this created the need to make temporary files for images selected from the database for steganographic operations.

The *stegano* library also required the message to be hidden to be a string. Using the library to hide and reveal text was easy because of this, but when hiding and revealing images I had to convert the secret image data to a hex string so I could do this with the library. Testing with different kinds of images I discovered that the library throws an exception if the message to be hidden is too long which I had not yet caught.

4. SUGGESTIONS FOR IMPROVEMENT

Because AES can encrypt text, text encryption could be added quite easily to the program, further enhancing security of hidden messages. Encrypted text could be hidden into an image, and then later revealed and decrypted. Text encryption could be implemented into the *hide_text()* function of *ImageHandler* if separate text encryption is not desired. Similarly the decryption of the text could be implemented into the *reveal_text()* function.

Adding images to the database is quite slow if many images are to be added. A path to a folder could be given and the images in the folder could be iterated through and added to the database to speed up the process.

Migrating the database to cloud would enable users to access their image files on multiple devices. For example, Google Firebase could be used for this.

Currently the CryptoCanvas class is quite large and there are a lot of functions created for selecting files. It could be beneficial to create a FileHandler class for these functions. This would be in-line with the single-responsibility principle in object-oriented programming, improving readability and maintainability of the codebase.

5. USE OF AI

ChatGPT-3 was utilized in this project several times. It proved particularly useful when troubleshooting errors and debugging issues in the code. There were occasions where ChatGPT could not pinpoint the problem and started hallucinating. The AI was also used for code generation with file operations, GUI design and database-related concerns.

File operations are typically very standardized so the AI could generate usable code easily. It had been a long time since I last used Tkinter for anything, so using the AI for help saved me time. When I was implementing the image preview window in the program, the AI could not help much since it was a specialized task, but it did provide me some useful template code. I had not used the *sqlite3* database library before, so I described what I wanted to achieve, and the AI was able to generate some template code I could then modify to get the functionality I desired. I ended up learning how the library is used through this project.

Overall, the AI is useful in general tasks related to code generation, but for specialized tasks it is often inaccurate and starts hallucinating. For troubleshooting and debugging purposes, the results vary greatly depending on the error. If the error is about logic and not syntax or use of a library, the AI is not helpful.

REFERENCES

[1] OWASP Cheat Sheet Series (2024). Cryptographic Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#algorithms (Referenced 14.4.2024)

[2] OWASP Cheat Sheet Series (2024). Password Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#argon2id (Referenced 14.4.2024)

[3] Alanzy, M., Alomrani, R., Alqarni, B. & Almutairi, S. (2023). Image Steganography Using LSB and Hybrid Encryption Algorithms. https://www.mdpi.com/2076-3417/13/21/11771 (Referenced 14.4.2024)