COMP6841 - Extended Cyber Security and Security Engineering

William Gaston - z5591798

Project Deliverables Report
avoDB

Summary

For my something-awesome project I focused on researching and developing an end-to-end encrypted Database-as-a-Service (DBaaS) program with integrated secure messaging functionality. Over the term, I designed and implemented a command line based tool that allows users to insert, store, and share data securely utilising both symmetric and asymmetric encryption practices.
The aim of my project was to provide users the ability to create and use databases with full control of their data as part of the 5 pillars of information assurance (*What are the five Pillars of Information Security*, 2024) - ensuring that it was only the client-side application which controls encryption and decryption. An end-to-end encrypted messaging system was also implemented to enable secure communication between users of the service.
This project demonstrates my learning and knowledge in applied cryptography, security by design, and database security, with a heavy emphasis on employing security driven best practice.

Project Overview

My project, avoDB, delivers a command line interface (CLI) based DBaaS program which combines end-to-end encrypted data storage and retrieval functionality with a secure messaging system created on top of a robust authentication mechanism. I aimed my project to accommodate users who want full control over the encryption and decryption of their data, offering client-side encryption which does not rely on the server for any functionality other than accessing ciphertext within the database.

The key features of my project include:

- End-to-end encrypted storage utilising symmetric and asymmetric algorithms
- Client-side encrypted messaging
- A robust key management system for generating and storing encryption keys
- A zero-knowledge backend server with minimal metadata exposure
- Secure authentication
- A modular repository structure posed for extension

To create my project I utilised the python programming language, and hosted it in a github repository. The backend was created with a local instance of a PostgreSQL server running in a docker container.

My project is primarily a proof of concept showing how encryption can be used to secure databases whilst also providing a learning opportunity for myself to utilise core principles learned throughout the course - cryptography, sql injections, security by design, layers of security - in a realistic development environment.

The above figure is a high-level view of what my project achieves from the user side.

Background

Software solutions are increasingly experiencing a shift away from local and hardware dependent implementations in favour of distributed, cloud architectures. Database as a Service (DBaaS) is a technology which has emerged as a result of this shifting environment and begun to be a critical component of modern web applications and other products - enabling automated management, scaling, operation and configuration of the underlying database. Unfortunately, this abstraction can lead to a reduction in security and data confidentiality as the database operates in a shared environment as part of the distributor's servers.

End-to-end encryption is a popular cryptographic framework which enforces a zero-trust ideology - the client side of the application is responsible for encryption and only ciphertext is distributed and stored. These properties make end-to-end encryption extremely beneficial for addressing the previously mentioned problems associated with creating and managing a DBaaS application. In particular it addresses the concern of multiple parties potentially having access to the underlying data of the database. If data is leaked or an attacker gains access to the database, they will not have access to the content as it is encrypted and only the client-side of the service holds the power to decrypt the information.

The DBaaS field directly correlates to the ideas of database security and cryptography, and intersects with the topics of key management and mitigation of SQL injection attacks, providing keen insight into the key principles underpinning secure data storage in cloud-hosted and distributed databases.

Problem Statement/Why

There are many commercial scale DBaaS applications provided by leading companies in the software space including Google, Amazon, MongoDB, IBM and more such as Firebase and Supabase, which offer competitive solutions which are both scalable and convenient for the consumer. However, a lot of these options lack end-to-end encryption and instead favour server-side encryption. This makes them vulnerable to data leaks and attacks arising from non-compliance of encryption standards, insider threats, or breaches of infrastructure. Some advanced offering (including MongoDB's Client-Side Field Level Encryption (CSFLE) (*MongoDB Docs*, no date) functionality) do allow for client-side and even field level encryption, but often come at the cost of efficiency and speed and are not as suitable for smaller scale applications and use cases.
There are also emerging companies such as Zama.ai which provides homomorphic encryption - a technique which allows you to perform computation directly on the encrypted data (Hindi, R., 2025) - and will prove revolutionary for encrypted database querying. However, this is computationally expensive and not generally suitable for a non-technical user.

My end-to-end encrypted database as a service, avoDB, will not be at a scale which can compete with any of these offerings but will instead be created as a small scale, personal implementation which will give me exposure to the fundamentals of cryptography, database security, and also the security by design lifecycle. It will be a simplistic showcase of end-to-end encryption and how it can be applied to databases, including through the real-world application of a messaging app. I will attempt to create a competent encryption scheme which balances both performance and security.

Research

Throughout development - particularly the design - of avoDB, I conducted research into one the field of cryptography theory and systems with the emphasis on gaining the knowledge needed to create a secure encryption scheme which prioritised security by design. The primary objective was to gain a sufficient level of understanding such that I could select, secure, and reason with the cryptographic tools I used, rather than implement complex mathematical algorithms and develop custom functions. Thus, my research surrounded understanding existing cryptographic functions and algorithms and how I can responsibly integrate these libraries into my project.

Encryption:

AES (Advanced Encryption Standard)

AES is a symmetric key block cipher operating on 128 bit plaintext blocks with up to a 256 bit key, formally adopted by the National Institute of Standards and Technology (NIST) for its

efficiency, speed, and security guarantees (*NIST, 2023*). It is a variation of the Rijandael Block Cipher created by Joan Daemen and Vincent Rijmenand employs a substitution-permutation architecture which offers “resistance against linear and differential analysis" due to the non-linear layer which comprises parallel applications of S-boxes“ and the linear mixing layer of the MoxColumns which “guarantees high diffusion over multiple rounds” (*Daemen, J. and Rijmen, V., 1999, p8*).
Each AES round consists of the following steps:

1. ByteSub: a non-linear transformation achieved through taking the multiplicative inverse in Galois Field (2^8) and performing an affine transformation as seen in the figure below:

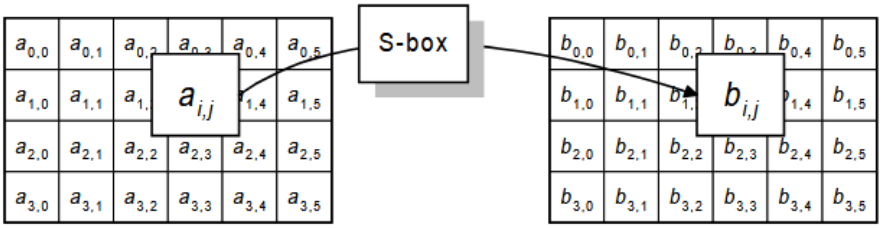
   1. First, taking the multiplicative inverse in GF(2^8), with the representation defined in Section 2.1. ‘00’ is mapped onto itself.
   2. Then, applying an affine (over GF(2)) transformation defined by:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

   The application of the described S-box to all bytes of the State is denoted by:

   `ByteSub(State)` .

   Figure 2 illustrates the effect of the ByteSub transformation on the State.



   (Daemen, J. and Rijmen, V., 1999, p11)

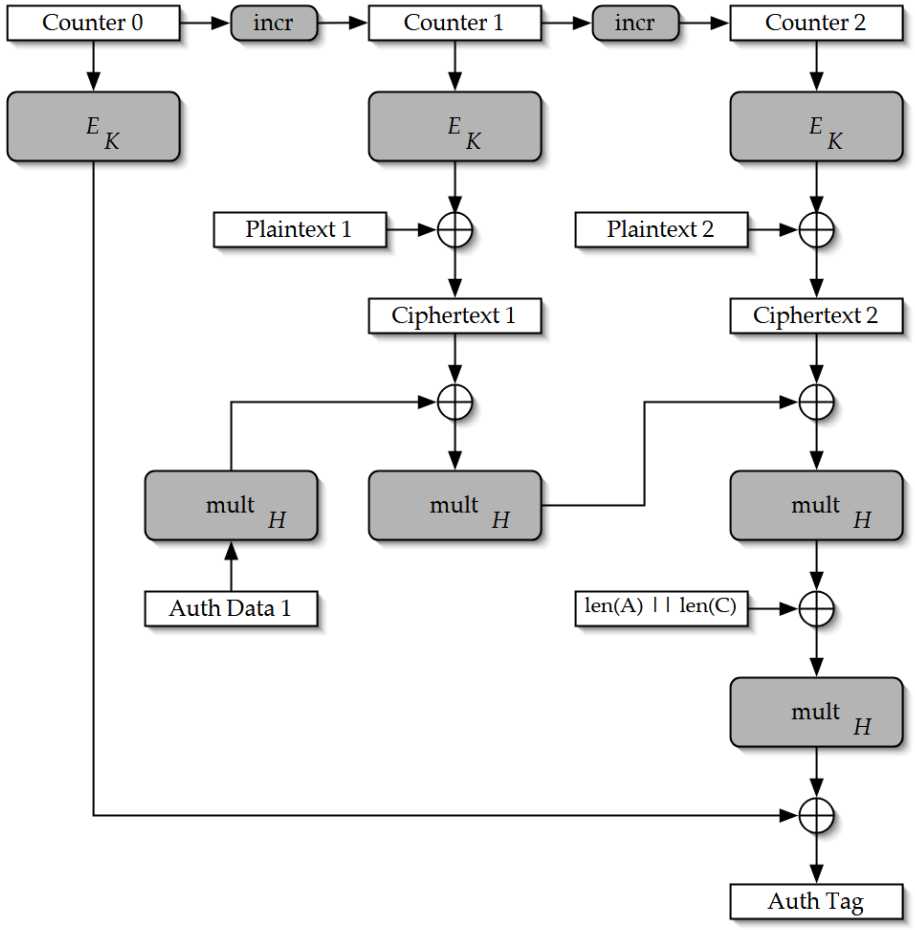
2. ShiftRow: “shifting the rows … over specified offsets”
3. MixColumn
4. Round Key Addition: A round key derived from the scheduled cipher key is XORed with the current ciphertext state.

(*Daemen, J. and Rijmen, V., 1999, p11-13*)

My use of the AES algorithm will be for the encryption of data stored within the database, allowing for efficiency whilst still maintaining a high degree of security. I will implement this algorithm through the python cryptography library (*Symmetric Encryption, 2013*) as it is of the highest standard and continually maintained to ensure robustness and protection against changing vulnerabilities.

GCM (Galois / Counter Mode)

GCM is a mode of operation for block ciphers which combines the classical counter mode of operation - which converts them to a stream with an encrypted counter which is XORed with the plaintext - with a GHASH function which performs hashing over the Galois Field 2^128 (*McGrew, D. and Viega, J., 2004, p5)* with a subkey to create an authentication tag which can be used to check data integrity.



(*McGrew, D. and Viega, J., 2004, p6*)
GCM, when combined with AES, provides both encryption and authentication within a single operation without compromising on performance. The only drawback is that the same iv should not be used with the same key as it may be vulnerable to common stream cipher attacks, but the inherent parallelism of binary Galois field multiplication makes the mode especially fast (*McGrew, D. and Viega, J., 2004, p1*).

Argon2

Throughout my research into password hashing, I found that prior to the 2010s there was no standard for password hashing functions, and that 2013 through 2015 a Password Hashing Competition (PHC) was run to establish a standard algorithm much like the NIST AES competition (Biryukov, A., Dinu, D. and Khovratovich , D. 2015). The winner of this competition was argon2, a memory hard password hashing function which has significant entropy and which could tune the amount of memory and data needed to calculate (Aumasson, Jean-P. (no date)). I chose to use the pypi argon2 library as my implementation of this algorithm (Moroz, I. (no date)).

RSA (Rivest-Shamir-Adleman)

RSA is an asymmetric public-key encryption scheme which utilises a private and public key for encryption and decryption. The security of this scheme is guaranteed by the difficulty of factoring the product of two large prime numbers with modern computational capabilities and its hardness is defined by the problem of integer factorisation. However, as RSA is deterministic, padding is required to ensure the message is obscured. Optimal Asymmetric Encryption Padding (OAEP) is one such padding scheme and uses a mask generating scheme to introduce randomness which can be recovered through the private key of the RSA scheme (*Wikipedia. (2020)*).
In avoDB, I will utilise RSA encryption to secure my symmetric keys for storage within the database. This will provide a mechanism to ensure only the correct users can access their data whilst also allowing for data transfer and sharing between users. I will use the python cryptography library to implement my encryption scheme.

Implementation and Rationale

High-level Overview / Repository Security

Every design decision and implementation choice throughout avoDB was designed with security as the forefront concern - with an architecture centered around a client-first approach with an otherwise zero-trust policy in regards to data storage and retrieval. This ensures that the server and underlying database is never given full trust of the data, keys, or encryption/decryption control scheme.
The fundamental ideas of my project center around the CIA triad of confidentiality, Integrity, Assurance as well as the Five Pillars of Information Assurance which add on Authentication and Non-Repudiation. As part of this I utilised the idea of client-side encryption which encrypts and decrypts the data on the application layer of the program prior and following transmission respectively. In this way, the backend and underlying Postgresql database are treated as an untrusted and potentially insecure storage layer.

The structure and content of my project repository follow the security by design principles, ensuring that the lifecycle of my project - not only development and design, but also future improvements or maintenance - is secure. This includes a strict set of .gitignore rules which hide sensitive data including keys, environment variables, and setup scripts - ensuring that these are not leaked.
I have also created a requirements.txt file for secure, vetted, and reliable dependencies to mitigate supply chain attacks and other security issues arising from untrusted dependencies.

Schema

As fundamentally a database-based project, I have designed a robust shema which abstracts the functionality of the service into workable chunks and which has a reasonable query performance time and level of redundancy. Each schema has minimal metadata exposure and only stores required fields, these tables include:

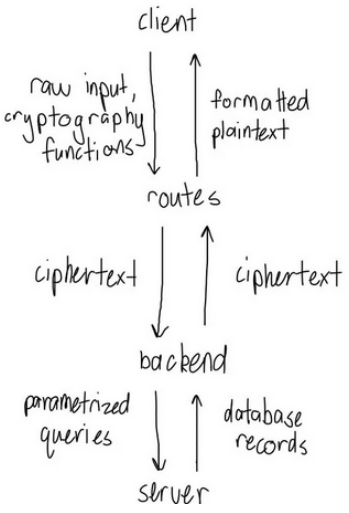
- UsersMeta: holds the user data including user id, username, hashed password, encrypted private key
- Databases: this is the highest level of the database service and holds user created databases, including, the database if, encrypted master key, encrypted name, and owner id
- Tables: this middle layer of the database service holds the the tables associated with a particular database instance and includes fields such as table id, database id, encrypted table name and schema
- Rows: this lowest layer of the database service holds the actual encrypted data a user inputs into a table, including fields such as row id, table id, iv, encrypted data.
- Messages: This table is solely used for the integrated messaging component holding values such as the encrypted message, sender and receiver ids, encrypted sender and receiver keys, and timestamp. Although it could also be created using the database service this would lead to a loss of performance and bloat which I deemed unreasonable.

```sql
-- dump of database schema

create table if not exists UsersMeta (
  user_id              UUID primary key,
  username             text not null unique,
  kek_salt             text not null,
  pk_iv                text not null,
  hashed_password      text not null,
  encrypted_private_key text not null,
  public_key           text not null,
  created_at           timestamp not null default current_timestamp
);

create table if not exists Databases (
  db_id                UUID primary key,
  owner_id             UUID not null references UsersMeta(user_id) on delete cascade,
  iv                   text not null,
  encrypted_db_name    text not null,
  encrypted_master_key text not null
);

create table if not exists Tables (
  table_id             UUID primary key,
  db_id                UUID not null references Databases(db_id) on delete cascade,
  encrypted_table_name text not null,
  encrypted_schema     text not null
);

create table if not exists Rows (
  row_id               UUID not null primary key,
  table_id             UUID not null references Tables(table_id) on delete cascade,
  encrypted_data       text not null,
  iv                   text not null,
  created_at           timestamp not null default current_timestamp
);

create table if not exists Messages (
  message_id             UUID not null primary key,
  encrypted_message      text not null,
  sender_id              UUID not null references UsersMeta(user_id),
  receiver_id            UUID not null references UsersMeta(user_id),
  sent_at                timestamp not null default current_timestamp,
  encrypted_sender_key   text not null,
  encrypted_receiver_key text not null,
  iv                     text not null
)
```

Secure Structure and Modular Isolation

The internal structure of avoDB creates modularisation and isolation of the different components needed for operation - separating the core responsibilities to help prevent privacy concerns and reduce control movement within the system.

The overall structure interacts like so:



These components include:

Client

The client holds the key modules used for user interfacing, cryptography, and session management. This directory is the ‘trust boundary’ (Andersen, L.R. 2025) of the system and holds the core implementation of the system’s security practices - holding the logic of the encryption system. Outside of this, the larger program structure has no knowledge of how the encryption and hashing takes place and simply interfaces with the functions defined within.

- /cli
  Provides the interface between the user and the system, implemented using click which handles parsing and validation of user input. The cli holds no logic and is simply an interface delegation mechanism.

- /cryptography
  This module is the core of systeem and handles the logic for everything related to ciphertext, secure storage, and authentication. In particular it handles key generation, and password hashing, holds key derivation functions and key serialisation methods, and is responsible for the implementation of encryption and decryption using both symmetric and asymmetric cryptographic functions.

- /storage
  Session management is the primary concern for the client-side storage module. It holds the functions which save and retrieve information which persists during a user’s use of the program - it is the interface which interacts with the operating system’s keyring.

Routes/Core

The routes/core module holds the primary logic of my application, in particular it provides the mechanism for which my client-side functions interact with the backend and server. It handles the enforcement of the client-side trust boundary - utilising the client-side routes to encrypt all plaintext prior to interaction with the server. Furthermore, the routes handle verification of database, table, and row ownership and authentication of messages and data.

- /auth.py
  This list provides the functionality for the operations defined in the auth segment of the program - handling authentication through providing login, register, logout, and user list functions. It primarily interacts with client side key generation and session management (although is not exposed to the algorithms which produce the results it handles).

    - Login
        1. query db to check if username exists
        2. return the hashed password from the db
        3. verify hashed password

4. save keys for session

- Register
    1. query db to check if username exists
    2. create uuid
    3. create key pairs
    4. serialize key pairs
    5. create kdf salt
    6. create pk iv
    7. hash password
    8. encrypt private key
    9. save keys for session

- Logout
    1. clear credentials

- User list
    1. find and format list of user id and names

- /tables.py
  The tables route file provides the core mapping and interface which connects the user to the backend of the actual database. It holds routes for creating, viewing and deleting databases, tables, and records and utilises functions from the cryptography module to encrypt and decrypt this data. Sessions held on the keyring are also used for operation validation to ensure users only access data they have access to.
  Some example workflows include:

    - Database creation
        1. generate master key
        2. generate iv
        3. generate db id
        4. encrypt db name
        5. encrypt master key
        6. create db via backend

    - Insert row
        1. check tb belongs to user
        2. validate data to schema
        3. generate iv
        4. generate rowId
        5. encrypt data
        6. check follows schema
        7. insert

    - Select rows
        1. check tb belongs to user
        2. get array of rows
        3. loop and decrypt

4. get schema
5. display

- /messages.py
  This module holds the logic of the integrated messaging function which I built into the database. It synthesises the sending and viewing of messages and also facilitates the ability to initiate a conversation which essentially puts the user id and public key of who you are communicating with into the keyring for quick access. Some example workflows are below:

  - Initiate Conversation
    1. get user's public key
    2. add user id and public key to keyring

  - Send Message
    1. get both users' ids from keyring
    2. get both users' public keys from keyring
    3. generate message id, iv, and encryption/decryption key
    4. encrypt message with key
    5. encrypt key with sender and recipient public keys
    6. send message / add to db

  - View Messages
    1. get sender/receiver ids from keyring
    2. get other party's name
    3. get messages sent by current user
    4. decrypt and add messages to list
    5. get messages received by current user
    6. decrypt and add messages to list
    7. combine and sort messages by timestamp
    8. format and print

Backend

The backend of avoDB is split into two main sections: the database server, the python query execution. My database is a custom PostgreSQL server running in a docker container - with inbuilt persistence via a data volume. To interact with this server, my backend python files utilise psycopg2 parameterised queries with a pool of connection objects which allow for concurrent operations.

- /dbinit.py
  This module is concerned with setting up the psycopg2 functions and objects which are required to access the server. They are only used in other backend functions and so the routes and client do not have access - this is a further separation of concerns and control albeit in the opposite vein to that implemented in the client-side encryption model. Some of these functions include initialising the database

connection pool, creating and releasing connection objects, and creating and releasing cursor objects.

- /auth.py
  The auth backend module holds the queries responsible for interacting with the user meta data table of the underlying database server. It adds new users to the service and is used to verify logins by fetching hashed passwords. The data for checking database, table, and row ownership are also present in this file.

- /db.py
- /table.py
- /rows.py
  All three of the above modules are respectively responsible for inserting into, selecting from, and modifying the core tables responsible for providing and managing the ciphertext of my database as a service application. They operate on the databases, tables, and rows tables and also control the mappings between each.

- /messages.py
  The message module holds the database queries needed for the secure end-to-end encryption component of avoDB. These queries interact with the messages table, but could easily be modified to work on a messages table made as part of the service. This includes functions to view a list of users who are in a conversation with the current user, send messages, view messages, and also helper functions for getting the receiver's public key used for encryption when sending messages.

Key Management and Cryptographic System

Overview

avoDB utilises a hybrid encryption system which combines the properties of both symmetric and asymmetric encryption techniques to strike a balance between security performance and efficiency.

- Symmetric Key Encryption
    - I use symmetric key encryption - utilising the AES-GCM algorithm to provide confidentiality and authentication - to encrypt the actual data within my database, this includes: database, table, row records, messages, user data.
- Asymmetric Key Encryption
    - I utilise asymmetric key encryption - utilising the RSA scheme combined with OAEP which uses SHA-256 internally - to encrypt, store, and share symmetric keys amongst other users and the database backend. The private key is created with the user's password and symmetrically encrypted using a key derivation scheme for storage in the database.
- Hashing
    - Hashing is used within avoDB to store and verify the user's password, to allow them access to the database service and also for encryption/decryption purposes with the private key. This uses the argon2 hashing algorithm.
- OS keyring
    - For session management and quick access to the user's credentials to reduce database access for repetitive actions like getting the user's keys, I utilised the keyring python module which acts as a wrapper for the operating system's native keyring.
- Key Generation
    - For the majority of key generation including that for master key, initialization vectors, salts, and uuids, I utilised the os.random function which leverages the operating system's inherent entropy to generate cryptographically secure sequences of bytes. To generate private and public keys I utilised the rsa algorithms.

How does it work?

1. Password Hashing
    - The user's input password is compared to the record correlated to their username (which has enforced uniqueness validated by the postgresql server). This record holds a version of their password which was hashed using the argon2 algorithm. This algorithm cannot be deterministically reversed and is thus safe to store within the backend. Using the argon2 verification tool a plain text password can be compared to this hashed version and can be verified as being the same or different.
2. Data Encryption
    - Sensitive data (such as schema definitions, table names) and records are stored in the database as ciphertext which is encrypted through the use of the database's master key and symmetric encryption - each row also has a unique iv.

    - The use of AES-CBC ensures the data is stored with confidentiality and cannot be interpreted without decryption..
3. Master Key Encryption
    - The symmetric master keys which are used for data encryption are stored in the database after being encrypted with the user's public key. This ensures that only users with the correct private key can decrypt and thus use the master key to decrypt the database's actual data.
    - In the messaging functionality, both the sender and receiver get the master key of the message encrypted with their respective keys, thus demonstrating the ability to share data between different users - and could thus be extended to encompass whole databases in the future.
4. Asymmetric Key Encryption
    - Each user has a unique public and private key which gives them access to the rest of the data, thus the private key cannot be exposed. To securely store the user's private key, I utilise the secrecy of their password - through using a key-encrypted-key with their password, their private key can be dynamically decrypted with symmetric encryption techniques.

Rationale / Justification

Advantages:

- Performance: Using symmetric encryption on the actual data is significantly faster than asymmetric encryption providing more efficient access and utilisation.
- Key Exchange: Asymmetric cryptography is exceedingly secure and computationally intensive if not impossible to reliably crack, thus utilising RSA keys makes it possible to share data without exposing secrets.
- End-To-End Trust: The user and thus client do not need to place any trust into the server or backend of the system. If their password is kept safe, then even if the entire database server is exposed or compromised, their data is still secure
- Access Control: The sharing of data and especially the messaging system is simply and robust and can rely on the sharing of public keys to be used to encrypt master keys - providing a more fine-grained approach
- Layers of Security: The multiple different keys and encryption techniques greatly reduce attack vectors - as long as the user's password is not lost, no data can be decrypted.
- Zero Trust: The underlying database does not have access to any un-encrypted keys or plaintext thus database breaches will not result in loss of data.

Disadvantages and Attack Vectors:

- Complex Key Management: Managing multiple key and cryptography types increases the complexity of the workflow and could make it harder to reason with the code and find errors - this is managed through clear abstraction and modularisation
- Password/Key Compromise: If the user's password or un-encrypted private key is revealed, an attacker has complete access to that user's data - however, each user has access to only their data by default, so one breach is isolated to one user not the entire system.

- SQL Injections: Taking user input at introducing it to an sql query can potentially lead to a bypass into the control flow of an application, causing leakage of the database, or the tables to be dropped. This is mitigated through the use of parameterised queries.

Security Feature and Properties

As mentioned in the executive summary and overview, the goal of this project is to explore how end-to-end encryption and the various associated techniques are used to mitigate security risks and abide by the 5 Pillars of Information Assurance:

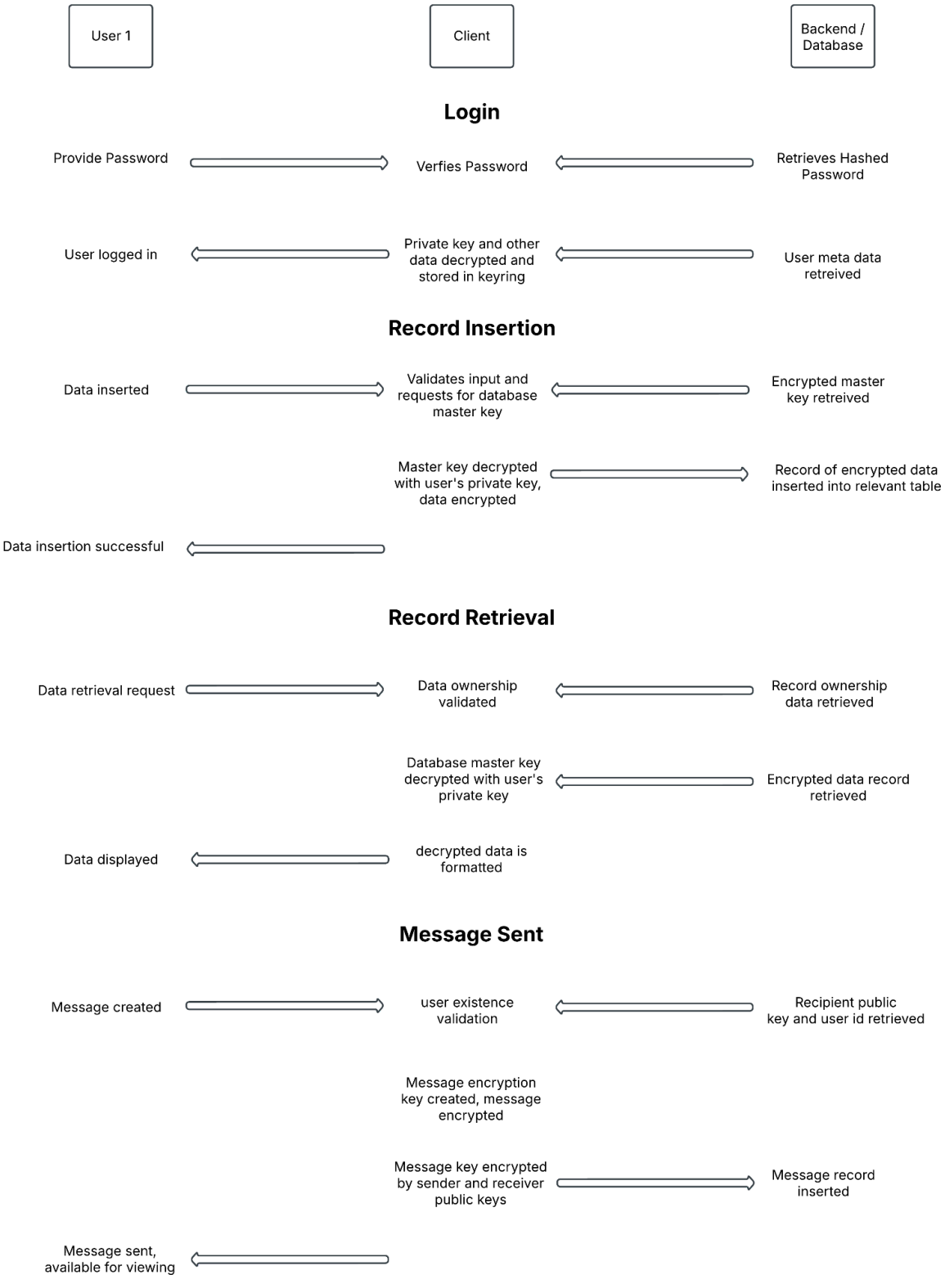
1. **Confidentiality**
   All database records are encrypted client side using AES-GCM, ensuring strong security and confidentiality ensuring the data is secured in transit and at rest, disallowing unauthorized access to the data.
2. **Integrity**
   The integrity of the data used in my database is ensured through the Galois / Counter Mode (GCM) which creates and verifies cryptographic/authentication tags. Tampering of the data - via changing bits - will cause the tag check to fail.
3. **Authentication**
   Encryption and decryption is tied to the possession of the correct keys which can only be decrypted using the user's password, thus only authorized users can access the data. GCM further utilises authentication tags which verify the identity of the sender and receiver.
4. **Availability**
   Though not a focus, the system supports a local, dockerized postgres server instance containerized to support data persistence, portability, and overall resilience.
5. **Non-repudiation**
   To establish non-repudiation, I employed digital signatures which link the user's private key to their messages, verifiably linking actions to the users which perform them.

Other security concepts explored include:

- Zero-trust: The server never has access to plaintext of unencrypted encryption keys, keeping data private even after a breach.
- Minimal attack vectors/points of failure: Since all data is encrypted and the encryption process follows Kerckhoff's Principle of algorithm exposure, the only attack vector is through client-side exposure of the plaintext password.
- Layers of security: There are three layers of privacy/confidentiality protection: password, private key, master key.
- Security by design: I utilised security by design throughout my project lifecycle to ensure best practices were met, this included extensive drafting and planning of the key implementation scheme and designing and coding of the software solution.

Encryption Flow Example

[plaintext] ===> sign ===> [plaintext , signature] ===> encrypt ===> [ciphertext]


User 1
Client
Backend / Database
Login
Provide Password
Verfies Password
Retrieves Hashed Password
User logged in
Private key and other data decrypted and stored in keyring
User meta data retreived
Record Insertion
Data inserted
Validates input and requests for database master key
Encrypted master key retreived
Master key decrypted with user's private key, data encrypted
Record of encrypted data inserted into relevant table
Data insertion successful
Record Retrieval
Data retrieval request
Data ownership validated
Record ownership data retrieved
Database master key decrypted with user's private key
Encrypted data record retrieved
Data displayed
decrypted data is formatted
Message Sent
Message created
user existence validation
Recipient public key and user id retrieved
Message encryption key created, message encrypted
Message key encrypted by sender and receiver public keys
Message record inserted
Message sent, available for viewing

Issues Encountered

Key Management Complexity

Issue: The complexity of my layered, hybrid key and encryption approach made it difficult to store, retrieve and utilise said keys without database knowledge.
Solution: I utilised local OS keyring modules for session management to reduce database overhead, and utilised encryption with the password as the sole point of failure.

Testing Difficulty

Issue: It is exceedingly difficult to robustly test cryptography using deterministic output, especially given the tight timeframe of the project.
Solution: I conducted extensive manual testing throughout development to ensure the different components function as expected.

Performance Reductions

Issue: Initially, it was difficult to use solely RSA asymmetric encryption as it would greatly reduce performance at higher scale.
Solution: I introduced a hybrid encryption scheme using symmetric encryption through AES as my primary, speed-focused confidentiality mechanism and asymmetric encryption through RSA for my robust key sharing and encryption system.

Complex Encryption Library

Issue: Navigating the python provided cryptography libraries is exceedingly confusing and difficult without prior knowledge and exposure.
Solution: I read a multitude of documentation and completed trial and error until I found a working solution.

Lack of Integrity/Authentication

Issue: My initial encryption scheme utilised AES-CBM which ensures confidentiality, but does not inherently support integrity or authentication. This makes my database and the data stored inside vulnerable to tampering with no method of ensuring the data is the same as that which was put into the database.
Solution: I conducted research and found that I had a couple of options.These included:

- message authentication code (MAC) which is a hash of the original message using a shared key and added to the stored data
- hashMAC (HMAC) where the shared key is used in the hashing process and not appended to the message. This is more desirable than a simple MAC as it is secure against reversal.
- AES Galois/Counter Mode (GCM) mode of operation builds on counter mode and implements a MAC using a GHASH function which uses Galois fields rather than primes. It is slightly weaker than a HMAC but significantly easier to implement and has the benefit of high performance with inexpensive resources.

I utilised the cryptography library primitives to implement the AES-GCM mode and algorithm.

Lack of Non-repudiation in Messaging

Issue: Using the hybrid encryption scheme of RSA and AES-GCM does not inherently provide signatures and guarantees of non-repudiation of messages within my messaging functionality.
Solution: I modified my encryption scheme to encrypt a JSON object containing the plaintext data and a signature created using the user's private key which could be verified with their public key during decryption. If tampering is detected, an exception will be raised. I implemented this using the cryptography primitives library and the sign and verify functions of RSA objects.

Professional and Ethical Issues

Data Ownership

As a database-centric project, avoDB is centered around the ethical consideration of data privacy, primarily the idea that users should have full access and control over their plaintext data and keys. There is no backdoor or method for anyone other than the user to access their unencrypted keys and data without the explicit use of their password. This aligns with the idea of data privacy through design.

Zero-Knowledge

A zero-knowledge database server means that the backend does not have access to the underlying information stored within it, providing users with increased privacy and confidentiality. However, this also provides the challenge of their being no point of recovery or trace if a user loses their password or accidentally deletes sensitive information held in their databases. This is very difficult to prevent from a non-user perspective other than through preventative measures such as thorough documentation within the client interface.

Professional Behaviour

Through this project, I was subject to a higher level of professional rigor compared to previous experiences due to the confidential nature of encryption, some of the key behaviour which I learned and demonstrated is:

- Maintained clear git commit history and structure, with ample documentation useful for extension
- Utilised .env files and a containerized server via a Docker implementation to manage sensitive data.
- Abided by Kerckhoff's law of transparency rather than 'security through obscurity'.

Analysis of Strengths and Weaknesses

Strengths

- Security Centered Design: All database features are centered around cryptographic principles such as client-side end-to-end encryption, and attempt to adhere to the CIA triad and 5 pillars of information assurance.
- Modularity: The codebase is modular and thus has the propensity for extension or maintenance.
- Self-contained: The project minimises attack vectors by ensuring the password is the primary point of failure.

These strengths are evidenced by successful encryption/decryption without server involvement, the ability to create and view messages without plaintext exposure in the server itself, manual testing of database, table, and record functions.

Weaknesses

- User experience: the interface to the system is command line based which provides a worsened user experience, lesser accessibility, and reduced convenience.
- Passwords are a major point of failure: There is no recovery for a person's account or data if their password is lost or compromised due to the zero-trust system.
- Limited scale: The system is currently only a proof of concept and does not offer full functionality of services outside of a single local instance.
- Lack of robust testing: Due to time constraints, automated unit testing has been overlooked → as a proof of concept this is not the focus.
- Edge Cases: My error checking of input is not completely sound and there are holes due to time constraints.

Reflections and Personal Development

This project was challenging primarily due to its open-ended nature. It pushed me beyond the theory learnt in class and lectures directly into a practical implementation of security concepts and systems, which is something I have not done prior. As the topic was chosen by myself I had little support outside of online resources and was left to my own devices to understand how to go about my design and development.

Key Challenges

- Conceptually, it was very challenging to create and design a suitable encryption scheme. The current complexity of my hybrid symmetric and asymmetric approach took significant research and self-guided study with little standardization.
- Understanding and applying cryptography theory was difficult, utilising dense libraries and trying to navigate the complexity of things like IVs, padding schemes, algorithms was very overwhelming.
- Debugging the encryption flow was fraught with challenges as it could not be easily reasoned with and the errors were often quite obscure.
- Approaching the task at a reasonable scale was difficult to navigate.

How I Grew

- I feel more confident in my understanding of zero-trust architecture, end-to-end encryption and cryptography, and also generally creating solutions with minimal guidance.
- I gained skills in tools such as Docker, PostgreSQL, and python modules such as cryptography, keychain and click.
- I developed a greater understanding and appreciation for the planning which goes into creating secure software solutions and the importance of adhering to the CIA triad and 5 pillars of information assurance.
- I gained a more comprehensive understanding on the importance of following the 5 pillars of information assurance and through iterative improvements to my encryption scheme saw the limitations in only following a select few. I now have more thorough intuition regarding the necessity and use case of different cryptography practices in ensuring these pillars are met and particularly reinforced theoretical knowledge introduced in the lectures in a more practical environment.

What I Changed

Initially when I began implementing my cryptography system I wanted to keep it reasonably lightweight and reflective of what I had learnt both in lectures and in independent study up until that point. So, I opted to using AES-CBC both an algorithm and mode of operation which were taught in lectures and which I thought I had a rudimentary understanding of. To a point, this approach did work, in fact, for my use case it appeared faultless. However, I did not fully anticipate the real world implications of this approach and had neglected to adhere to the 5 pillars of information assurance - notably, I had placed a large focus on confidentiality and had neglected authentication and integrity. I read about the importance of integrity and message authentication and determined that these qualities were essential for my database application but had not yet been implemented.

This fuelled my transition from CBC to GCM. GCM is a mode of operation which combines cryptography, authentication, and integrity in a single step during the encryption/decryption process. It was conceptually very difficult for me to begin understanding the mathematics behind this mode as it makes use of Galois fields - which I do not have any background in - but the application of the mode through the use of cryptography libraries made it reasonably simple to setup and actually streamlined my algorithms as padding was no longer required, and I did not need the additional overhead of creating custom MACs.
For my basic database portion of the project, I had covered the pillars which I deemed the most essential, mainly confidentiality, authentication, and integrity. However, in completing the messaging service I encountered the issue of neglecting non-repudiation. Although AES and GCM protect the data and ensure it is not tampered with, my application lacked user message guarantees. To fix this I had to modify the inherent workflow of my encryption and decryption process; rather than add a new mode of operation, I needed to implement signing functionality using the user's private key to verify their identity. This added a new layer of complexity and abstraction as I now added the signature to the message payload prior to encryption, it did however guarantee non-repudiation.

These iterative improvements are reflective of two key principles I learned throughout this project, notably, the importance of security by design and security in layer. As I improved my design, I learned how the different pillars of information assurance form layers around a message to ensure it is suitably protected - confidentiality is not enough, integrity, authentication, and non-repudiation are equally as important. I have also learned the importance of security by design through my own difficulties with not anticipating the whole problem from the start - experiencing the troubles that arise when using and modifying complex, modern yet authenticated encryption modes to model changing conditions. Most importantly I internalized and gained a deeper appreciation of the security mindset and approaching problems with a security-first mindset.

Conclusion

My project, avoDB, demonstrates the concepts of client-side, end-to-end encryption and security by design in the form of a CLI-based DBaaS program. It achieves my goal of introducing a zero-trust backend structure and ensuring that the 5 pillars of informational assurance are followed. This experience was very valuable in allowing me to apply my theoretical knowledge and participate in self-guided learning in the security field.

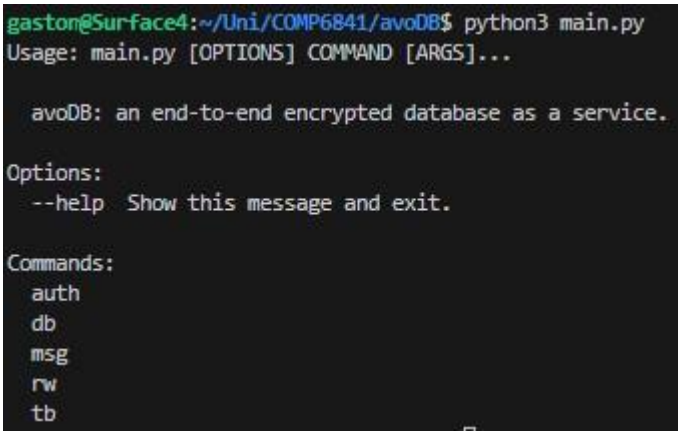
Evidence of Outcomes and Achievements

This section provides screenshots of the main interface an d corresponding output of my program along with code snippets of the different types of database operations in my application, including:

- Creation
- Selection
- Deletion

I also provided some evidence of my encryption and session management scheme.

The primary entry point to my program is through the main.py file which prompts the user the following choices:



Each set of cli options and commands is created like so:

```python
import click

from api.routes.auth import *

@click.group()
def auth():
    pass

@click.command(help="--username <*username*> --password <*password*>")
@click.option('--username', prompt=True, help="your username")
@click.option('--password', prompt=True, hide_input=True, help="your password")
def login(username, password):
    loginFunc(username, password)

@click.command(help="--username <*username*> --password <*password*>")
@click.option('--username', prompt=True, help="your username")
@click.option('--password', prompt=True, hide_input=True, help="your password")
def register(username, password):
    registerFunc(username, password)

@click.command(help="")
def userList():
    userListFunc()

@click.command(help="")
def logout():
    logoutFunc()

auth.add_command(login)
auth.add_command(register)
auth.add_command(logout)
auth.add_command(userList)
```

1. User Authentication

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py auth
Usage: main.py auth [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  login     --username <*username*> --password <*password*>
  logout
  register  --username <*username*> --password <*password*>
  userlist
```

1.1. Registration

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py auth register
Username: test
Password:
Registration Successful
successfully added credentials to keyring
```

```python
def registerFunc(username, password):
  # 1. query db to check if username exists
  # 2 create uuid
  # 3. create key pairs
  # 4. serialize key pairs
  # 5. create kdf salt
  # 6. create pk iv
  # 7. hash password
  # 8. encrypt private key
  # 9. save keys for session

  checkIfLoggedIn()

  userNameExists = checkUsernameExits(username)
  if userNameExists:
    print('Username already exists, please choose another')
    return False

  userId = generateUserId()
  privateKey, publicKey = generateKeyPair()
  privateSerialisedKey, publicSerialisedKey = serialiseKeyPair(publicKey, privateKey, password)
  salt = generateSalt()
  iv = generateIV()
  hashedPassword = hashPassword(password)
  encryptedPrivateKey = encryptPrivateKey(privateSerialisedKey, password, salt, iv)

  encodedEncryptedPrivateKey = base64.b64encode(encryptedPrivateKey).decode('utf-8')
  encodedPublicKey = base64.b64encode(publicSerialisedKey).decode('utf-8')

  successful = addUserToDB(userId, username, hashedPassword, salt, iv, encodedEncryptedPrivateKey, encodedPublicKey, datetime.datetime.now())
  if successful:
    print('Registration Successful')
    setCredentials(userId, privateSerialisedKey, publicSerialisedKey, password, iv)
  else:
    print('Failed To Register')
```

1.2. Login

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py auth login
Username: test
Password:
Already logged in
```

```python
def loginFunc(username, password):
  # 1. query db to check if username exists
  # 2. return the hashed password from the db
  # 3. verify hashed password
  # 4. save keys for session

  checkIfLoggedIn()

  userNameExists = checkUsernameExits(username)
  if not userNameExists:
    print('Username does not exist')
    return

  userId = getUserId(username)
  hashedPassword = getHashedPassword(userId)
  if not verifyPassword(hashedPassword, password):
    print('Incorrect password given')
    return

  # get private key
  encryptedPrivateKey, publicKey, salt, iv = getUserData(userId)
  encryptedPrivateKey = base64.b64decode(encryptedPrivateKey)
  publicKey = base64.b64decode(publicKey)
  privateKey = decryptPrivateKey(encryptedPrivateKey, password, salt, iv)
  setCredentials(userId, privateKey, publicKey, password, iv)

  print('Successfully Logged in')
```

1.3. Userlist

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py auth userlist

Users:

username    userId
----------  ------------------------------------
will        64394da6-e804-4623-a8be-6265bca045fb
admin       42db0783-ff2d-43a9-8041-8f1921d992fa
test        20891b6f-092b-4482-b874-0b628f359e35
```

1.4. Logout

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py auth logout
Successfully logged out
```

2. Databases

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py db
Usage: main.py db [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  dbcreate  --name <*name*>
  dbdelete  --dbId <*dbId*>
  dblist
```

2.1. Creation

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ avodb db dbcreate
Name: myNewDb
Successfully added database
```

2.2. Deletion

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py db dbdelete
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Successfully deleted db
```

2.3. List

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py db dblist
DB_id                                dbName
-----------------------------------  --------
19a65cbb-d594-4fa7-8bd8-b0e23ac588b1  mydb
```

3. Tables

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py tb
Usage: main.py tb [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  tbcreate  --dbId <*dbId*> --name <*name*> --schema <*schema*>
  tbdelete  --dbId <*dbId*> --tbId <*tbId*>
  tblist    --dbId <*dbId*>
  tbschema  --dbId <*dbId*> --tbId <*tbId*>
```

3.1. Creation

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py tb tbcreate
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Name: mytable
Schema: name,message
Successfully added table
```

```python
def addTable(tableId, dbId, tbName, schema):
  qry = "insert into Tables(table_id, db_id, encrypted_table_name, encrypted_schema) values(%s, %s, %s, %s) returning db_id"
  cursor, connection = cursorCreation()

  try:
    cursor.execute(qry, [tableId, dbId, tbName, schema])
    connection.commit()
  except Exception as e:
    connection.rollback()
    print('Table Creation Failed: ', e)

  value = cursor.fetchone()
  cursorRemoval(cursor, connection)

  if value is None:
    print('failed to add table')
    sys.exit(1)

  print('Successfully added table')
  return True
```

3.2. Deletion

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py tb tbdelete
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Tbid: 8597132c-c065-44fe-bd31-a40d290c9f8f
Successfully deleted table
```

3.3. List

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py tb tblist
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
tableId                               tableName    tableSchema
------------------------------------  -----------  -------------
8597132c-c065-44fe-bd31-a40d290c9f8f  mytable      name,message
```

3.4. Schema

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py tb tbschema
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Tbid: 8597132c-c065-44fe-bd31-a40d290c9f8f

Schema:
 name,message
```

4. Rows

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py rw
Usage: main.py rw [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  insert    --dbId <*dbId*> --tbId <*tbId*> --data <*data*>
  rwdelete  --dbId <*dbId*> --tbId <*tbId*> --rwId <*rwId*>
  rwlist    --dbId <*dbId*> --tbId <*tbId*>
  select    --dbId <*dbId*> --tbId <*tbId*>
```

4.1. Creation

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py rw insert
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Tbid: 8597132c-c065-44fe-bd31-a40d290c9f8f
Data: will,mymessage
Successfully added row
```

```python
def rwInsertRoute(dbId, tbId, data):
    # 1. check tb belongs to user
    # 2. validate data to schema
    # 3. generate iv
    # 4. generate rowId
    # 5. encrypt data
    # 6. check follows schema
    # 7. insert
    if checkDBBelongsToUser(getUserID(), dbId) == 0:
        print('Database does not belong to you. Please choose another')
        sys.exit(1)
    if checkTBBelongsToUser(getUserID(), tbId) == 0:
        print('Table does not belong to you/this database or does not exist. Please choose another')
        sys.exit(1)

    iv = generateIV()
    rowId = generateUserId()

    encryptedMasterKey = base64.b64decode(getMasterKey(dbId))
    masterKey = decryptWithPrivateKey(getPrivateKey(), encryptedMasterKey, getPassword())

    encryptedData = encryptMessage(data, iv, masterKey)

    insertRow(rowId, tbId, encryptedData, iv)
```

4.2. Deletion

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py rw rwdelete
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Tbid: 8597132c-c065-44fe-bd31-a40d290c9f8f
Rwid: de79f91c-9a01-48c4-bbcb-dcd7e4c47411
Successfully deleted row
```

```python
def deleteRow(tbId, rwId):
  qry = "delete from Rows where table_id = %s and row_id = %s"
  cursor, connection = cursorCreation()

  try:
    cursor.execute(qry, [tbId, rwId])
    connection.commit()
  except Exception as e:
    connection.rollback()
    print('row deletion failed:', e)

  cursorRemoval(cursor, connection)

  print('Successfully deleted row')
  return True
```

4.3. Select

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py rw select
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Tbid: 8597132c-c065-44fe-bd31-a40d290c9f8f

mytable:

name    message
------  ---------
will    mymessage
blah1   blah2
```

4.4. List

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py rw rwlist
Dbid: 19a65cbb-d594-4fa7-8bd8-b0e23ac588b1
Tbid: 8597132c-c065-44fe-bd31-a40d290c9f8f

mytable:

rowId                                 name
------------------------------------  -------------
de79f91c-9a01-48c4-bbcb-dcd7e4c47411  will,mymessage
```

5. Messaging

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py msg
Usage: main.py msg [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  initiateconvo  --userId <*userId*>
  sendmsg        --message <*message*>
  viewconvos
  viewmsgs
```

5.1. Initiate Conversation

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py msg initiateconvo
Userid: 42db0783-ff2d-43a9-8041-8f1921d992fa
Successfully initiated conversation
```

5.2. Send message

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py msg sendmsg
Message: heyyyy!!!!
Message sent successfully
```

5.3. View messages

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py msg viewmsgs
admin                       Me
--------------------------  ------------------
                            heyyyy
hi
how are you?
                            i am good thank you
                            how are you
i'm good too
i think my project is done
                            heyyyy!!!!
```

```
def viewMsgsRoute():
  senderId = getUserID()
  recipientId = getConvoUserID()

  otherUser = getUsername(recipientId)

  encryptedUserSentMessages = viewMsgs(senderId, recipientId, True)

  userSentMessages = []

  for message in encryptedUserSentMessages:
    encryptedSenderKey = base64.b64decode(message[3])
    messageKey = decryptWithPrivateKey(getPrivateKey(), encryptedSenderKey, getPassword())
    decryptedMessage = decryptMessage(message[0], message[2], messageKey).decode('utf-8')
    sentAt = message[1]

    userSentMessages.append([decryptedMessage, sentAt])

  encryptedUserReceivedMessages = viewMsgs(recipientId, senderId, False)

  userReceivedMessages = []

  for message in encryptedUserReceivedMessages:
    encryptedSenderKey = base64.b64decode(message[3])
    messageKey = decryptWithPrivateKey(getPrivateKey(), encryptedSenderKey, getPassword())
    decryptedMessage = decryptMessage(message[0], message[2], messageKey).decode('utf-8')
    sentAt = message[1]

    userReceivedMessages.append([decryptedMessage, sentAt])

  taggedMessagesCurr = [('curr', timestamp, message) for message, timestamp in userSentMessages]
  taggedMessagesOther = [('other', timestamp, message) for message, timestamp in userReceivedMessages]

  combinedMessages = taggedMessagesCurr + taggedMessagesOther
  combinedMessages.sort(key=lambda x: x[1])

  rows = []
  for sender, timestamp, message in combinedMessages:
    if sender == 'curr':
      rows.append(["", message])
    else:
      rows.append([message, ""])

  print(tabulate(rows, headers=[f'{otherUser}', 'Me']))
```

1. View conversations

```
gaston@Surface4:~/Uni/COMP6841/avoDB$ python3 main.py msg viewconvos
username    userId
----------  ------------------------------------
admin       42db0783-ff2d-43a9-8041-8f1921d992fa
```

```python
def viewConvos(currUserId):
  qry = "select distinct u.username, u.user_id " \
  "from Messages m " \
  "join UsersMeta u ON ( " \
  "(m.receiver_id = u.user_id AND m.sender_id = %s) OR "\
  "(m.sender_id = u.user_id AND m.receiver_id = %s));"
  cursor, connection = cursorCreation()

  try:
    cursor.execute(qry, [currUserId, currUserId])
    connection.commit()
  except Exception as e:
    connection.rollback()
    print('Registration Failed: ', e)

  value = cursor.fetchall()
  cursorRemoval(cursor, connection)

  if value is None:
    return False

  return value
```

6. Cryptography

6.1. Symmetric Encryption

```python
def encryptMessage(data, iv, masterKey, privateKey):
  gcmKey = AESGCM(masterKey)
  iv = base64.b64decode(iv)
  deserialisedPrivateKey = deserialisePrivateKey(privateKey, getPassword())
  encoded_data = data.encode('utf-8')
  signature = getSignature(deserialisedPrivateKey, encoded_data)

  signedData = json.dumps({
    "data": base64.b64encode(encoded_data).decode('utf-8'),
    "signature": base64.b64encode(signature).decode('utf-8')
  }).encode('utf-8')

  cipherText = gcmKey.encrypt(iv, signedData, associated_data=None)

  return base64.b64encode(cipherText).decode('utf-8')
```

```
def decryptMessage(ciphertext, iv, masterKey, publicKey):
    gcmKey = AESGCM(masterKey)
    iv = base64.b64decode(iv)
    try:
        decodedText = gcmKey.decrypt(iv, base64.b64decode(ciphertext), associated_data=None)
    except InvalidTag:
        print('Tampering of ciphertext detected')
        sys.exit(1)

    deBundledData = json.loads(decodedText.decode('utf-8'))
    data = base64.b64decode(deBundledData["data"])
    signature = base64.b64decode(deBundledData["signature"])

    deserialisedPublicKey = deserialisePublicKey(publicKey)

    try:
        verifySignature(deserialisedPublicKey, signature, data)
    except InvalidSignature:
        print('Invalid signature in data')
        sys.exit(1)

    return data
```

6.2. Asymmetric Encryption

```
def encryptWithPublicKey(item, publicKey):
    deserialisedPublicKey = deserialisePublicKey(publicKey)
    encryptedItem = deserialisedPublicKey.encrypt(
        item,
        asym_padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return encryptedItem

def decryptWithPrivateKey(privateKey, encryptedItem, password):
    deserialisedPrivateKey = deserialisePrivateKey(privateKey, password)
    decryptedItem = deserialisedPrivateKey.decrypt(
        encryptedItem,
        asym_padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return decryptedItem
```

6.3. Key Generation/KDF

```python
import base64
import os
from cryptography.hazmat.primitives.asymmetric import rsa
import uuid

def generateMasterKey():
  masterKeyBytes = os.urandom(16)
  return masterKeyBytes

def generateSalt():
  salt = os.urandom(16)
  return base64.b64encode(salt).decode('utf-8')

def generateIV():
  iv = os.urandom(16)
  return base64.b64encode(iv).decode('utf-8')

def generateUserId():
  userId = uuid.uuid4()
  return str(userId)

def generateKeyPair():
  privateKey = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
  )

  publicKey = privateKey.public_key()
  return privateKey, publicKey
```

```python
import base64
from argon2.low_level import hash_secret_raw, verify_secret, Type

def derivePasswordKey(password, salt):
  hashedPassword =hash_secret_raw(
    password.encode(),
    base64.b64decode(salt),
    time_cost=2,
    memory_cost=64 * 1024,
    parallelism=4,
    hash_len=32,
    type=Type.D
  )
  return hashedPassword
```

6.4. Hashing

```python
from argon2 import PasswordHasher

hasher = PasswordHasher()

def hashPassword(password):
  hashedPassword = hasher.hash(password)
  return hashedPassword

def verifyPassword(hashedPassword, password):
  try:
    hasher.verify(hashedPassword, password.encode())
    return True
  except Exception:
    return False
```

6.5. Session Management

```
def setCredentials(userId, privateKey, publicKey, password, iv):
  keyring.set_password(servicePrK, username, privateKey.decode())
  keyring.set_password(servicePbK, username, publicKey.decode())
  keyring.set_password(serviceP, username, password)
  keyring.set_password(serviceU, username, userId)
  keyring.set_password(serviceIV, username, iv)
  print('successfully added credentials to keyring')

def initiateConvo(userId, publicKey):
  keyring.set_password(convoUserId, username, userId)
  keyring.set_password(convoPublicKey, username, publicKey)

def getPrivateKey():
  secret = keyring.get_password(servicePrK, username)
  if secret is None:
    print("not logged in, please log in first")
    sys.exit(1)

  return secret.encode()

def getPublicKey():
  secret = keyring.get_password(servicePbK, username)
  if secret is None:
    print("not logged in, please log in first")
    sys.exit(1)

  return secret.encode()

def getPassword():
  secret = keyring.get_password(serviceP, username)
  if secret is None:
    print("not logged in, please log in first")
    sys.exit(1)

  return secret.encode()
```

6.6. Authentication Signing

```
def getSignature(privateKey, data):
  signature = privateKey.sign(
    data,
    padding.PSS(
      mgf=padding.MGF1(hashes.SHA256()),
      salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
  )

  return signature

def verifySignature(publicKey, signature, data):
  try:
    publicKey.verify(
      signature,
      data,
      padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
      ),
      hashes.SHA256()
    )
  except InvalidSignature:
    raise
```

References

(NIST), N. , Dworkin, M. , Sonmez Turan, M. and Mouha, N. (2023), Advanced Encryption Standard (AES), Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.FIPS.197-upd1, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=936594 (Accessed June 27, 2025)

Individual Contributors, Symmetric Encryption (2013) *Symmetric encryption - Cryptography 46.0.0.dev1 documentation*. Available at: https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/#cryptography.hazmat.primitives.ciphers.algorithms.AES256 (Accessed: 27 June 2025).

Aumasson, Jean-P. (no date) *Password hashing competition*, *Password Hashing Competition*. Available at: https://www.password-hashing.net/ (Accessed: 27 June 2025).

Biryukov, A., Dinu, D. and Khovratovich , D. (2015) *Argon2: the memory-hard function for password hashing and other applications*. Available at: https://www.password-hashing.net/argon2-specs.pdf (Accessed: 27 June 2025).

Kaliski, B., Jonsson, J. and Rusch, A. (2016) *PKCS #1: RSA cryptography specifications version 2.2* [Preprint]. doi:10.17487/rfc8017.

Key serialization (no date) *Key Serialization - Cryptography 46.0.0.dev1 documentation*. Available at: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/ (Accessed: 27 June 2025).

What are the five Pillars of Information Security (2024) *InfoSec Governance -*. Available at: https://isgovern.com/blog/what-are-the-five-pillars-of-information-security/ (Accessed: 27 June 2025).

Client-side field level encryption - database manual V7.0 - MongoDB Docs (no date) *Client-Side Field Level Encryption - Database Manual v7.0 - MongoDB Docs*. Available at: https://www.mongodb.com/docs/v7.0/core/csfle/ (Accessed: 27 June 2025).

Hindi, R. (2025) *Announcing the Zama Confidential Blockchain Protocol and Our Series B*, *Zama.ai*. Available at: https://www.zama.ai/post/announcing-the-zama-confidential-blockchain-protocol (Accessed: 27 June 2025).

Andersen, L.R. (2025) ‘What is a trust boundary?’, *moxso.com*, 1 January. Available at: https://moxso.com/blog/what-is-a-trust-boundary (Accessed: 27 June 2025).

Moroz, I. (no date) *Argon2*, *PyPI*. Available at: https://pypi.org/project/argon2/ (Accessed: 27 June 2025).

Daemen, J. and Rijmen, V., 1999. AES proposal: Rijndael - Available at: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf (Accessed: 09/07/2025)

McGrew, D. and Viega, J., 2004. The Galois/counter mode of operation (GCM). submission to NIST Modes of Operation Process, 20, pp.0278-0070. Available at: https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf (Accessed 09/07/2025)

Wikipedia. (2020). Optimal asymmetric encryption padding. [online] Available at: https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding [Accessed 9 Jul. 2025].