pman

Architecture and design description

09.04.2021

Contents

1 Introduction

This documents serves as an introduction to the architecture and design of `pman`, which is a command-line based password manager for Linux platforms. `pman` will be written in C and it revolves around managing a file-based password database that the user can manage with different commands.

Before the introduction of the architecture, the build system of `pman` is shortly described. This will include the chosen build system, build tools, some relevant compiler options, and the different linters and static analyzers used in the project.

Given the relatively simplicity `pman` as a program, the architecture is succinct and contains only a few critical architectural views. Despite this, making an architecture design is critical, as then secure design and, most importantly, secure handling of passwords can be emphasized in the construction of `pman` as early as possible.

The utilized architecture views consist of activity, logical and deployment views. The activity view was decided instead of the usual use-case view due to only real user being the user of the command-line. Logical view was an obvious decision given the security requirements described in the previous paragraph. Finally, the deployment view provides a look in to the different libraries that will be built as a part of the `pman` project.

At the very end, this document will also go in to the detailed design of `pman`, which will include detailed interfaces of the different components described in the architecture description including the component's respective interface documentations. Even though C does not support classes, to which UML is quite heavily biased to, class diagrams will be utilized in the interface descriptions.

2 Project structure and build tools

2.1 CMake build system

CMake was an rather obvious choice for a cross-platform build system, as the only real other candidate would have been Make. Modern CMake is well integrated with different C compiler options and tools.

2.1.1 Folder structure

The pman project is split to its architectural components using concise folder structure. Each subfolder/library will contain their own *CMakeLists.txt* that specifies how the possible source files in the folder are built. The folders of the project with their purposes are listed in table 1.

Folder	Purpose
cmake/	CMake scripts and modules
include/	Header files (*.h*)
src/	Source files (*.c*)
tests/	Unit test files

Table 1: Folder structure of pman

2.1.2 Clang and C17

The C compiler for the project was chosen to be Clang due to its superior tool capabilities and integration that is heavily utilized. Clang provides many highly capable static analysis and linting tools, which for a large C project are more than welcome.

The newest complete C standard, C17, was chosen due to all of the useful features provided in C11. pman does not have portability constraints, so choosing the newest C standard poses no clear downsides.

2.1.3 CTest and unit testing

CTest is a unit test driver integrated with CMake, so it was a self-explanatory choice for the project. CTest implementation for the project can be seen in *tests/* subfolder of the project.

The pman project utilizes unit testing heavily, which is why a Unity library is used to extend the capabilities of the unit tests. Unity provides easy-to-use unit testing interface similar to Google Test etc. while also fully supporting C.

2.2 Static analysis and linting

As security is one of the key design principles of pman, the build system integrates several tools to facilitate this principle. Static analysis and linting are used during the

building process to eliminate as many bugs and potential security risks as possible, as early as possible.

2.2.1 Compiler warnings

All C compilers have a way of generating warnings from a possibly buggy code, and Clang is no exception. To utilize warnings to their full effect, the `pman` build process enables every single feasible warning provided by the Clang compiler. Additionally, every warning produced by the compiler is transformed in to an error, which forces fixing each and every one of them.

2.2.2 Clang-Tidy linter

`clang-tidy` is a linting tool for C/C++, which provides an extensible framework for diagnosing typical programming errors. These include style violations, interface misuse, and bugs.

`clang-tidy` is integrated in to the build process of `pman`, and all of its warnings are interpreted as errors. Additionally, the project also supports using CLion IDE, which further integrates with `clang-tidy`.

The utilized checks and project styling parameters are listed in the *.clang-tidy* file at the root of the project. Along with enabling almost all possible checks, a consistent naming scheme is enforced.

2.2.3 Clang Static Analyzer

Along with the `clang-tidy` mentioned in the previous section, Clang also provides a powerful static analyzer purpose-built for C/C++code. For such a small project as `pman`, the only minor drawback is increase in compilation-time, which is well worth the benefits.

The Clang Static Analyzer is integrates with `clang-tidy`, and therefore, is also ran during the normal build process. The project also implements a script for running the static analyzer separately in *scripts/* folder.

3 Architecture

3.1 Process view

This section first introduces the activity diagrams of a few common tasks, and afterwards goes through the main commands of `pman`. The activity diagrams seek to explain the basic workflow of each command, for example what is done in event of an error, or when authentication fails etc.

The activity diagrams should act as basis for the implementation of the different commands, due to the previously mentioned reasons. Clear error handling flow is critical for the secure usage of `pman`.

3.1.1 Common tasks

Prompting database password from the command-line and authentication are one of the more commonly used operations. For this reason, these operations were separated out from the main commands.

Database password prompt is done whenever, for example, `pman` spots that the login has timed-out, or when the user has not logged in at all while issuing a command. By default the password prompt must hide the echo from the terminal. The diagram can be seen in figure 1.

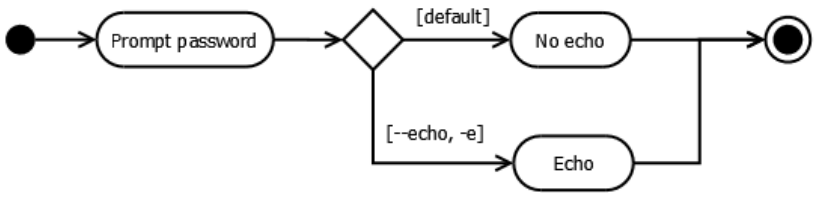

Figure 1: Activity diagram for password prompt

Much like password prompt, authentication is a very common operation as well. As `pman` allows for saving logins for a set period of time, the authentication is non-trivial, which is why it is abstracted out of the main commands. The figure 2 introduces the authentication workflow.

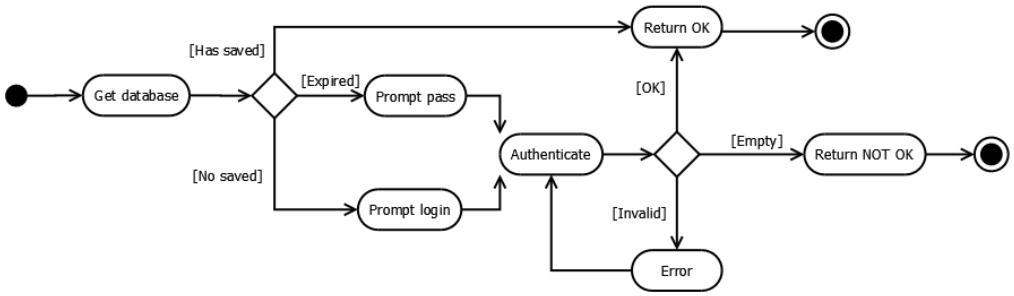

Figure 2: Activity diagram for the authentication process

Accessing the database is something which almost every command does at some point. The database implements a lot of mechanisms to make breaking into it that much more difficult, and all of these operations must be abstracted away.

3.1.2 Main commands

The main commands are commands that are issued to the `pman` in the command line, which tells the what the user wants to do. Examples of main commands are initializing a database, adding an entry to database, or listing a database. These commands form the main workflow of `pman`.

The database initialization command `new` is the very first command that the user will use with `pman`. The activity diagram for this command is described in figure 3.

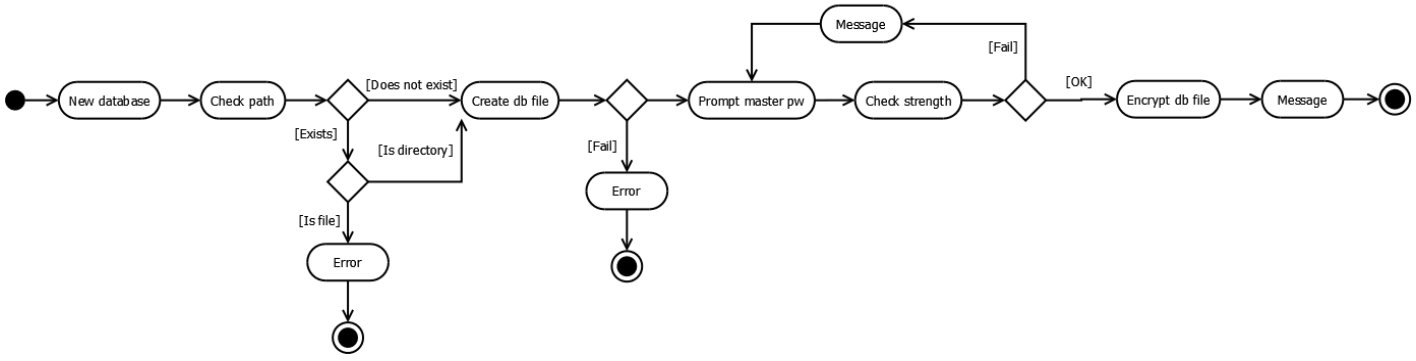

Figure 3: Activity diagram for the database initialization

If the initialization of the database is successful, the user will `login` to the database either implicitly (by issuing a command), or explicitly by using the `login` command. The activity diagram for `login` command can be seen in figure 4.

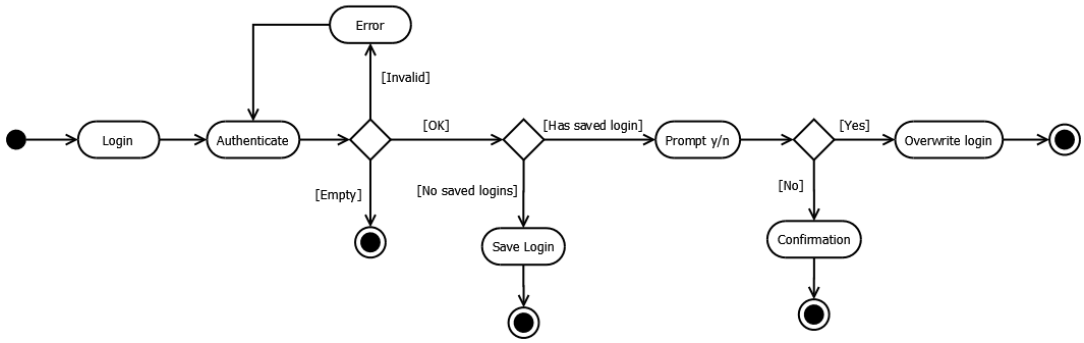

Figure 4: Activity diagram for logging in to the database

The user can add entries to the database with the `add` command. The workflow of this command is described in figure 5.

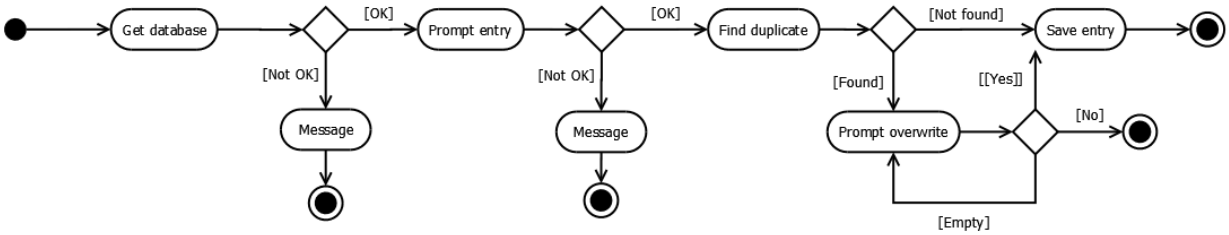

Figure 5: Activity diagram for adding a new entry

To fetch a password matching a username, the user issues a `get` command. The workflow of the `get` command can be seen in figure 6.

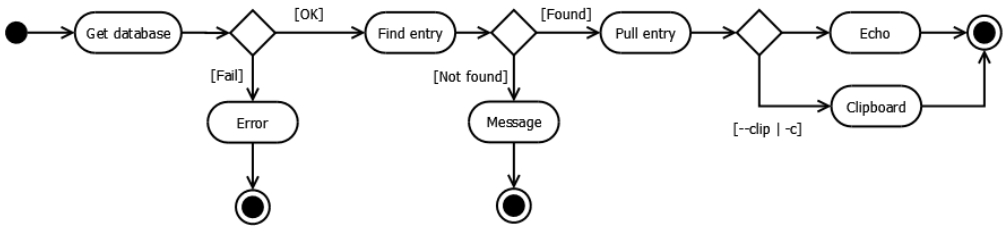

Figure 6: Activity diagram for fetching a password

Afterwards, the user might want to modify the just added entry in the database. This can be performed with the `edit` command and its activity diagram is visible in figure 7.

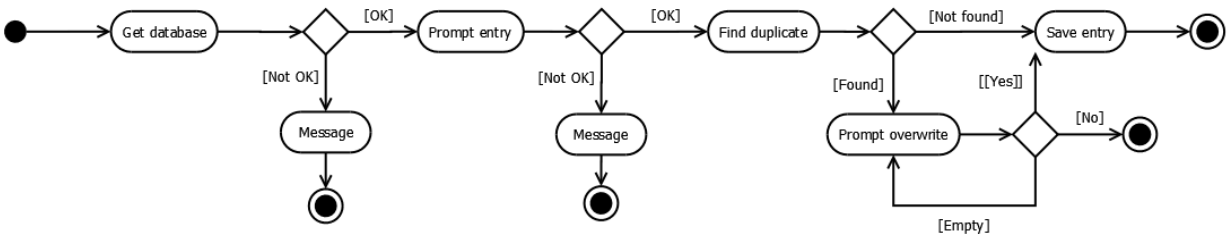

Figure 7: Activity diagram for editing an existing entry

If the user finds that there is an unneeded entry in the database, the user can issue `del` command on the entry. This command is described in figure 8.

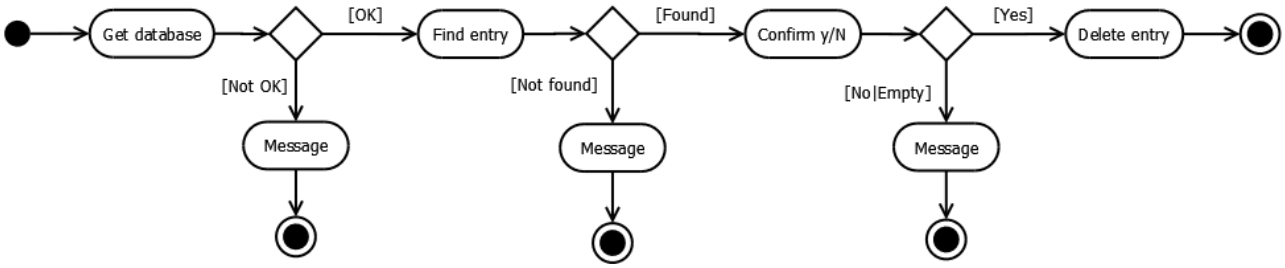

Figure 8: Activity diagram for deleting an entry

Finally, when the user has gathered enough entries in the database, he might want to view all of the entries. This can be done with the `list` command, which is introduced in the figure 9.

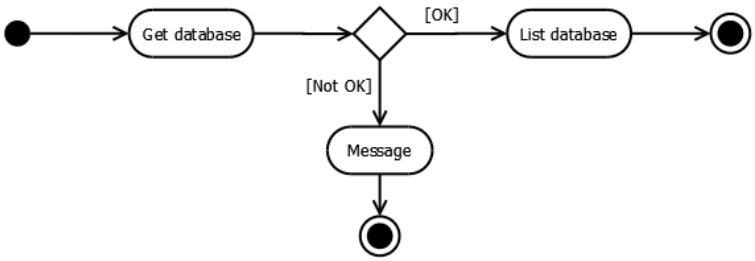

Figure 9: Activity diagram for listing entries in the database

The activity diagram for the `list` command is really simple, as the implementation details of the command are not a priority.

3.2 Implementation view

The main goal of logical view is to present the components of `pman` and define the interface through which they communicate. The primary decisional forces here are isolating components that require one of the following: emphasis on security, cross-platform requirements, likely to change over time, or the component can be generalized to a library.

By making an effort to separate the components of the system this early, we can minimize the effects of changes to the implementation. That is, as long as the interfaces are clearly defined. The logical architecture view of `pman` is visible in figure 10.

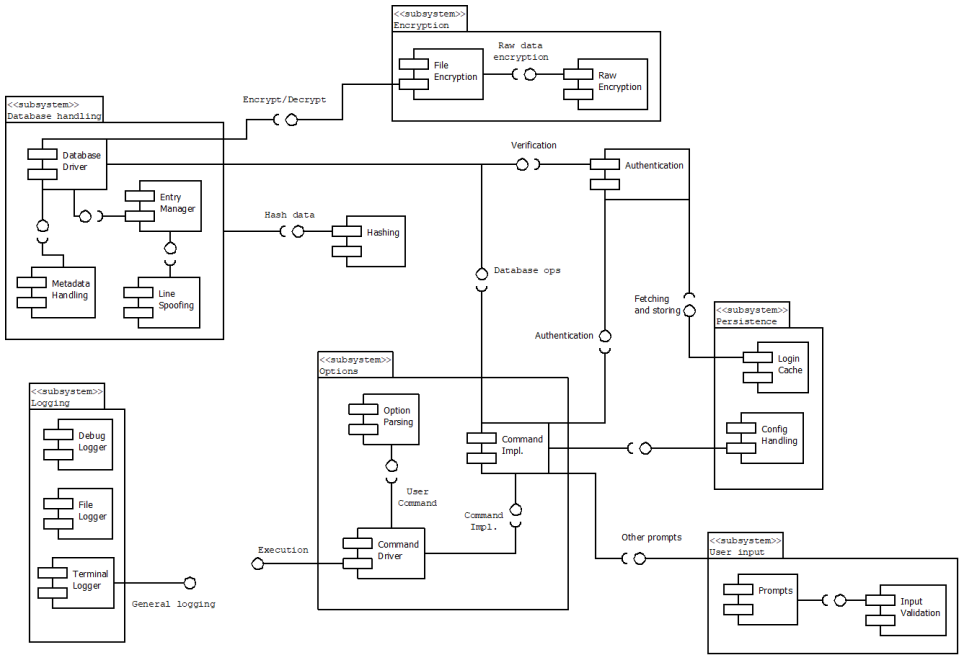

Figure 10: Logical view of the architecture of `pman`

As can be seen from figure 10, the implementation structure of `pman` is split in to seperate subsystems, which each consist of several different components. The most notable subsystems are database handling and options parsing. This is not a surprise given that secure database handling and flexible option handling are one the key aspects of `pman`.

3.3 Deployment view

The deployment view seeks to define how `pman` is deployed as whole, meaning what kind of libraries will be built and what will be linked to the executable. The deployment view also shows the environment the `pman` is designed to be ran in.

The chosen programming language C drives the architectural view quite heavily, as the entire model of deploying libraries and executables is innate to C. Many of the libraries linked with `pman` are also C libraries. The deployment view can be seen in figure 11.

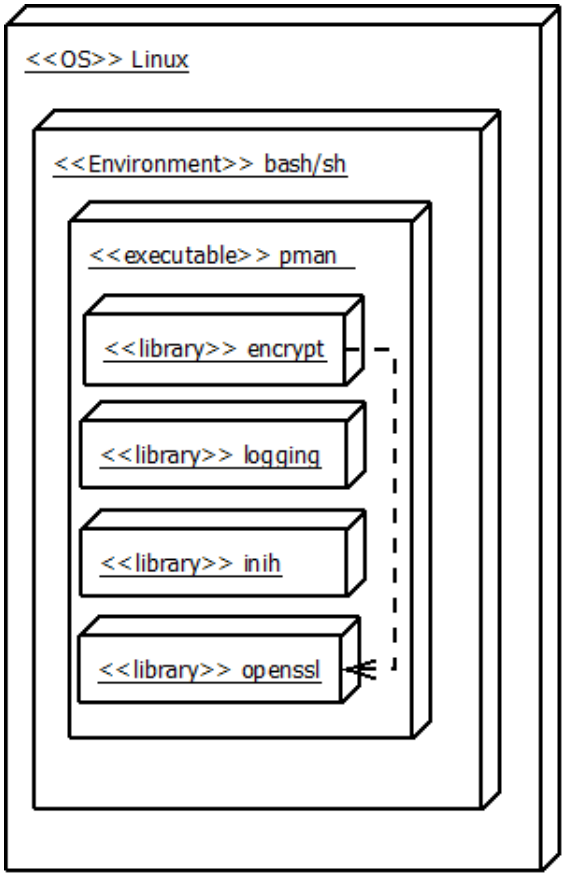

Figure 11: Deployment view of the architecture of `pman`

As is apparent in figure 11, the execution environment is almost solely consist of terminal utilities innate to Linux. Other than that, the `pman` executable is linked with cryptographic utilities like `openssl` and configuration file libraries `inih`. The build process also produces its own libraries like `logging` and `encrypt`.

4 Detailed design

The following section will go in to the detail of the design of the different components in the architecture of `pman`. The detailed design will include the different interfaces (at function level) and the different structures that the components will implement.

One should bare in mind that even though C does not support the object oriented paradigm, the detailed design will be depicted with class diagrams. In the actual implementation the class methods of will be free functions acting on common data (`structs`).

4.1 Encryption

The design of encryption was broken into two parts in the architecture: File Encryption and Raw Encryption. These two components accomodate the needs of `pman` when it comes to encryption and decryption. The detailed design of these two components can be seen in figure 12.

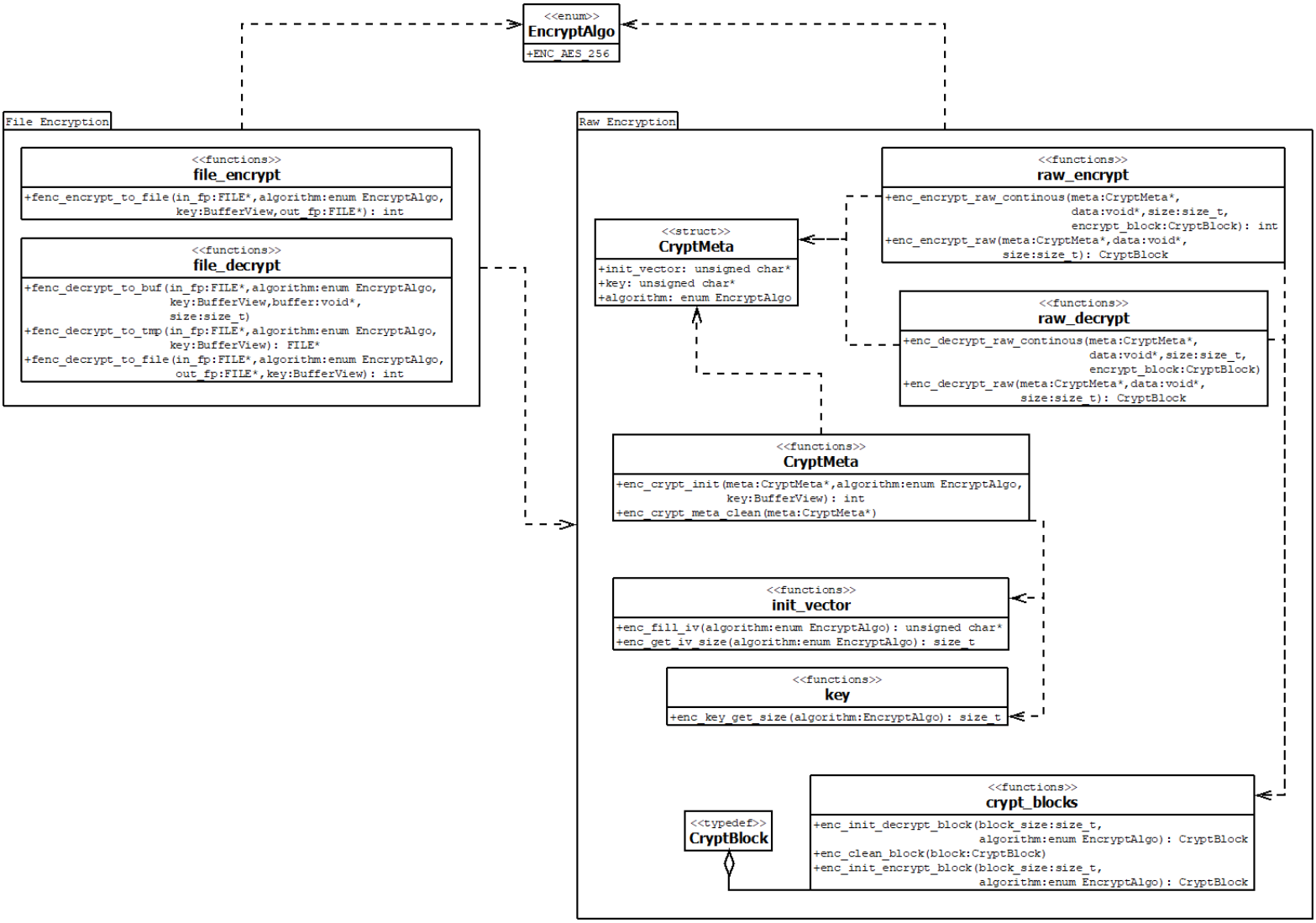

Figure 12: Interfaces and design of the encryption components

The class diagrams of figure 12 show that the file encryption side offers quite a minimal interface compared to the raw encryption side. This is because data encryption itself holds a lot of implementation details (initialization vectors, different block sizes etc.) that need not to be revealed when all the user wants to do is encrypt or decrypt a file.

The basic idea of the design is that the user provides the password database (file) that he wants to encrypt, algorithm and the key that will be used. From here the implementation will construct the `CryptMeta` object that is used with the raw encryption functions, which will contain the key, the initialization vector, and the requested algorithm.

Then, depending on the size of the requested file encryption/decryption the functions will either construct a reusable `CryptBlock` with an optimal size, or use the one-time encrypt/decrypt functions provided by the interface (continous and non-continous). The continous ones were designed to accomodate extremely large databases, where constantly allocating and deallocating new blocks would not be feasible.

4.2 Database handling

4.3 Hashing

Hashing inherits many of the same design aspects from encryption, as the operations are quite similar with, of course, hashing being quite simpler. The design of the hashing component can be seen in 13.

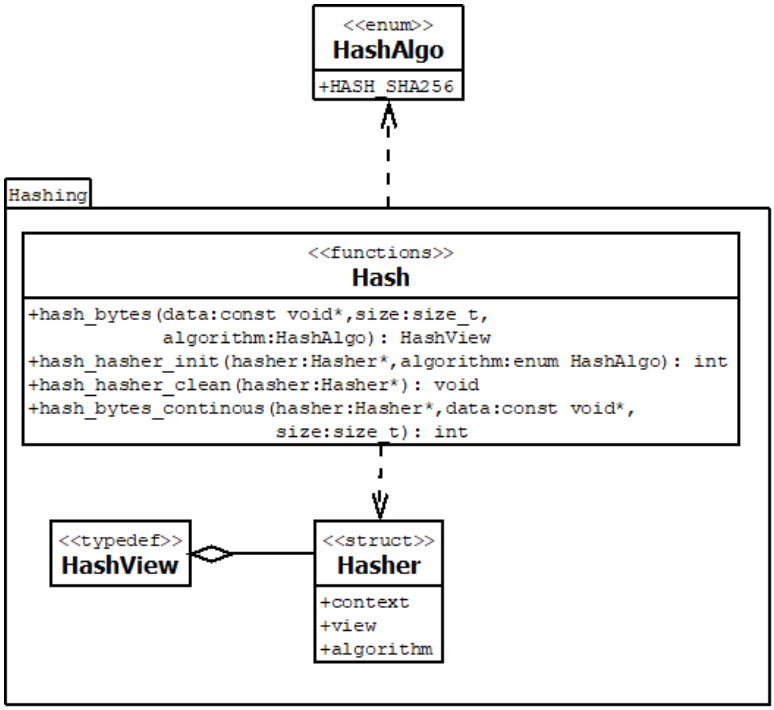

Figure 13: Interfaces and design of the hashing component

Both one-time and continous hashing are supported as with encryption. The user will either construct a `Hasher` object to support continous hashing, or use the one-time hashing provided with function `hash_bytes`.

4.4 Authentication

4.5 Persistence

4.6 Options

4.7 User input

4.8 Logging