POLITECNICO DI TORINO

Department of control and computer engineering

Master degree course in Computer Engineering

Master Degree Thesis
in
Embedded Systems

A Secure Password Wallet based on the SEcube framework

Supervisors
Prof. Paolo Ernesto Prinetto
Dr. Giuseppe Airò Farulla

Candidates
Walter Gallego Gómez
matricola: s225140

July 2017-2018

This work is subject to the Creative Commons Licence

To my mom
† In memory of my father

Summary

Nowadays, having a large quantity of digital passwords is the norm, and as their number increases, it becomes impossible to memorize all of them, specially in the case of strong passwords which are very long and/or very random, and therefore hard to remember. These has led users to relay on software applications to save their passwords, the most common cases being web browsers and password wallets. The drawback of this approach is that security may be compromised, since all the passwords are stored in the same place an attacker could gain access to them. For users particularly interested in ensuring the security of their systems, this software based approach may not be acceptable. This work presents an alternative solution based on the SEcube™ framework that guarantees the security of the stored passwords.

The SEcube™ (Secure Environment cube) framework consist of an open source security-oriented hardware platform designed by the Blu5 Group, and a set of open source software libraries developed by European research institutions. The core of the framework is the SEcube chip, which integrates three key security elements in a single package: A fast floating-point Cortex-M4 CPU, a high-performance FPGA and an EAL5+ certified Security Controller (Smart Card). This elements, in conjunction with the software libraries allows developers with little to no knowledge in information security to implement of highly reliable security applications.

The desktop application developed in this work, named **SEcubeWallet**, was written in C/C++ and Qt. The application runs in a computer, denominated host, and the SEcube chip, known as device has to be connected to it via USB. The host request for security operations to the device, using the SEcube software libraries. Each device has a hard written master password, and in order to use it, it is necessary to perform an authentication procedure entering said password from the host.

The wallets are managed using SecureSQlite, one of the SEcube libraries, which wraps the functionalities of the SQlite standard to create SEcube secured databases. In short, the data of interest is encrypted using the SEcube

device and it is stored as a file in the host. This encrypted file can only be decrypted if the SEcube device is connected and if the user authenticates using the master password. As the core operations are performed by the device, not by the host, the encryption/decryption can be done in any computer where an appropriate version of Qt is installed and the device is connected.

As front end, the application presents the user a pleasant and intuitive graphical user interface. With it, the user can easily create, delete, open, and modify password wallets. The GUI is configurable and as it runs in Qt is cross-platform. Additionally, the application allows the user to generate passwords using the PwGen library, and passphrases using an ad hoc developed function. This, together with the possibility to realistically check the entropy of the generated pins using the zxcvbn dropbox library, encourages the users to secure their systems with strong passwords/passphrases.

In conclusion, SEcubWallet is an excellent application for the storing and management of passwords. As it relays on the SEcube framework it is trustworthy and the information is virtually impossible to steal. Thanks to the developed GUI it is easy to use, and it offers some interesting functionalities to increase the user experience.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Application Development

1.1 Useful concepts definition

More detailed explanations of the following concepts will be given later on, but it is helpful to shortly define them here so this work is more easily readable.

1.1.1 Wallet

A password wallet is a digital form of securely keeping passwords and some meta information. In this work, a Wallet is stored as a Sqlite DataBase. A wallet can have as many tables as the user wants. For instance an user could have a table for storing social media passwords, another one for work-related passwords and a last one for credit cards and bank accounts passwords. Finally, Each table has a set of defined fields (Username, Domain, Password, Date, Description).

1.1.2 SEcube

SEcube is a custom chip produced by the Blu5 group [3] that integrates an ARM CPU, a FPGA and a SmartCard. The chip is specifically designed for security purposes, allowing developers to implement encryption/decryption functions that are executed fast and are guaranteed to be reliable. The chip can be connected to a PC by USB, Ethernet etc...., so an application running on the PC can use the SEcube to encrypt/decrypt some date.

1.1.3 SEcube SDK

The SEcube Open SDK is a set of open libraries designed to make the development of applications using SEcube more convenient. There are two types of Libraries: Host side (PC) and device side (SEcube) libraries. In general host side functions make requests to device side functions and wait for their response. Moreover, Libraries are divided in four and two hierarchical levels of abstraction, for host and device side respectively. Level0 and Level1 are the lowest levels and are present in both host and device. L0 provides the communication protocols while L1 provides basic security APIs.

1.1.4 SEfile

SEfile is a Level 2 API that allows users to encrypt/decrypt data (files in the hard disk), so they can only be read when the SEcube chip is connected to the PC. When the SEcube is not connected, it is impossible to read the files as the information necessary to decrypt them is physically stored in the SEcube.

1.1.5 Sqlite DB

1.2 Design

The purpose of this section is to give the reader a clear overview of how the application works in general terms.

A simplified design architecture is displayed in figure 1.1. It shows which Software Libraries are used by the application and when it uses them.

The following is a brief explanation of these Libraries and they usage. More details about each Library and why they were chosen are given in section 1.3 and section 1.4 deals with the actual implementation.

1.2.1 L0 and L1 Authentication libraries

When the user starts the application the first steps to perform are to open the communication with the device using Level0 functions from both host and device, and to authenticate the user by checking the login pin, using Level1 functions (again, from both sides). In the basic design diagram we can see how the SEcubeWallet uses the authentication functions and they in turn communicate with the SEcube chip.

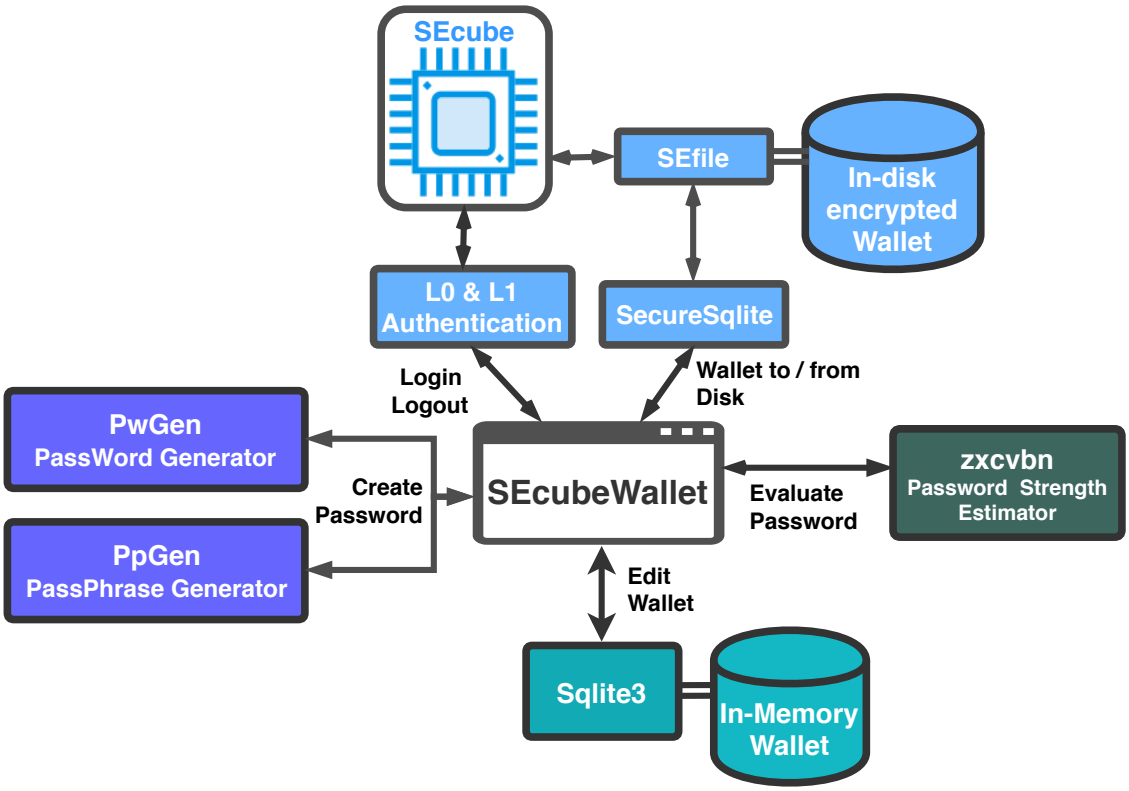


Figure 1.1: Basic Design: Used Libraries

1.2.2 SecureSqlite3

As explained before, Wallets are stored as Sqlite DataBases. Fortunately, the SEcube SDK already provides a Level2 API for creating and managing encrypted Sqlite DBs, called SecureSqlite3. This API exploits the SEfile API to wrap some of the functions of the original Sqlite3 library, avoiding OS calls.

SecureSqlite allows to create/edit/save/open databases that when written to the disk are encrypted and can only be read when the SEcube is connected. Additionally, as it is implemented using wrappers, the developer only needs to include the source files in the project and can manage SecureSqlite DB with the same functions used for regular Sqlite DB.

We can see in the diagram the SEcubeWallet application using SecureSqlite to read/write the encrypted wallet stored in disk.

1.2.3 Sqlite3

Because the use of SecureSqlite involves a call to the SEcube, a regular Sqlite3 DB is also used, but this DB is never saved to the disk. Sqlite3 allows for the creation of an In-Memory DB, i.e. a DB whose content is always in the application's memory space and is therefore secured by the operating system.

The In-memory DB is used for editing. When the user want to save the wallet, i.e. write it to the disk, the contents of the In-Memory DB are dumped to the encrypted SecureSqlite DB. When the user opens a wallet from the disk, the reverse process occurs.

With the In-memory DB, unnecessary calls to the SEcube are avoided while maintaining the contents secured.

1.2.4 PassWord Generator

As the purpose of the application is to securely store passwords, said passwords should be as strong as possible. It does not make sense to protect a password that can be easily cracked by a hacker using brute force. That is why the application also includes a Password Generator.

PwGen is an open source library that generates passwords, that can either be easy to remember, or completely random. Random passwords are more secure, but as the are difficult to remember, their use only makes sense when the user stores them in a wallet manager. Among other aspects, length and characters used (Numbers, Upper cases) can be configured too.

When the user is adding a new entry to a wallet, they can chose to enter a password or to automatically generate one.

1.2.5 PassPhrase Generator

In addition to the Password Generator PwGen, the user has the possibility to generate PassPhrases instead. PassPhrases are a popular alternative because they are easier to memorize and therefore can be longer, which in turns make them more secure. Further details about the usefulness of PassPhrases and how they compare against regular Passwords is given in section 1.3.5.

Although the PassPhrase Generator is not a library per se, it was included in the diagram because of its close relation with the PwGen and zxcvbn libraries. It was developed by the author, and it works by selecting random words out of dictionary files. The main options of the generator are which dictionaries to use, how many words to select and the minimum length of each word.

1.2.6 Strength Estimator

To give the users feedback on how good the password they are about to store is, the application uses the open source project zxcvbn to give an estimation of the passwords entropy, and how long it would take for a hacker to break it. zxcvbn bases its calculation in a number of factors, among them if the password is a common word, or a combination of them, a last name, a date, or letters close to each other in a keyboard (thus the name zxcvbn). With the estimator users are encouraged to create good passwords that are not necessarily completely random and difficult to remember, or annoying to type.

To recap, these are the key aspects of the used libraries:

- To start the connection with the SEcube, the Level0 library is used
- To authenticate the user, by checking if the entered login pin is the same as the pin stored in the SEcube, the Level1 library is used.
- An in-memory database is used for editing the wallet.
- An encrypted in-disk data base is used for storing the wallet in disk.
- The application includes a password generator with several options.
- The application also includes a password strength estimator, so the user has an idea of how good their passwords are.

1.3 Frameworks, Libraries and software tools

As explained in the previous section, the core of the design is the use of the SEcube chip to perform security operations in order to encrypt/decrypt some data stored in the host (PC). The requests to the chip are made from the Qt application developed in this work, which runs in the host. Said application exploits the existing C libraries SEfile and SecureSqlite to ease the communication with the SEcube. Additionally, the application also makes use of a random password generator PwGen and a strength estimator zxcvbn open libraries.

In the following sections a review of the SEcube platform's hardware and software components is given. Then a brief explanation of the C++/Qt framework and why it was chosen. Finally the additional used libraries are presented.

1.3.1 The SEcube framework

"The SEcube™ (Secure Environment cube) Open Security Platform is an open source security oriented hardware and software platform, designed and constructed with ease of integration and service-orientation in mind. The hardware part of the platform was originally designed by Blu5 Group [3], whereas the software libraries stem from a strong cooperation among international research institutions." [10].

The main **hardware** products, explained in detail in the following sections, are:

- The Chip, named SEcube Chip, or simply **SEcube**
- The Development Board, named **SEcube DevKit**
- The USB Stick, named **USEcube Stick**.

The SEcube chip is the main hardware component, and both the devkit and USB Stick are designed around it.The Development Board provides several communication protocols as well as debugging capabilities. For the final product the board would be of course too inconvenient to carry, and instead the USEcube Stick is preferred.

The SEcube Chip

"The SEcube™ (Secure Environment cube) is a powerful chip which integrates three key security elements in a single package. A fast floating-point Cortex-M4 **CPU**, a high-performance **FPGA** and an EAL5+ certified Security Controller (**Smart Card**). The result of this innovative combination gives an extremely versatile secure environment in a single SoC, in which developers can rapidly implement complex applications and appliances. ... The SEcube™ is the ultimate solution for high-end design, delivering integration of a flexible, configurable and certified secure element." [9]

We can then see the SEcube chip as a powerful device offering the flexibility of an ARM CPU, the speed of an FPGA and the reliable security of a certified Smart Card, all bounded together and easily integrated in any project thanks to the available communication protocols, among them USB, UART, Ethernet and JTAG.

The chip includes a true random number generator which relies in 240 noise seeds, all physical and therefore unpredictable. This allows the creation of true random noise. Additionally the user can choose what type of noise they want to generate, for instance white or Fourier noise.

In figure 1.2 a simplified SEcube architecture is shown.

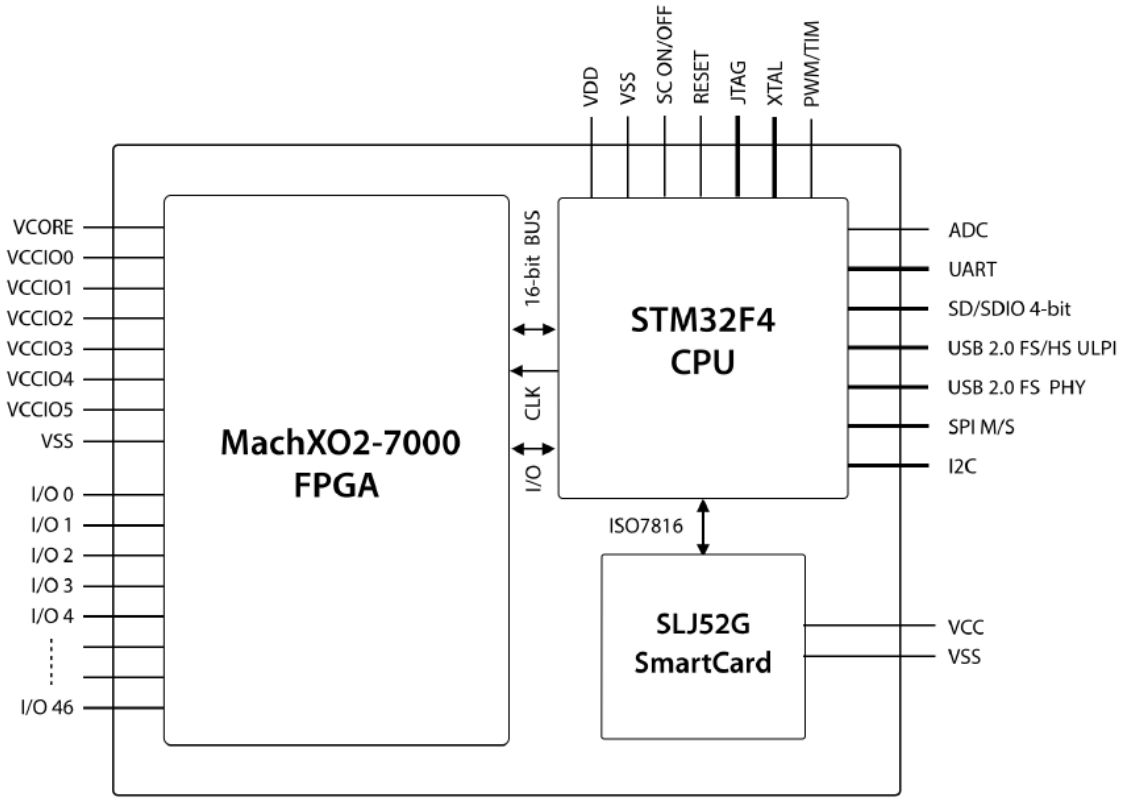


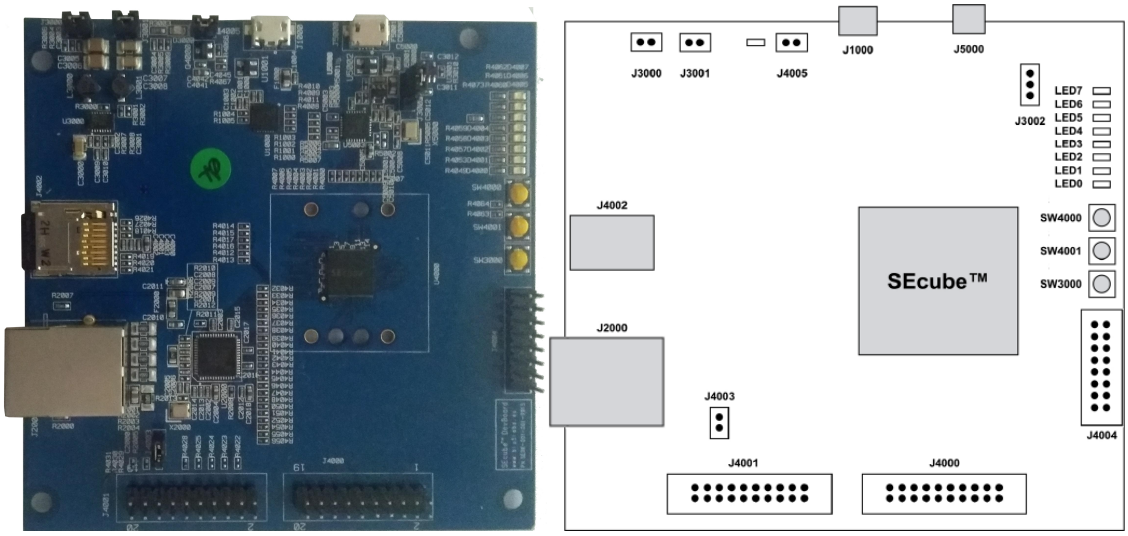
Figure 1.2: SEcube Block Diagram

Development board: The SEcube DevKit

The development board integrates the SEcube chip with several peripherals that allow the user to easily communicate, program and debug. (Figure 1.3)

The main peripherals in the SEcube devkit are:

- **J1000:** USB 2.0 to UART
- **J2000:** Ethernet 10/100 socket
- **J4000:** SEcube embedded FPGA and CPU GPIOs
- **J4001:** SEcube embedded CPU JTAG
- **J4002:** microSD card
- **J4004:** SEcube embedded FPGA and CPU GPIOs



(a) (b)

Figure 1.3: SEcube Devkit

- **J5000:** USB 2.0 High Speed
- **LEDx:** Leds
- **SWx00y:** Switches

Final product: USEcube Stick

For the final product, its is desired that the user carries all the SEcube functionalities in a small and convenient package, so they can encrypt/decrypt the passwords in any PC by just connecting the USEcube Stick and running the SEcubeWallet application.

The USEcube Stick is compatible with any Operating System and the SEcube functionalities are easily exposed to applications and services without installing any driver.

The USEcube offers only the strictly required components: The SEcube chip, a USB 2.0 High-Speed interface and an SDcard socket. See Figure 1.4 for more details.

Since the USEcube Stick storage capability is based on a external microSD card, the security of the system is improved, as this allows to have a separation of encrypted data from the encryptor/decryptor. Additionally, both the size and the speed can be tuned per the user requirement and can be changed at any time, just replacing the microSD, without buying a new

USEcube Stick. The microSD card socket is embedded in the USB connector allowing to save space making the USEcube Stick very compact and, at the same time dust and water-resistant. Since the USEcube Stick is not provided with the JTAG interface, to inject the firmware previously developed and tested on the SEcube DevKit, all the devices come with an embedded secure boot loader.

(a) (b)

Figure 1.4: USEcube Stick

L2 Security APIs

"The software libraries and design environment allow developers who are not willing or able to produce the security APIs and protocols themselves to exploit the ready functions provided (currently as APIs and soon as services) within the SEcube platform and experience the platform as a high-security black box." [11]

"From the user/developer point of view, the APIs have been implemented targeting two nested environments depending on where physically the code runs:

- **Device-Side**, including the libraries of basic functionalities that are executed on the embedded processor of the SEcube™-based hardware device.
- **Host-Side**, containing libraries of functions executed on the host PC and interface functions for calling services and processes residing on the embedded processor of the SEcube™ device.

From the architectural point of view, the Host-Side Libraries have been implemented targeting 4 hierarchical abstraction levels, and namely:

- **Level 0:** Communication Protocol and Provisioning APIs
- **Level 1:** Basic Security APIs (Level1 Host-Side – L1)
- **Level 2:** Intermediate Security APIs (Level2 – L2)
- **Level 3:** Advanced Security APIs (Level3 – L3).

At each level, each component represents a "service" for the upper level and relies on "services" provided by the next lower level, only." [11]

"Level L2 relies on L1 services to provide the APIs for implementing more abstract secure functionalities. Typical examples include APIs for the protection of data both at rest and in-motion, or negotiating parameters (e.g., keys, algorithms) for establishing secure sessions, without being forced to understand in details all the low-level hardware and security mechanisms."[11]

L2 can be considered as the merge of two projects: **SEfile**, concerning data at rest, and **SElink**, concerning instead data at motion.

For our project we rely heavily on the development tools provided by the SEfile project, for the secure storage, usage and retrieve of data that requires a high degree of confidentiality, in our case, digital passwords.

SEfile

"SEfile targets any user that, by moving inside a secure environment, wants to perform basic operation on regular files. It must be pointed out that all encryption functionalities are demanded to the secure device in their entirety. In addition, SEfile does not expose to the host device details about what, or where it is reading/writing data: thus, the host OS, which might be untrusted, is totally unaware of what it is writing". [11].

SecureSqlite

1.3.2 Sqlite3

1.3.3 Graphical User Interface: the Qt framework

The application's graphical user interface was developed using the **Qt framework**, version 5.8.0.

"Qt is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS and others. Qt is not a programming language on its own. It is a framework written in C++. A preprocessor, the MOC (Meta-Object Compiler), is used to extend the C++ language with features like signals and slots. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and applications/libraries using it can be compiled by any standard compliant C++ compiler like Clang, GCC, ICC, MinGW and MSVC".[1]

For writing, compiling and debugging source code, the IDE **Qt Creator**, version 4.2.1 was used.

"Qt Creator provides a cross-platform, complete integrated development environment (IDE) for application developers to create applications for multiple desktop, embedded, and mobile device platforms, such as Android and iOS. It is available for Linux, macOS and Windows operating systems".[4].

The reasons behind the use of Qt are as follows:

- Qt is a C++ library, and as such, allows for a seamless use of the C libraries SEfile and SecureSqlite, which are the backbone of this project.
- Qt is cross-platform, meaning the developed application can be compiled to work on any of the major OSes. In particular, the development was carried out an tested on a Linux machine, but the application should work with no problems in Windows and MacOS.
- Because of good designed and ready-to-use display items such as tables, menus and dialogues, it is possible to focus in writing the functional portions of the application without worrying too much about the GUI. And as it is open source, any Qt item can be modified and extended when it does not meet the expectations out of the box. In this project several display elements were improved, as will be seen in section **??**.
- Thanks to the multitude of functions dedicated to ease the use of C++ libraries and OS calls, one can be more productive, and the resulting code is more reliable. For instance, this project makes extensive use of such libraries, like QSqlDatabase, QString, QProcess, etc. Again, more details are given in section 1.4
- Related works by research group TESTGROUP from Politecnico di Torino

using the SEcube framework, are written in Qt. Namely SecureSqlite-Browser and SEfile_TXT, were used as base in the initial stages of development. This two projects can be found in the SEfileSDK available online [5].

- Qt is widely used, meaning it is possible to find tons of documentation, forums and additional libraries on the web. This also ensures the Qt framework will have continuous support from the developers and the community.

1.3.4 PwGen: Pronounceable Password generator

The most secure type of passwords are random ones. A random password sufficiently long is considered to be virtually unbreakable. But this rises two problems: First of all, humans are inherently bad at creating true random passwords. Second, a random password is not suited to be remembered or even used (as it probably is too annoying to type). These two reasons motivated the inclusion of a Password Generator.

pwgen is an open source program that generates human friendly passwords that are also secure. It is available in the official Linux repositories, and there is a Windows version as well, but in this work the source files where used.

"The pwgen program generates passwords which are designed to be easily memorized by humans, while being as secure as possible. Human-memorable passwords are never going to be as secure as completely completely random passwords. In particular, passwords generated by pwgen without the -s option should not be used in places where the password could be attacked via an off-line brute-force attack. On the other hand, completely randomly generated passwords have a tendency to be written down, and are subject to being compromised in that fashion" [15].

pwgen offers several options that can drastically change the type of generated password. Here is a list of the options available for users of SEcube-Wallet:

- **Length:** The desired length of the password. It is recommended to be at least 12 for non-random passwords and 8 for random ones.
- **-0, no numerals:** Don't include numbers in the generated passwords.
- **-A, no capitalize:** Don't bother to include any capital letters in the generated passwords.

- **-B, ambiguous:** Don't use characters that could be confused by the user when printed, such as 'l' and '1', or '0' or 'O'. This reduces the number of possible passwords significantly, and as such reduces the quality of the passwords. It may be useful for users who have bad vision, but in general use of this option is not recommended.
- **-c, capitalize:** Include at least one capital letter in the password.
- **-n, numerals:** Include at least one number in the password.
- **-s, secure:** Generate completely random, hard-to-memorize passwords. These should only be used for machine passwords, since otherwise it's almost guaranteed that users will simply write the password on a piece of paper taped to the monitor.
- **-v, no vowels:** Generate random passwords that do not contain vowels or numbers that might be mistaken for vowels. It provides less secure passwords to allow system administrators to not have to worry with random passwords accidentally contain offensive substrings.
- **-y, symbols:** Include at least one special character in the password.

By default pwgen behaves as if the options **-nc** were used, that is, pronounceable passwords with at least 1 capital letter and 1 number.

The strongest passwords this program can generate are obtained with the options **-ys**, as it results in random passwords with special symbols. They are very hard to remember, and as said previously, should only be used if the user is willing to open the SEcubeWallet application each time they need to use a password.

1.3.5 zxcvbn: Password strength estimation

An important feature to have in a password manager is the possibility to realistically estimate how strong a password is, i.e., how hard could it be for hackers to crack it, as there is no point in using the SEcube system to protect weak passwords, that could be easily guessed with brute force attacks. As it is out of the author expertise to write a reliable function to make this estimation, it was decided to use a trusted project developed during the dropbox hackweek event in 2012. The estimator called **zxcvbn** was originally written in JavaScript aiming for an easy integration with multiple web browsers and OS. Fortunately, the community ported the library to a wide variety of languages including Python, Ruby and C/C++. In this work

the later was used. The project is Open Source and available for free use on GitHub [7].

zxcvbn is regarded by the community as one of the most reliable and mathematically advance open source password estimators. In security forums and discussion it always pops out as an excellent tool, much better than other passwords estimators commonly used in web pages. In [13], the author compares zxcvbn to other popular java meters and arrives to the conclusion that only zxcvbn is reliable enough to actually give an useful feedback. In [14], the author makes an evaluation of several password generators and strength estimators. PwGen and zxcvbn, the two libraries used in this work, always give excellent results.

"For over 30 years, password requirements and feedback have largely remained a product of LUDS: counts of Lower- and Uppercase letters, Digits and Symbols. LUDS remains ubiquitous despite being a conclusively burdensome and ineffective security practice. zxcvbn is an alternative password strength estimator that is small, fast, and crucially no harder than LUDS to adopt. Using leaked passwords, we compare its estimations to the best of four modern guessing attacks and show it to be accurate and conservative at low magnitudes, suitable for mitigating online attacks. We find 1.5 MB of compressed storage is sufficient to accurately estimate the best-known guessing attacks up to 105 guesses, or 104 and 103 guesses, respectively, given 245 kB and 29 kB. zxcvbn can be adopted with 4 lines of code and downloaded in seconds. It runs in milliseconds and works as-is on web, iOS and Android". [16]

"People of course choose patterns — dictionary words, spatial patterns like `qwerty`, `asdf` or `zxcvbn`, repeats like `aaaaaaa`, sequences like `abcdef` or `654321`, or some combination of the above. For passwords with uppercase letters, odds are it's the first letter that's uppercase. Numbers and symbols are often predictable as well: `133t` speak (3 for e, 0 for o, @ or 4 for a), years, dates, zip codes, and so on. As a result, simplistic strength estimation gives bad advice. Without checking for common patterns, the practice of encouraging numbers and symbols means encouraging passwords that might only be slightly harder for a computer to crack, and yet frustratingly harder for a human to remember. xkcd nailed it". (see figure 1.5). [8]

To put it in other words, the authors of the project argue that a password like **`correcthorsebatterystaple`** (a nonsense English phrase) is more strong than a password like **`Tr0ub4dour&3`**, even if the former does not have any upper cases or numbers, and the latter seems more complicated.

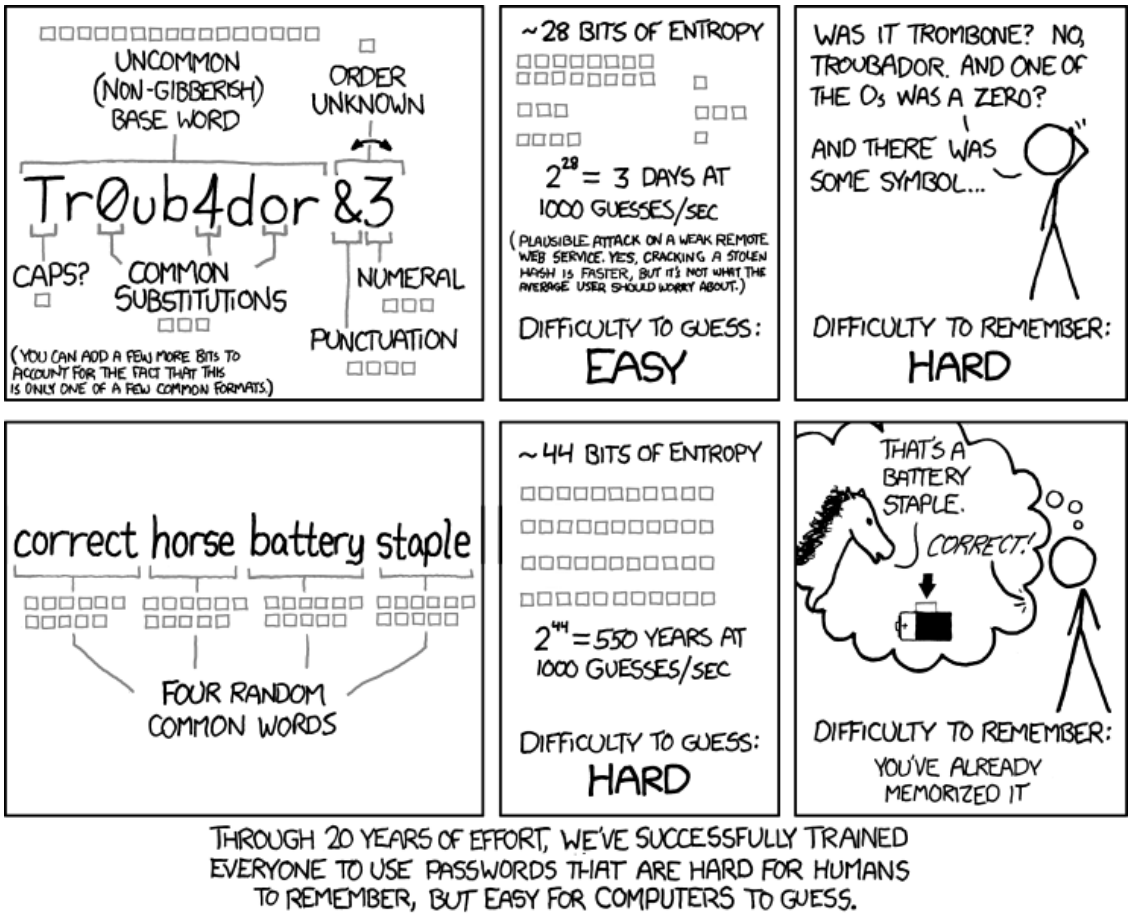


Figure 1.5: Password strength, xkcd [17]

The table in figure 1.6 (taken from [8]), show how zxcvbn is different from strength meters used in popular web services. (disclaimer: The data in the table is from 2012). From it we can learn:

1. Passwords like **`qwER43@!`**, which is a spatial password: it uses the keys `qwer4321`, with shift pressed for the keys `er` and `21` (the `@` symbol in the English keyboard is `shift+2`), is not considered week by most of the meters, but it should. It is probably due to the fact that it includes a combination of numbers and symbols that makes it look strong, but in reality, because of keyboard spatiality, is not.
2. Passwords like **`Tr0ub4dour&3`**, which is generated by replacing some of Troubadour letters with numbers, and adding two more characters, is regarded as a very strong password for all of the meters except zxcvbn. Even if the base word is uncommon, and it has some variations, it is not

	qwER43@!	Tr0ub4dour&3	correcthorsebatterystaple
zxcvbn	Weak	So-so	Great!
Dropbox (old)	Great!	Great!	So-so
Citibank	Medium	Strong	1 number required
Bank of America	(not allowed)	(not allowed)	(not allowed)
Twitter	✓ Password is perfect!	✓ Password is perfect!	✓ Password is perfect!
PayPal	Weak	Strong	Weak
eBay	Strong	Strong	(not allowed)
Facebook	•••••••• Password strength: Strong	•••••••• Password strength: Strong	•••••••••••••••••••••••• Password strength: Weak
Yahoo!	Very strong	Very strong	Weak
Gmail	Strong	Strong	Good

Figure 1.6: comparison between zxcvbn and popular websites' strength meters

long enough to be considered so strong.

3. A password like **correcthorsebatterystaple** is not considered strong by most of the meters except zxcvbn, and it is not even allowed in some cases because it lacks numbers, Upper-cases or symbols.

The superiority of zxcvbn over the other meters in the table may seem like cherry picking, but the way zxcvbn is constructed explains these differences.

Matching

Enumerates all the (possibly overlapping) patterns it can detect. Currently zxcvbn matches against:

- **Dictionaries:** Common words the user is likely to use as password. Multiple dictionaries, in a simple .txt format can be used. In this work, we present a few: English words, Italian words, names and surnames, Burnett's 10,000 common passwords, words from tv and films. The match has an associated frequency rank, where words like the and good have low rank, and words like photojournalist and maelstrom have high rank. This lets zxcvbn scale the calculation to an appropriate dictionary size on the fly, because if a password contains only common words, a cracker can succeed with a smaller dictionary. For all dictionaries, match recognizes uppercasing and common `l33t` substitutions.
- **Spatial keyboard patterns:** Some users are likely to choose passwords based on spatial pattern. For instance a user could choose the first row of letters from right to left: `poiuytrewq` as they password. QWERTY keyboard, Dvorak keyboard, and keypad are considered.
- **Repeats**: Users are also prone to use repetition of characters, like `rrrr`.
- **sequences**: Numeric or alphabetic sequences like `123` or `fedcba`
- **years and dates**: The year or full date of a special event, like anniversary or birthday. Years from 1900 to 2019 are considered and dates in different formats. (3-13-1997, 13.3.1997, 1331997).

Entropy calculation of a single pattern

Depending on the type of matching, the entropy calculation is done differently, but for all the cases the idea is the same: How many different cases a hacker would have to try before guessing the pattern? For example, for

the repeat case, if the user chooses `zzzzz`, as it is repeated five times, and if we assume the hacker starts by the letter a, then the number of cases would be $N = 26 \times 5 = 130$. (The sequence the hacker would try is: `a, b, c, d...,z, aa, bb, cc,...,...,zzzz, aaaaa, bbbbb,....zzzzz`).

As the number of possible cases can be pretty large, the entropy is not given as a raw value but as $e = log_2(N)$, known as the entropy bits, and in some cases as $f = log_{10}(N)$, known as the log entropy. In the example, entropy bits: $e = log_2(130) = 7bits$.

The entropy bits and log entropy are related by:

$$N = 2^e = 10^f$$

$$f = e \times log_{10}(2)$$

$$e = f \times log_2(10)$$

Minimum entropy search of whole password

Given the full set of possibly overlapping matches, the algorithm finds the simplest (lowest entropy) non-overlapping sequence. For example, if the password is damnation, that could be analysed as two words, dam and nation, or as one. It's important that it be analysed as one, because an attacker trying dictionary words will crack it as one word long before two.

zxcvbn calculates a password's entropy to be the sum of its constituent patterns. Any gaps between matched patterns are treated as brute-force "patterns" that also contribute to the total entropy. That a password's entropy is the sum of its parts is a big assumption. However, it's a conservative assumption. By disregarding the "configuration entropy" — the entropy from the number and arrangement of the pieces — zxcvbn is purposely underestimating, by giving a password's structure away for free: It assumes attackers already know the structure (for example, surname-bruteforce-keypad), and from there, it calculates how many guesses they'd need to iterate through.[8]

From entropy bits to rank and estimated crack time

To estimate the cracking time, it is necessary to make some assumptions about what kind of attack will be subjected the user. zxcvbn considers four possible scenarios according to the number of attempts/time the hacker can do:

1. **Online throttling (100 per hour):** Online attack on a service that ratelimits password authentication attempts.

2. **Online no throttling (10 per second):** Online attack on a service that does not ratelimit or where an attacker has outsmarted ratelimiting.
3. **Offline slow hashing (1e4 per second):** Offline attack. assumes multiple attackers, proper user-unique salting, and a slow hash function with moderate work factor, such as bcrypt, scrypt, PBKDF2.
4. **Offline fast hashing (1e10 per second):** Offline attack with user unique salting but a fast hash function like SHA1, SHA256 or MD5. A wide range of reasonable numbers anywhere from one billion to one trillion guesses per second, depending on number of cores and machines. Ballparking at 10B per sec.

zxcvbn then ranks a password with a security level from 0 to 4 according to its entropy value:

- **Level0 if ($N < 10^3$):** Too guessable, risky password
- **Level1 if ($N < 10^6$):** Very guessable, protection from throttled online attacks.
- **Level2 if ($N < 10^8$):** Somewhat guessable, protection from unthrottled online attacks.
- **Level3 if ($N < 10^{10}$):** Safely unguessable, moderate protection from offline slow-hash scenario.
- **Level4 if ($N > 10^{10}$):** Very unguessable: strong protection from offline slow-hash scenario

Where N is the number of possibilities a hacker would have to try for crack the password. So for instance, if the password is level 2, it could be cracked in around 10^8 guesses.

For Level0 the above rule in terms of the entropy bits is $e < log_2(10^3)$. In terms of the log entropy bits, it simply is $f < 3$.

The level and estimated crack time for each type of attack is presented to the user. With this information, the user will, hopefully, choose a Level4 password. Additionally, the user also receives feedback about how the password was cracked, so they know how to improve it.

1.3.6 PassPhrase Generator

From the previous two sections there seems to be a disagreement on what a good password looks like. PwGen can generate totally random passwords

or pseudo-random pronounceable passwords, but even the later go against what zxcvbn proposes: PassPhrases that are very easy to remember, but long enough to give excellent entropy results. To fill this gap, a PassPhrase generator that gives results along the lines of `CorrectHorseBatteryStaple` is used.

The PassPhrase generator developed by the author works by randomly picking out words from dictionary files. The user can tune the PassPhrase generation as follows:

- **Dictionaries:** The user must select appropriate dictionaries, containing a sufficiently large number of lines (larger than 10000) to ensure the picked words are really random. The English and Italian dictionaries used by zxcvbn are a good example. The user can work with as many dictionaries as desired, an the format must be one word per line. Only the first word of each line is counted, as everything after a space is trimmed.

- **Number of words:** The user can configure the number of words the generated PassPhrases are composed of. The recommended size is four, but it can be as long as the user wants.

- **Minimum Length of Words:** With this option is possible to select only random words whose length is higher than a certain value. This is to make sure the resulting PassPhrase is too short and therefore too insecure. The drawback here is that the higher the selected threshold, the fewer the available words in the dictionaries.

- **Only use infrequent words:** If the dictionaries follow the same format as those used for zxcvbn, that is, the words are ordered by frequency, having the most uncommon words in the lower part of the dictionary, the user can then ask to generate PassPhrases containing only unusual words. The drawback here is, again, fewer words to choose from. The percentage of words that are used is configurable.

- **Capitalize first letter:** To make the PassPhrases more readable, the first letter of each word can be capitalized.

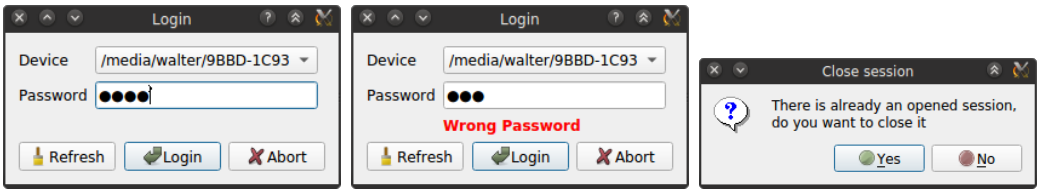
1.3.7 Device side development: Eclipse

1.4 Implementation

In the following sections each of the elements and functionalities of the application will be explained, how they were implemented, some interesting pieces of code and examples of use.

1.4.1 User authentication

When the user starts the application, the first window to appear is the Login Dialogue, shown in figure 1.7a. In it the user is asked to enter the login pin and by clicking accept the Challenge-Based Authentication process between the SEcubeWallet application and the SEcube chip starts. If the authentication fails because the entered pin is wrong, the message in 1.7b is shown. If it fails because there was already an opened session, the confirmation dialogue shown in 1.7c appears. If the authentication is completed successfully the user is granted access to the main window.



(a) Login Dialogue (b) Wrong Password (c) Already an open session

Figure 1.7: Login Dialogue and possible outcomes

The authentication process begins with the discovery of SEcube devices connected to the PC. This is achieved using Level0 APIs as seen in the listing 1.1. Each discovered device is added to the `QComboBox` displayed in the login dialogue and to a `QList`.

The user then selects one of the discovered devices using the `QComboBox`, enters their login pin and clicks accept. This triggers the listing 1.2. The first step is to open the device communication using the Level0 function `L0_open`. Then `L1_login` starts the actual challenged based authentication using the login pin entered by the user. Using this type of authentication ensures the login pin is never communicated between the devices and stolen with a physical attack on the USB cable. Rather, a random number is generated in the host and transmitted to the device. The login pin is then used in

Listing 1.1: Connected Devices discovery

```
//// *** variables declaration ***
se3_disco_it it;
QList<se3_disco_it> device_found;
QComboBox* chooseDevice;
bool found = true;

//*** Refresh button slot *****
L0_discover_init(&it); //initialize iterator
while((found = L0_discover_next(&it))){ //move to next device
  chooseDevice->addItem(QString::fromLocal8Bit(
    it.device_info.path, -1)); //add to GUI
  device_found.push_back(it); //add to QList
}
```

both the host and the device to encrypt this random number using a `pbkdf2` function. The resulting key on the device is sent to the host, who compares it with its own key. If they are the same it means the login pin entered by the user is equal to the login pin stored in the device, and the only information transmitted are random numbers that an attacker cannot understand. (The actual authentication procedure implemented is a little bit more complex, but is based on the same idea described here).

After both device and host have the same key, it is used to encrypt the communication channel. A token is generated in the device and transmitted on the encrypted channel to the host. This token is a random number, and is used from that point on to validate any communication between host and device. The SEcubeChip does not accept any command from the host if the token it sends is not the correct one. The only command accepted without a token is off course, login. In the logout procedure, this token is cleared (set to zeros), so a login later on is possible.

One problem found during the development of the application is the follwin: If after login in, the SEcubeWallet application crashes, the logout command, that is usually issued when closing, is never executed, and the device remains with an active token value. Therefore, when the user launches the application again, the login will fail, because the device expects a token value from the host. To solve this issue a few options were considered:

- Make sure the application never crashes. Because software applications are rarely completely bug-free, and even if they are, an external problem

Listing 1.2: Open device and try to login

```
// *** variables declaration ***
//use selected index at QComboBox to retrieve 'it' from QList.
int device_index = chooseDevice->currentIndex();
se3_disco_it it = device_found.at(device_index);

se3_session s;
se3_device dev;
int ret;
bool logout = false; //if true, L1_login logs out first

//*** Accept button slot *****
if(!dev.opened) // open communication with device
  if((L0_open(&dev, &(it.device_info), SE3_TIMEOUT))!= SE3_OK)
    exit(1); //error

ret = L1_login(&s, &dev, pin, SE3_ACCESS_USER, logout); //login

if (ret != SE3_OK){     //error at login
  if (ret == SE3_ERR_PIN) //The password is wrong
    show_wrong_pass_messsage;

  else if (ret == SE3_ERR_OPENED){
    // there is already an opened session, ask user if he wants
       to close it
    if(confirmation_dialog_reply == Yes){
      logout = true;
      call_this_function_again;
      //next time L1_login will close the existing session
    }
  }else
    exit(1);//other error
}else
  accept();//All ok, go to main window
```

like a bug in the OS can make them crash, this option is not feasible.

- Make sure the logout command is issued even in the case the application crashes. This option sounded promising, and a two-process idea was even developed. Process 1 is only in charge of calling Process 2, in which the actual application was executed. if Process 2 crashes, Process 1 remains alive and performs the logout procedure. To make this work, shared

memory was used to communicate the session variable (were the token is stored), so both processes could send commands to the device. The idea was latter on dropped because of two main reasons: First of all, it did not solve the case were the problem is external (OS bug), secondly, it was too complicated because of the sharing memory mechanism (The session variable is a fairly complex structure, with a lot of pointers), and because the token was being shared by two processes, this could open another possibility of attacks.

- As the two previous ideas failed, it was decided that a small modification to the login behaviour on the SEcube firmware was necessary. The modification consist on letting the login function clear the token field if necessary. This does not compromise the security of the system because access to the chip is only granted if the login pin entered by the user is the right one. One concern that may rise is that, while an application is using the SEcube, another one could close the session by issuing a Login command, but this is not possible because the `L0_open` function only allows one process to communicate with the chip at a time, using a file locker in the .se3magic file saved in the SEcube SDcard.

The new behaviour is implemented in listings 1.3. If after a crash the session in the SEcube remains open, and the host tries to login again, the SEcube returns the new error code `SE3_ERR_OPENED`. The host then can decide if it wants to force the SEcube to close the opened session so it can login, with the new command `SE3_CMD1_LOGOUT_FORCED`, which forces a logout without checking the token. After this command the host can login as usual. This steps are included in the host side `L1_login` function, which now has an additional parameter to control whether to force a logout or not. This parameter is the one used in listings 1.2, set to true when the user clicks YES in the confirmation dialogue asking whether or not to close the previous session.

1.4.2 Main Window

The SEcubeWallet GUI's main window developed using Qt is shown in figure 1.8.

The main window is composed of the following elements:

- **Table View:** Used for displaying the wallet entries. It resizes smoothly with the window, can be ordered by any of the columns, and the passwords are hidden by default but can be shown if the user wants to.

Listing 1.3: Modification in SEcubeFirmware, file se3_cmd1.c

```
1  if (se3c1.login.y) { // if there is already an opened session
2    if (memcmp(se3c1.login.token, req_params.token,
        SE3_L1_TOKEN_SIZE)) { //and token mismatch
3      if (req_params.cmd==SE3_CMD1_CHALLENGE)//someone (maybe same
         user after a crash) trying to login.
4        return SE3_ERR_OPENED;//notify host there is already an
           opened session, if host wants to continue, it will call
           SE3_CMD1_LOGOUT_FORCED
5      else if (req_params.cmd==SE3_CMD1_LOGOUT_FORCED)//if the
         user agreed to close the existing session by forcing a
         logout
6        req_params.cmd=SE3_CMD1_LOGOUT;//call logout as usual
7      else
8        return SE3_ERR_ACCESS;
9    }
10 }
```

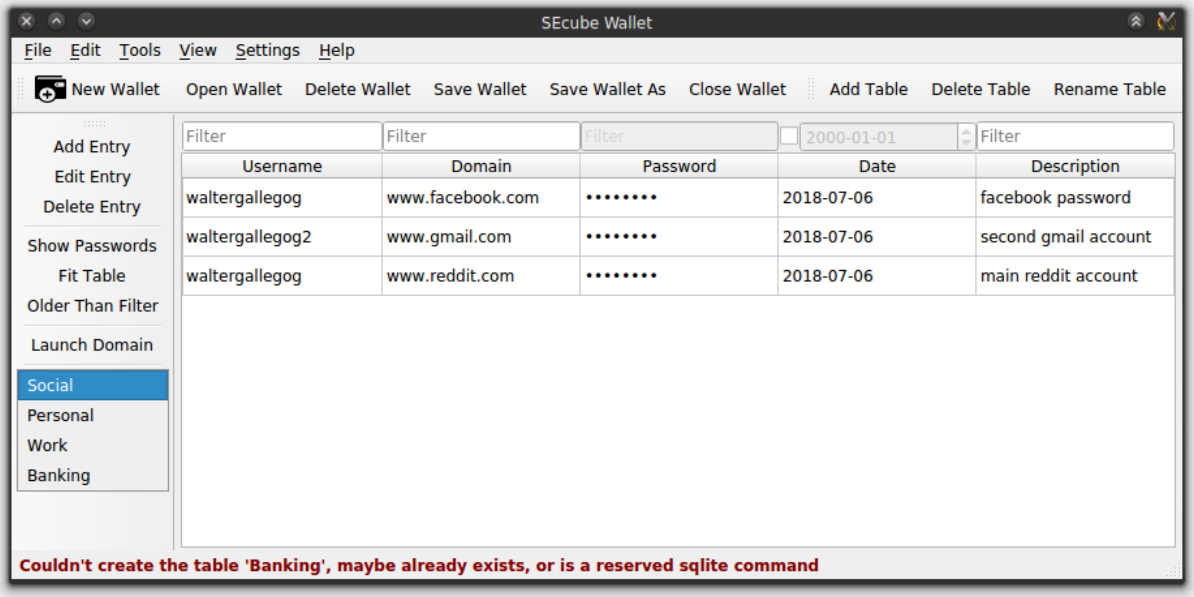


Figure 1.8: SEcubeWallet main window

- **Filters:** The user can search in each of the table's columns using filters. These filters are implemented inside a separate container, but they resize together with the table.
- **Entries Tool Bar:** It is positioned to the left of the table. It has the actions: add/edit/delete entries, show passwords, fit table, change date filter, launch domain and select table.
- **Tables Tool Bar:** It is positioned to the top right of the table. It has the actions: add/rename/delete table.
- **Wallets Tool Bar:** It is positioned to the top left of the table. It has the actions: new/open/delete/save/save as/close Wallet. All of the above Tool Bars are movable.
- **Menu Bar:** It is positioned at the top of the window. It contains all the previous actions, plus preferences and help.
- **Status Bar:** Positioned at the bottom of the window, it is used to display some success/error messages to the user and the current wallet name.

1.4.3 Preferences Subwindow

1.4.4 Help Subwindow

1.4.5 New Wallet action

When the user triggers the the `New_Wallet` action, the first step to execute is to check if there is another wallet opened and if it has unsaved changes. If, so the confirmation dialogue in figure 1.9 is shown, so the user can decide whether to save the changes, discard them, or cancel the creation of a new wallet.

In case the user clicks `Save`, the `Save_Wallet` action is triggered before continuing. `Discard` continues without saving, and `Cancel` returns without doing anything.

If the process continues, the next step is to close any previous in-memory database handlers, save the `table_view` geometry (if any), and open a new in-memory database using the Qt class `QSqlDatabase`, as seen in listings 1.4.

As explained before the in-memory data base is used for editing. It has the advantage of being fast because there is no access to the hard disk, and

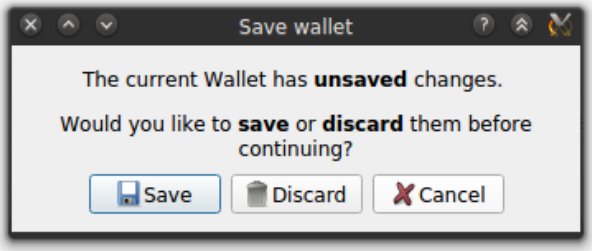


Figure 1.9: Save Confirmation dialogue

Listing 1.4: New in memory database

```
1  QSqlDatabase dbMem; //The database handler, declared in header
2
3  //Check if sqlite is installed on OS
4  if(!(QSqlDatabase::isDriverAvailable("QSQLITE")))
5    exit (1); //the application does not work without Sqlite
6
7  if (dbMem.isOpen()){
8    save_table_geometry;
9    dbMem.close();//close any prev. opened database
10 }
11
12 dbMem = QSqlDatabase::addDatabase("QSQLITE");
13 dbMem.setDatabaseName(":memory:"); // in-memory database
14 if(!dbMem.open()){
15   return;//Error opening, do nothing
16 }
```

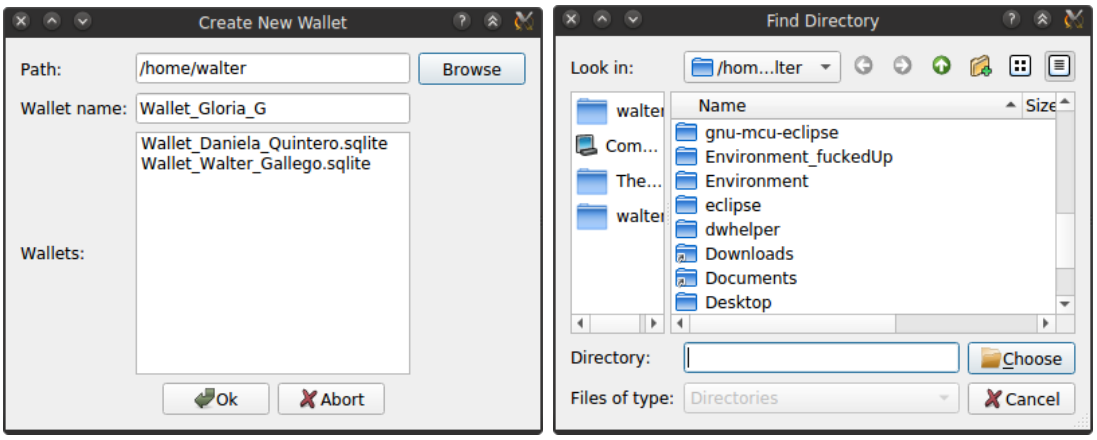
secure, because all the data is in the application memory space, and therefore is protected by the OS.

The last step is to update the GUI state, by enabling some action like `Add_Table` and `Save_Wallet`, and disabling others, like `Delete_Table` and `Rename_Table`.

1.4.6 Save Wallet action

To write the wallet contents to the disk, it is necessary to have a filename, so the first step is to check if the user already entered one (from previous saves).

If not, with the dialogues in figure 1.10 the user can choose the directory and the filename to save.



(a) User can enter a New name. Current wallets are displayed

(b) if `Browse` is clicked a `QFileDialog` is launched

Figure 1.10: Save Wallet dialogues

The need for two dialogues instead of a regular file browser comes from the fact that the chosen filename will not be readable from the OS, since `SEfile` also encrypts it. Similarly, wallets already saved in the directory cannot be displayed with a regular file browser, so it is necessary to use the `SEfile` function `secure_ls` and display its output in the list seen in figure 1.10a. The declaration of this function is in listings 1.5. To chose the working directory, it is enough to use the `QFileDialog` class. If the user wishes they chose an existing filename and can overwrite the correspondent wallet.

Listing 1.5: `secure_ls` declaration

```
1  /* This function identifies which encrypted files and encrypted
       directories are present in the directory pointed by path
     and writes them in list. It only recognizes the ones
     encrypted with the current environmental parameters.*/
2
3  uint16_t secure_ls(     //returns 0 in case of success
4    char *path,           //[in]Path to the directory to browse
5    char *list,           //[out]Allocated array to store filenames
6    uint32_t *list_length//[out]Num of char written in list
7  );
```

After having a filename the next step is to read all of the tables in the current in-memory database, row by row. For each row a `sqlite` statement of the form `INSERT INTO table VALUES(user, dom, pass, date, desc)` is created. All of them are merge into a single statement which is executed into the secured in-disk database. This ensures only one access to the SEcube and disk. This process is somewhat slow and the GUI is disabled while it is performed. A simplified version of the code is shown in listing 1.6

One problem found during the open process of a secured in-disk wallet (that will be explained latter on) is that the first table is always corrupt and gives the error: `database disk image is malformed`. The error only occurs when using the SEcube version of the sqlite library. Because it was impossible to find the origin of the error, it was decided to use a workaround: In the save wallet process, an empty table is inserted at the beginning of the in disk database (see line 16 in listings 1.6). When opening the wallets, the empty table is simply ignored. With this, the real tables are always correctly read an the application works as intended.

1.4.7 Save Wallet As action

This action is very simple, it just clears the current filename (if any), and calls the `Save_Wallet` action; as there is no filename, the user is forced to enter a new one. The only point to be careful about is that, in case the `Save_Wallet_As` process is aborted, the previous filename needs to be recovered, so before clearing, the filename is temporary stored in case it is needed.

1.4.8 Open Wallet action

Similarly to the `New_Wallet` action, the first step is to check for unsaved changes and ask the user if save them, discard them or cancel, with the dialogue in figure 1.9

If the user decides to continue, the dialogues in figure 1.11 allow them to choose from the list the wallet to open.

The application proceeds doing the inverse process to the `Save_action`, that is, read all the tables from the secure in-disk database and create an in-memory database with this data. To do so, the listings in 1.7 is used. First, the in-disk data base is opened as read only, and a list named `tables` with the existent tables in it is generated. Then an in-memory database is created.

Listing 1.6: simplified Save process

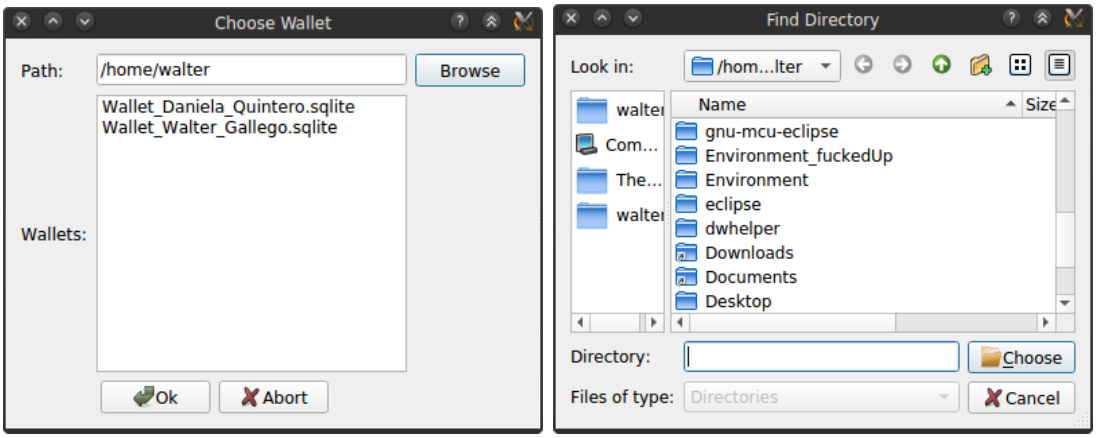
```
sqlite3 *dbSec;     //Secure database declaration, in header
QSqlDatabase dbMem;//The database handler, declared in header
QSqlQuery query;    //To exec sqlite statements, dec. in header

//Create Sqlite DB, with filename specified by user If SEcube is
    connected, the resulting file is encry.
sqlite3_open_v2 (fileName.toUtf8(),
                 &dbSec,
                 SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE ,
                 NULL)

QString finalSql; //To Merge all the sqlite statements.
static const QString insert =
  QStringLiteral("INSERT INTO '%1' VALUES(%2);"); //statement

dbMem.tables(QSql::Tables); //list of the tables in in-memory DB
tables.prepend("NoEmpty");  //Workaround: prepend Empty table
foreach (const QString table, tables) { // loop all the tables
   QString sql = "create table "+table+ //Create table statement
                 "(id integer primary key, "
                 "Username TEXT, "
                 "Domain TEXT, "
                 "Password TEXT, "
                 "Date TEXT, "
                 "Description TEXT );" ;
  sqlite3_exec(dbSec, sql.toUtf8(), NULL, 0, &zErrMsg);// exec

  if (table=="NoEmpty"){//just an empty table
    set_values_to_empty;
    finalSql += insert.arg(table).arg(values.join(", "));
  }else{
    query.prepare(QString("SELECT * FROM [%1]").arg(table));
    query.exec()
    while (query.next()){ //row by row
      values = query_read_row();
      finalSql += insert.arg(table).arg(values.join(", "));
    }
}
//single write into secure database, fill the tables
sqlite3_exec(dbSec, finalSql.toUtf8(), NULL, 0, &zErrMsg);
```



(a) User chooses the wallet to open from the list

(b) if `Browse` is clicked a `QFileDialog` is launched

Figure 1.11: Open Wallet dialogues

Finally, for each table in `tables`, its contents are read, a correspondent table is created in the in-memory database, and the later is populated with the read contents from the in-disk DB.

The `SELECT FROM` statements in `sqlite3` require the use of a *callback* function, which is called for every result row, and receive the actual data from the data base as an `argv[]` argument. In the open process two of these functions are needed, shown in listing 1.8. In `create_TableList`, the list `tables` is build by simply reading the only element in the `argv[]` array, as each row only consists of a table's name. In `populatetable`, the table in the in-memory DB is populated, row by row with each call. In this case the `argv[]` array holds the values in a single row coming from the in-disk DB.

Finally the GUI is updated, by enabling some elements and disabling others.

1.4.9 PwGen: Pronounceable Password Generator

As seen from previous sections, the PwGen program is open source and available in the official Linux repositories. A very simple way of including its functionalities into the SEcubeWallet application would be to use a `Qprocess` to call PwGen as an external program. Although is tempting to use this solution because of its simplicity, there are three drawbacks with this approach:

Listing 1.7: Simplified Open Wallet action

```
sqlite3_open_v2(fileName.toUtf8(),
                &dbSec,
                SQLITE_OPEN_READONLY, NULL)); //open in-disk DB

QString tableNames="SELECT name FROM sqlite_master "
                   "WHERE type='table' "
                   "ORDER BY name;";
sqlite3_exec(
  dbSec, tableNames.toUtf8(),
  callback_createTableList,//builds the 'tables' list
  this, &zErrMsg);

if (dbMem.isOpen()){
  save_table_geometry;
  dbMem.close();//close any prev. opened database
}
dbMem = QSqlDatabase::addDatabase("QSQLITE");
dbMem.setDatabaseName(":memory:");
dbMem.open();
query = QSqlQuery(dbMem);

foreach (const QString table, tables){
  QString sql = "create table "+table+
                "(id integer primary key, "
                "Username TEXT, "
                "Domain TEXT, "
                "Password TEXT, "
                "Date TEXT, "
                "Description TEXT );";
  if (table!="NoEmpty"){ //Workaround: ignore empty
    query.prepare(sql);
    query.exec(); //create table in in-mem DB
  }
  QString SqlStatement =
    QStringLiteral("SELECT * FROM '%1';").arg(table);
  sqlite3_exec(
    dbSec, SqlStatement.toUtf8(),
    callback_populateTable, //populates in-mem DB
    this, &zErrMsg);
}
```

Listing 1.8: Callback functions for Sqlite3 SELECT

```
//Build TableList from in-disk DB
callback_createTableList(int argc, char**argv, char**azColName){
  tables << argv[0]; //only one arg, the table name
  return 0;
}
//fill 'table' in in-mem DB with data from in-disk DB
callback_populateTable(int argc, char **argv, char **azColName){
  if (table=="NoEmpty") // we dont want NoEmpty in the in-mem db
    return 0;
  int i;

  static const QString insert =
    QStringLiteral("INSERT INTO %1 VALUES (%2);");

  QStringList values;
  QString aux;
  for(i = 0; i<argc; i++){//argv holds values from a single row
      aux = argv[i];
      values << "'"+aux+"'";
  }
  query.prepare(insert.arg(table).arg(values.join(", ")) );
  query.exec();
  return 0;
}
```

1. It would require for the user to install the PwGen program, as it is usually not included in common Linux distributions.

2. It would not be very portable, because even if there is a PwGen version for windows, the available version or input parameters could differ in different platforms.

3. Security could be compromised. As PwGen needs to communicate the generated password back to the SEcubeWallet application, an attacker could steal the password in this process.

For these reasons, it was decided to embed the PwGen sources directly into the application, this ensure the password never leaves the application memory space. To include the sources into the application, some slight modifications (mostly simplifications) to the PWGen `main()` function were necessary. This is because the original sources are intended for the use of PwGen

as a independent console program called by users, so the `main()` contains code dedicated to parse the input arguments in the standard `argc argv[]` fashion. Only the `pwgen.c` and `pwgen.h` — files where the `main()` is implemented — were modified.

Options GUI

As all the options for PwGen besides the password length are yes or no questions, the checkable list shown in figure 1.12 is perfect for this purpose.

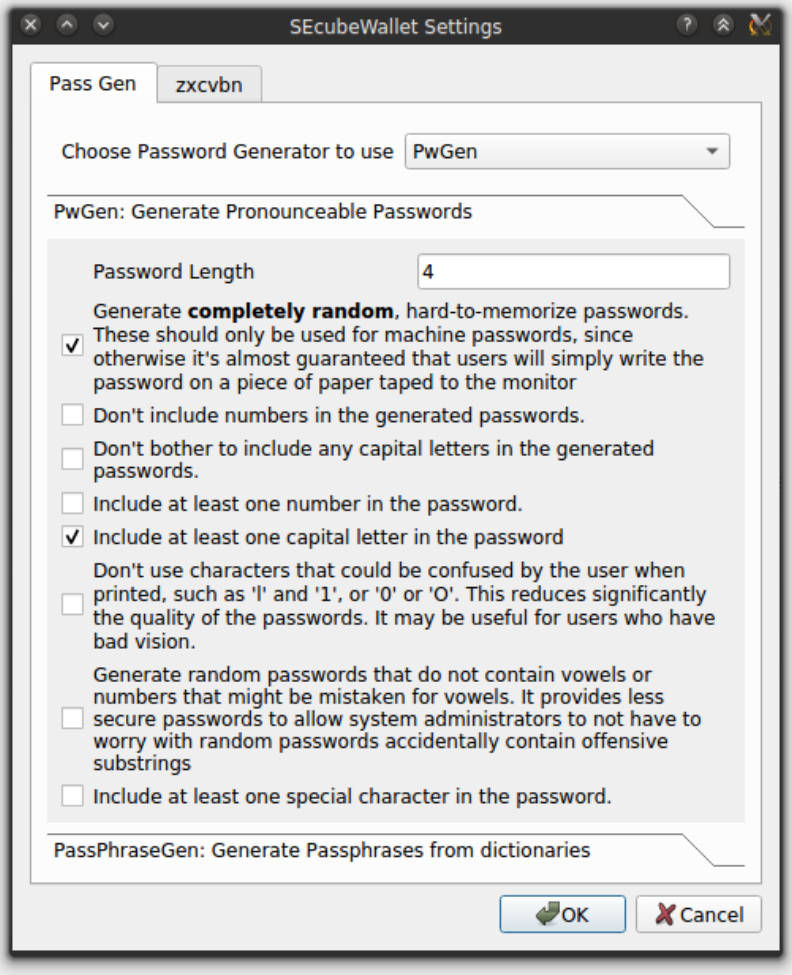


Figure 1.12: PwGen settings in the preference window

When the `OK` button is clicked, all the values are saved using the `QSettings`

class, so they are available even after restarting the application.

Usage

When the `Generate Password` button in the `AddEntry` class is clicked, the settings values stored in with the `QSettings` class are read and from them a `char[]` with the PwGen options syntax is built. Before calling the Generator, it is necessary to allocate a memory space equal to the desired password length (in `char`) and pass this buffer as a pointer (`char*`) to PwGen. PwGen will write the generated password in this space. This steps can be seen in the listing 1.9

Results

A few examples of the resulting Passwords are show in table 1.1, with their respective zxcvbn Log entropy score and Level. As expected, the passwords with highest entropy are those obtained using the `-s` (Random) and `-sy` (Random and especial characters) options. Even with only 6 characters it is possible to get a Level 4 password like `TBw4)9`. The drawback is, they are hard to remember and type.

On the contrary, a password obtained with the `-BA0` (No ambiguous, do not capitalize, no numbers, pronounceable) options, like `nofosootei`, only reaches a Level2, but it is very easy to remember and type.

The large number of options PwGen offers, together with the PassPhrase generator and the l33t converter, give the users a lot of types of passwords, each suited for different situations, given SEcubeWallet great flexibility.

1.4.10 zxcvbn Password strength estimator

The original zxcvbn project, developed in CoffeeScript became so popular it was ported to a large variety of languages. In this work the C/C++ version available at [7] was used. The files used in this project are:

- **zxcvbn.c** Main source file
- **zxcvbn.h** Main header file
- **dict-generate.cpp** Used for generating the dictionary sources
- **Makefile** To compile the dictionary generator and main program.
- **words-*.txt** A few examples of dictionary files in plain text format.

Listing 1.9: PwGen call inside AddEntry

```
//read user settings (if existent)
if (settings.value("passGens/pwgen/1cap").toBool())
    options.append("c");
if (settings.value("passGens/pwgen/1num").toBool())
    options.append("n");
if (settings.value("passGens/pwgen/1spec").toBool())
    options.append("y");
if (settings.value("passGens/pwgen/noAmb").toBool())
    options.append("B");
if (settings.value("passGens/pwgen/noCap").toBool())
    options.append("A");
if (settings.value("passGens/pwgen/noNum").toBool())
    options.append("0");
if (settings.value("passGens/pwgen/noVow").toBool())
    options.append("v");
if (settings.value("passGens/pwgen/random").toBool())
    options.append("s");

//check if user entered an integer, if not, default is 16
if(settings.value("passGens/pwgen/len").toInt())
    length = settings.value("passGens/pwgen/len").toInt();

//allocate space for password
buf = (char*)malloc(length+1);
if(!buf){
  return;//error, could not allocate
}
//actual call to password generator
main_pwgen(
  options.length(),               //int, number of options
  options.toLatin1().constData(), //char *, options
  length,                         //password length
  buf                             //char *, to return the
      password
);
genPass = QString::fromLatin1(buf,length);
free(buf);
```

Besides source files, zxcvbn also needs to compile the dictionary files, but first lets define what is a dictionary, why are they important and why they need to be compiled (For general dictionaries only. User dictionaries are

Table 1.1: A few PwGen generated passwords

Password	Length	Options	Log Entropy & Level
iesohGhai3	10	-	9.75 (Level 3)
ees0cooLo2	10	-	10.47 (Level 4)
dX042wKqlW	10	s	17.86 (Level 4)
@!,Q*l5}+H	10	ys	18.15 (Level 4)
TBw4)9	6	ys	11.62 (Level 4)
B7t34Lck	8	v	11.87 (Level 4)
nofosootei	10	BA0	6.50 (Level 2)

small and can be added at runtime).

General dictionaries

Dictionaries are a crucial part of the algorithm, because they are used estimate the security level of a password according to how common the used words (if any) are. A password containing words present in any of the dictionaries will be easier to crack as hackers will probably try out those specific words or a combination of them.

General dictionaries contain a large number of words that are useful for all users. Examples of these type of dictionaries (included in this work) are:

- 100000 English words from wikipedia.
- 88800 Last names from the US census database
- 39000 English words from tv and film from the wikiproject [6]
- 47000 Most common passwords from Burnett [12]
- 15480 Italian words from the BADIP project [2]
- 4276 Female names from the US census database
- 1220 Male names from the US census database

As the dictionary files in plain text are pretty large, the algorithm does not read from them directly. Instead, a `DicNodes` array is generated, using

the tool `dict-generate`, and this array is compiled into the source code. To add their own dictionaries, the users need to make sure they are saved as plain-text, (.txt UTF-8), and stored into the zxcvbn directory. The files must have one word per line, with the first word being the most common one. So for instance, in the English dictionary the first word is `the` and the last one is `surma`. This is important as it is used to calculate the entropy of the passwords. A password containing the word `surma` is far more secure that one containing the word `the`.

Static Library vs Shared Library

Because the dictionaries are transformed into a source file and then compiled together with the main program, it is not possible to add, remove or modify dictionary files after the sources are compiled. Therefore the zxcvbn library can not be embedded into the SEcubeWallet application as a static library (or using the C sources directly), but rather, a **shared library** approach was followed, which allows the dynamic unload/update/load of the library. This has some performance penalties over static libraries, but it is the only way to give the users the possibility of customize the dictionaries as they please.

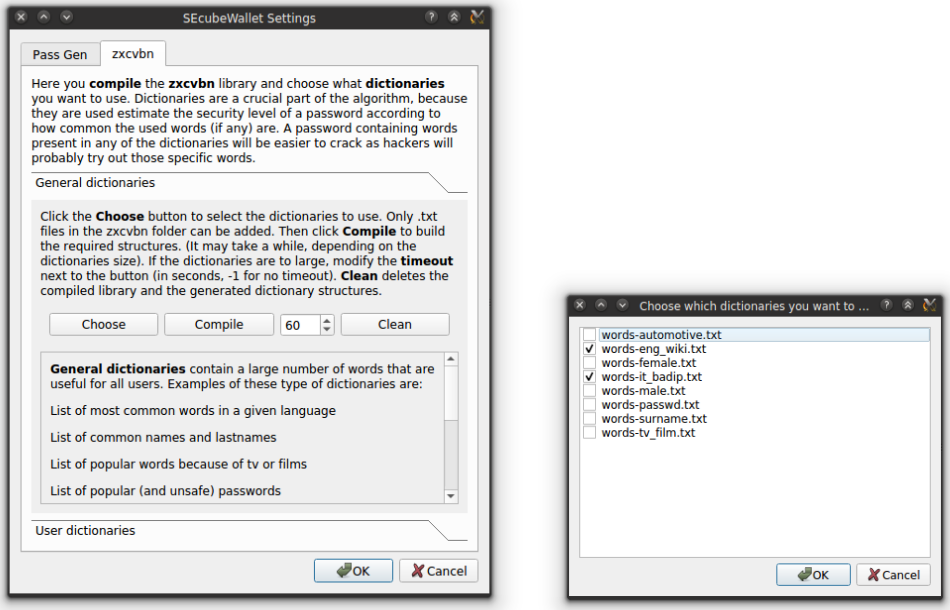
Compilation process

The steps performed by the `makefile` in order to compile the dictionaries and sources are:

1. Compile the source file `dict-generate.cpp` to obtain the dict generator executable`dictgen`.
2. Execute `dictgen` with the names of dictionaries to process as input argument. As a result the file `dict-src.h` is created.
3. Compile the files `zxcvbn.c` and `zxcvbn.h` together with the just generated `dict-src.h`, using the gcc flag `-fPIC` so the resulting object file `zxcvbn-inline-pic.o` is suitable for library inclusion.
4. Generate the shared library `libzxcvbn.so` from the object file. This is the library used by the SEcubeWallet sources.

The compilation process can be started by the user from the preference window, where they can also select the dictionaries to use, or clean the generated files. The compiling is made with OS calls, through the use of `QProcess`. To avoid the application from crashing or getting stuck, the `Qprocess` has

a timeout. As the compile process may take a while depending on the dictionaries size, this timeout can be configured by the user. In figure 1.13 the GUI for these actions is shown.



(a) Tab dedicated to zxcvbn preferences

(b) Checkable dialogue to enable dictionaries, appears after clicking the Choose button

Figure 1.13: zxcvbn general dictionaries configuration

Dynamic Library loading

To manage the zxcvbn shared library at runtime it is possible to use the qt class `Qlibrary`, which provides access to the functionality in the library in a platform independent way. To use it, is necessary to pass as argument to the constructor the path to the library. Then load it and resolve the desired functions. If no errors are found, the functions can be used as usual. Finally unload the library when it is not needed any more. See Listing 1.10.

Listing 1.10: Qlibrary basic usage

```
1
2  /***** In header file *****/
3  //Main zxcvbn function type
4  typedef double (*ZxcvbnMatch_type)(const char *Passwd,
5                                     const char *UserDict[],
6                                     ZxcMatch_t **Info );
7  //Function used to free the Info structure
8  typedef void (*ZxcvbnFreeInfo_type)(ZxcMatch_t *Info);
9
10 QLibrary * zxcvbnLib = 0;
11 ZxcvbnMatch_type ZxcvbnMatch = 0;
12 ZxcvbnFreeInfo_type ZxcvbnFreeInfo = 0;
13
14 /***** In cpp file *****/
15 zxcvbnLib = new QLibrary(zxcvbn_lib_path);
16
17 if(zxcvbnLib->load()){
18   ZxcvbnMatch = (ZxcvbnMatch_type) zxcvbnLib->resolve("
       ZxcvbnMatch");
19   ZxcvbnFreeInfo =(ZxcvbnFreeInfo_type) zxcvbnLib->resolve("
       ZxcvbnFreeInfo");
20 }
21
22 if (!ZxcvbnMatch || !ZxcvbnFreeInfo ){
23   //error: Any of the two functions was not resolved correctly
24 else{
25   //we can use the functions normally
26
27 //When not needed any more
28 zxcvbnLib->unload();
29 ZxcvbnMatch = 0;
30 ZxcvbnFreeInfo = 0;
31 free(zxcvbnLib);
```

User Dictionaries

The user dictionary contain words that are relevant only to a specific user. For example, if the application is used to increase the strength level of passwords used by employees in a company, adding the company's name to the dictionary is a good idea. Furthermore, if the company works in the automotive business, related words as motor, aerodynamic, wheels etc. should

be added. By adding those words to the user dictionary, the strength level of passwords using them will decrease, and so the user will be encouraged to never use words that are too easy to guess. The key to a good password is in its randomness. When a hacker is trying to crack one, they will for sure try words relevant to the target.

From the GUI the user can add words manually, or can load them from a text file, but as the words are saved as a simple array, the text file size should not be too large. For large files, it is better to add them as General dictionaries.

Estimator Usage

After the library is loaded and the functions resolved, to use the estimator one simply needs to call the main function `ZxcvbnMatch` whose declaration we see in Listing 1.11

Listing 1.11: ZxcvbnMatch function declaration

```
1 double ZxcvbnMatch(        //Returns: entropy value in bits.
2
3   const char *Passwd,      //The password to be tested. Null
       terminated string.
4
5   const char *UserDict[], //User supplied dictionary words to be
       considered particulary bad. Passed as a pointer to array
       of string pointers, with null last entry (like the argv
       parameter to main()). May be null or point to empty array
       when there are no user dictionary words.
6
7   ZxcMatch_t **Info        //The address of a pointer variable to
       receive information on the parts of the password. This
       parameter can be null if no information is wanted. The data
       should be freed by calling ZxcvbnFreeInfo().
8 );
```

To obtain the password strength level, it is necessary to compare the `zxcvbnMatch` return value (The entropy in bits) as seen in section 1.3.5. The strength level is shown to the user with a progress bar. To the user may be more relevant to see some estimates about how long it would take for an attacker to crack the password. This information can be obtained from the entropy, assuming some numbers for the attempts/time the attacker can

perform. These results are shown in a table like the one in figure 1.14

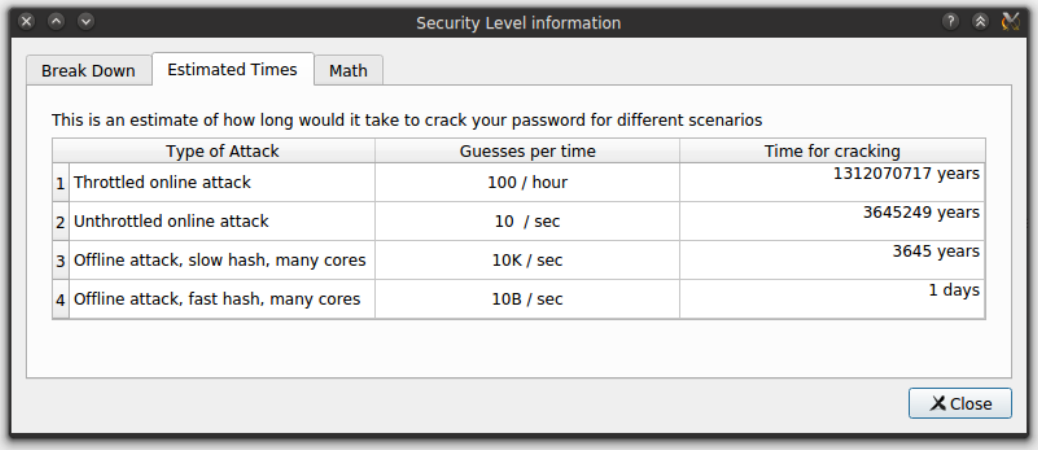


Figure 1.14: Crack times for different attacker capabilities

Some interesting additional information can be obtained from `ZxcMatch_t ** Info`. By traversing the data in this pointer, it is possible to see how the zxcvbn algorithm broke down the password. The user can see this information in a table like the one in figure 1.15

1.4.11 PassPhrase Generator

The `PassPhraseGen` C++ function implements the PassPhrase Generator. The function call is done from the `AddEntry` class, when the `Generate Password` button is clicked.

`AddEntry` reads the configuration values stored as `QSettings`, asserts them and then makes the call. This values can be modified by the user in the preferences window, shown in figure 1.16. After the user selects the dictionaries and tunes the available options, they must click the apply button, which will trigger a line by line read of the dictionaries, to count the number of lines, that is the number of words available. This is necessary as it is not possible to know how many lines a file has without counting them, and the total number is required in order to generate properly bounded random numbers in the PassPhraseGen function.

The function declaration is shown in listings 1.12

The function first generates `numWords` random numbers in the range `[1, totalLen]` with the Qt function `QRandomGenerator` (introduced in version

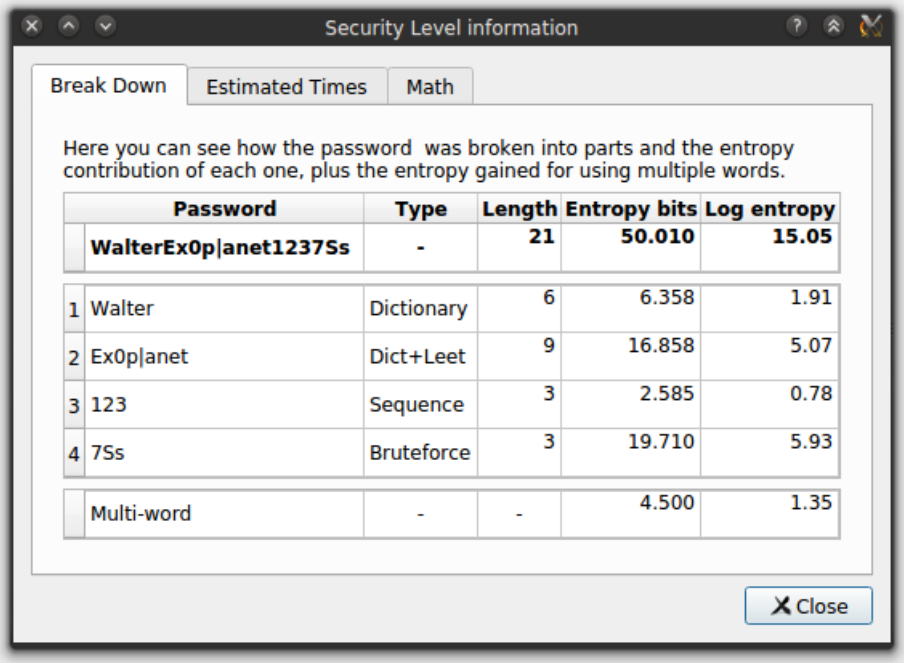


Figure 1.15: Password broke down by the zxcvbn algorithm

Listing 1.12: PassPhraseGen function declaration

```
QString PassPhraseGen(   //Return: Generated PassPhrase
  QString path,          //path to dicts
  QStringList dicts,     //list of dicts
  QStringList dictsLen, //cumulative list of dics lengths
  int totalLen,          //total number of candidate words
  int numWords,          //number of words in the password
  bool ppgenMinLenEnab, //use only words longer than min len
  int ppgenMinLen,       //min length
  bool capFirst,         //uppercase first letter of each word
  bool ppgenLowerEnab,  //use only lower part of dicts
  int ppgenLower         //how much of the lower part to use
);
```

5.10). The generated numbers represent *Line numbers* in the dictionary files. As each line contains a word, the function indeed extracts random words. There are a few things to consider in this process:

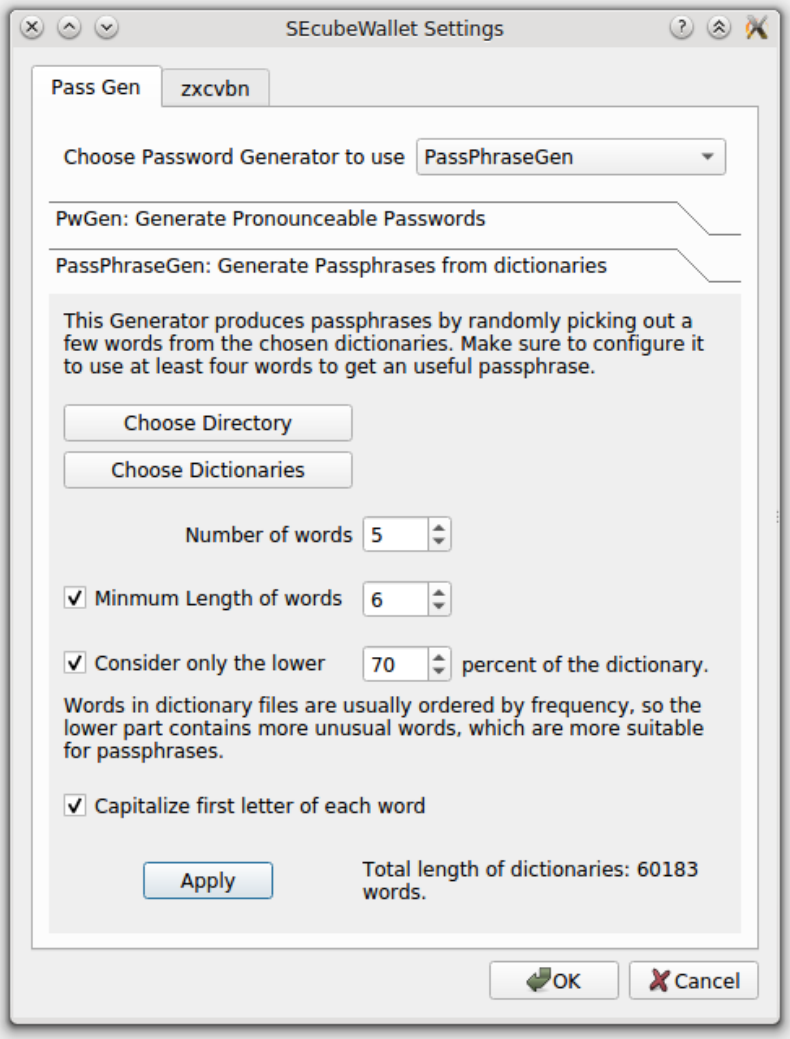


Figure 1.16: Settings for PassPhrase Generator

1. It is impossible to read a random line from a TextFile without reading all the previous lines first. So, in order to extract the words it is necessary to read the dictionary line by line and keep a counter to know the number line we are at. To speed up the process making sure each line is read only once, the random numbers are sorted in ascending order first. The dictionary can then be read line by line extracting the words where the counter equals one of the random numbers. When the last random line is extracted, the file can be closed.

2. There can be multiple dictionaries. The generated random numbers span the total of available words, so some random lines will be in some dictionaries, some in others. Therefore, using the `dictsLen` list, which contains the cumulative lengths of the dictionaries, we need to determine in which dictionary and in which internal line, each random number is. With this information, and the random numbers ordered, it is possible to extract the words efficiently and making sure dictionary files are opened only if necessary and only once.

3. When the minimum length option is enabled, the total number of available words is reduced. The algorithm accounts for this fact by counting only the lines with a word larger than the minimum length. This is done both in the preference window, where the total number of lines is counted, and at the word extraction process. In this way, the random lines are pinpointed as before, by reading line by line and comparing the counter; the working logic is not altered, it just ignores the "disabled" short lines.

4. Finally, to consider only the lower part of each dictionary the preferences window counting process is not altered, and the modifications are all done at the extraction process. If for example, the user wants to work with the lower 30%, the random generated numbers are now bounded to `[1, (0.3)totalLen]`. The corresponding dictionary and internal line for each random number are calculated by taking into account that the first 70% of each dictionary must be skipped. With this two values, the files can be read as in the simple case. As this process is different from the minimum length one, they do not interfere with each other.

Table 1.2 presents some PassPhrases examples for different configurations, along side the score given by the zxcvbn estimator. The estimator uses the same dictionaries as the generator, so this assumes a worst case scenario where the hacker has access to all the possible words the user considered when creating the PassPhrase.

Two dictionaries where used: `words-eng_wiki.txt` with 100000 lines and `words-it_badip.txt` with 15480 lines (around 6 times smaller), so most of the extracted words will be English.

The results in the table indicate the most important parameter is the number of words. Three words are enough to reach a zxcvbn level 4, which, as seen in previous sections, is very secure. A two word PassPhrase as long as `DrammaturgicoSbatacchiare` is not better than the shorter `ImmobileCwSites`,

Table 1.2: PassPhrases examples for different configurations

PassPhrase	No. of words	Min. word Len	% of dict. used	Log Entropy & Level
Cocchio	1	-	-	4.27 (L1)
Melun	1	-	-	4.93 (L1)
Legitimately	1	8	-	4.55 (L1)
Woodhaven	1	8	30%	4.94 (L1)
VestaOrman	2	-	-	7.78 (L2)
ShorelineCech	2	-	-	9.18 (L3)
MongoliaSimpsons	2	8	-	7.30 (L2)
McinnisPhaya	2	-	30%	9.14 (L3)
ZucchiniSalamandra	2	8	30%	9.19 (L3)
SacchettiVigevano	2	8	30%	9.11 (L3)
DrammaturgicoSbatacchiare	2	12	-	8.98 (L3)
MalformationsAstrophysical	2	12	-	9.60 (L3)
LatinaInterchangeFbo	3	-	-	13.5 (L4)
OsaAymanCantinflas	3	-	-	12.98 (L4)
ImmobileCwSites	3	-	-	11.43 (L4)
RimmelBragFaenza	3	-	30%	13.491 (L4)
RecliningCanberraEcuadorian	3	8	-	13.69 (L4)
SeashellsHippocraticCameroun	3	8	30%	14.90 (L4)
InaspettatoRothschildsDisconcerting	3	8	30%	14.48 (L4)

because the latter has one more word. The minimum word length also influences the PassPhrase entropy, but their effects are not as pronounced. Finally, working only with to the lower part of the dictionary may seem to

not have any effect, but in reality its use is crucial: it ensures the generated PassPhrases do not contain any of the most common words, like `the` or `essere`. From the results on the table we can not appreciate this fact, but at least we learn the option does not do any harm either. (Although if the attacker finds out the user is generating PassPhrases using only the 30% most uncommon words on a given language, they job would get easier).

Bibliography

[1] About Qt. https://wiki.qt.io/About_Qt.

[2] Banca dati dell'italiano parlato . http://badip.uni-graz.at/en/corpus-lip/list-of-lemmata.

[3] Blue5 Group. http://www.blu5group.com/.

[4] Qt Creator. http://doc.qt.io/qtcreator/.

[5] SEcube SDK. https://www.secube.eu/resources/.

[6] Wiktionary:Frequency lists. https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.

[7] zxcvbn: Low-budget password strength estimation, c/c++, on github. https://github.com/dropbox/zxcvbn.

[8] zxcvbn: Realistic password strength estimation. https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/.

[9] *SEcube Data Sheet introduction*, 2015.

[10] Airo, G., Prinetto, P., Carelli, A., Somma, G., and Varriale, A. *SEcube Development Kit: Getting Started*, 2017.

[11] Airo, G., Prinetto, P., Ferri, N., Carelli, A., Scalia, G., Somma, G., and Varriale, A. *SEcube Development Kit: L2 User Manual*. 2017.

[12] Burnett, M. 10,000 Top Passwords. https://xato.net/10-000-top-passwords-6d6380716fe0.

[13] Stockley, M. Why you STILL can't trust password strength meters. https://nakedsecurity.sophos.com/2016/08/17/why-you-still-cant-trust-password-strength-meters/.

[14] Toponce, A. A document evaluating different open source password generators and password strength testers. https://gist.github.com/atoponce/173c113ea4a81a9657148ce5d4fa2fd3.

[15] Ts'o, T. pwgen(1) - Linux man page. https://linux.die.net/man/1/pwgen.

[16] Wheeler, D. L. zxcvbn: Low-budget password strength estimation. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 157–173. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler.

[17] xkcd. Password strength. https://xkcd.com/936/?