SE*cube*™ Development Kit

L2 User Manual

Release: January the 16th, 2017


HW
SW
Open
Source

Proprietary Notice

The following document offers information, which is subject to the terms and conditions described hereafter.

While care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinion expressed are in good faith and while every care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinion expressed are in good faith and while every care has been taken in preparing this document.

Authors

Giuseppe AIRÒ FARULLA *(CINI Cyber Security National Lab)* giuseppe.airofarulla@polito.it
Alberto CARELLI *(CINI Cyber Security National Lab)* alberto.carelli@polito.it
Nicola FERRI
Paolo PRINETTO *(President, CINI)* paolo.prinetto@polito.it
Giulio SCALIA
Giorgia SOMMA *(Business Development Manager, Blu5 Labs Ltd)* *giorgia.somma@blu5labs.eu*
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Acknowledgment

Authors would like to thank the following persons for their valuable support:
Frederik GOSSEN
Pascal TROTTA

The present work has been partially supported by CISCO and developed within the Project "FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks".

Trademarks

Disclaimer

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "AS IS" BASIS AND ITS AUTHORS DISCLAIM ALL WARRANTIES, EXPRESS, OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY TAHT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PURPOSE.

THE SOFTWARE IS PROVIDED TO YOU "AS IS" AND WE MAKE NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER WITH RESPECT TO ITS FUNCTIONALITY, OPERABILITY, OR USE, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PURPOSE, OR INFRINGEMENT. WE EXPRESSLY DISCLAIM ANY LIABILITY WHATSOEVER FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOSS REVENUES, LOST PROFITS, LOSSES RESULTING FROM BUSINESS INTERRUPTION OR LOSS OF DATA, REGARDLESS OF THE FORM OF ACTION OR LEGAL THEREUNDER WHICH THE LIABILITY MAY BE ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY LIKELIHOOD OF SUCH DAMAGES

Table of content

1. Introduction

The **SE*cube*™** (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[1], whereas the software libraries stem from a strong cooperation among five international research institutions, including:

- *Blu5 Labs Ltd*, Blu5 Group, Ta Xbiex, Malta –
  Reference: Antonio VARRIALE, av@blu5labs.eu
- *CINI Cyber Security National Lab*, Torino, Italy –
  Reference: Paolo PRINETTO, paolo.prinetto@polito.it
- *Lero, The Irish Software Research Centre*, University of Limerick, Limerick, Ireland –
  Reference: Tiziana MARGARIA, tiziana.margaria@lero.ie
- *LIRMM*, CNRS, Montpellier, France –
  Reference: Giorgio DI NATALE, giorgio.dinatale@lirmm.fr
- *TU Dortmund*, Dortmund, Germany –
  Reference: Bernard STEFFEN, bernhard.steffen@tu-do.de

The level L2 of the security APIs and its functionalities are presented in Section 2.

Functionalities offered from level L2 are conceptually grouped into two main projects: **SE*file***, detailed in Section 3, and **SE*link***, detailed in Section 4.

The software libraries, in conjunction with the above-mentioned 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 **SE*cube*™** platform and experience the platform as a high-security black box. Conversely, security experts can enjoy the openness and good documentation to verify, change or re-write the pre-existing software, starting from basic low-level blocks or even redefine entirely the whole system.

All the software is released in source code under GPLv3 license[2].

Leveraging the platform thought, we intend to create and nurture over time a community for developers at the different levels of security competence and in different application domains. This will ease sharing project, knowledge, and resource and provide the collectivity of members with specialized support tailored to their needs.

[1] www.blu5group.com

[2] https://www.gnu.org/licenses/gpl-3.0.en.html

2. The L2 Security APIs

The **SE***cube*™ Hardware device family comprises 3 major members, which are detailed in the "Getting Started" manual (please refer to Chapter 2):

- The *Chip*, named **SE***cube*™ Chip, or simply **SE***cube*™
- The *Development Board*, named **SE***cube*™ DevKit
- The USB Stick, named **USE***cube*™ Stick.

The functionalities of these devices can be expressed leveraging the **SE***cube*™ Open Source Software Libraries, then hereinafter referred to as the **SE***cube*™ SDK, introduced in the "Getting Started" manual (please refer to Chapter 5).

The **SE***cube*™ SDK consists in a set of multi-level and open source C libraries, collectively referred to as APIs in the sequel.

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 **SE***cube*™-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 **SE***cube*™ device.

On the *Host-Side*, where the software is tailored for existing devices (e.g., laptops or Desktop PC) that see the **SE***cube*™ hardware as an external peripheral which exposes services described in the "Getting Started" manual (section 5.2).

The **SE***cube*™ device is thus seen by the host as a closed black box providing services. The host starts the service request by sending the related command and the optional data packets, through a proper interface, per a custom protocol.

The *Host-Side* Libraries are designed to be scalable, i.e., for dealing with multiple devices, and portable on different Operating Systems, thus limiting the usage of and isolating platform-dependent modules. They practically run on top of the host OS, directly relying on the OS System calls. To improve portability and migrations, the libraries are organized in such a way that all the OS-dependent sub-modules (e.g., communication interface, file system, etc.) be easily identifiable.

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

- *Communication Protocol* and *Provisioning* APIs (Level0 Host-Side – L0)
- *Basic Security* APIs (Level1 Host-Side – L1)
- *Intermediate Security* APIs (Level2 – L2)
- *Advanced Security* APIs (Level3 – L3).

From the architectural point of view, the libraries have been implemented targeting hierarchical abstraction levels.

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

This document focuses on the *Host-Side* only, and particularly on the Intermediate Security APIs (Level2 – hereinafter referred to as L2).

2.1. The Purpose

Level L2 relies on L1 services to provide the APIs for implementing more abstract secure functionalities (Figure 1). 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.

Level L2 exploits principally the following functionalities provided from Level L1 and L0:

- Secure key storage: cryptographic keys of custom length can be stored inside the device
- Encryption and decryption of data stream: using any of the stored keys
- Authentication of data streams: using any of the stored keys.

In addition, L2 is an abstraction layer that could be easily used for developing new applications (Level L3) on top of it.

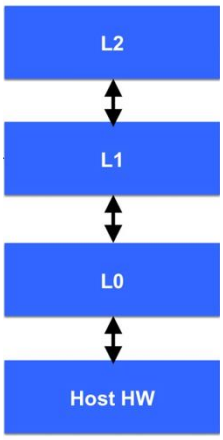

Figure 1 – L2 Services

2.2. Functionalities Offered

While developing the APIs for the L2 level, it has been chosen to include the secure layer in the user space domain, so that it can be directly used by any application programs leveraging to the developed custom functions. Given that, a set of dedicated wrappers have been developed to substitute system calls, especially for what concerns accessing the File System and the network interfaces manager.

The basic principle is that any application using the provided API instead of the standard system calls can create and manage entities that are protected (i.e., authenticated and confidential) in all its parts. This is true both for data intended to be stored on a physical support (i.e., *data at rest*), which are cyphered and signed up to their name, and data package to be sent over an insecure network channel (i.e., *data at motion*), which are hidden from malicious attackers up to their transport-layer header.

Contemporaneously, both the encryption-decryption and the authentication processes are totally transparent to the user/application, which still work in terms of regular files or network packages.

Conceptually, APIs belonging to the L2 abstraction layer expose functionalities needed for any user who wants, by moving inside a *secure environment*, to perform basic operation on regular files and network packages. The concept of a secure environment reflects the need of providing a simple mechanism to allow the user to customize the parameters of the secure session, by configuring the keys to be used and the algorithms to be enforced through all the valid life of the session itself.

APIs belonging to the L2 currently aim at easing the development of secure applications managing *data at rest*, i.e., data meant to be stored on untrusted physical supports, and *data at motion*, i.e., data meant to be sent on untrusted network channels after the establishment of a shared secure session. Given that, L2 is identified as the merge of two projects: **SE***file*, concerning data at rest, and **SE***link*, concerning instead data at motion.

3. SE*file*

Any OS provides an abstraction layer in its kernel space, used to separate file system generic operations from their implementation. This is performed by defining a clean Virtual File System (VFS) interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally. Data protection is provided at this level of abstraction, by means of a dedicated security engine, hereinafter referred to as *secure layer*.

One approach is to develop a secure layer which operates in user space (Figure 2 on the left). This approach allows the developer to provide security functionalities without modifying the underlying operating system, which it is not always permitted. On the other hand, those secure functions do not override the standard ones, instead proposing themselves as a secure alternative. An interesting feature in this case is given by the possibility to develop a portable layer, meaning that it is valid for different Operating Systems (OSs).

Typically, OSs vendors follow instead another approach, based on a security level lying under the virtual file system, hence not guaranteeing portability (Figure 2 on the right). The secure layer, in this case, is transparent to the application/user.

In any case, whichever is the chosen approach, malicious user, or software, still may exploit existing flaws in the application accessing to the secure layer or even in the secure layer itself. A countermeasure to protect effectively data, thus, resorts to hardware key management techniques applied to powerful embedded systems that can perform complex cryptographic operations while, at the same time, increasing the confidence of data security. A secure device can guarantee data protection also when the host machine is compromised.

SE*file* is a file system which exploits the hardware key management exposed from APIs Level L1 and other functionalities from the **SE*cube*™** device (Figure 3). It has been developed having in mind the needs to ensure both simplicity of usage and security for *data at rest*: it allows secure storage, retrieve and usage of information that could not be trusted if stored elsewhere, e.g., any personal computer, or cloud service provider.

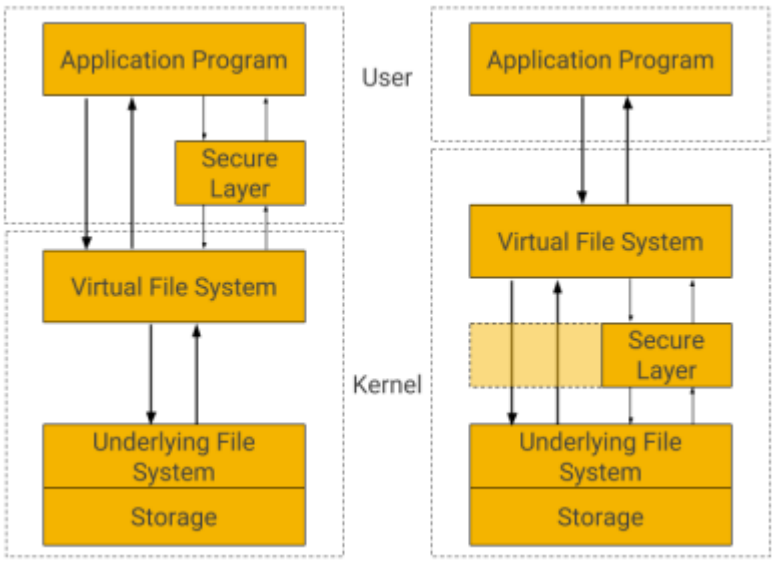

Figure 2 - Secure Layer and Virtual File System: two different approaches

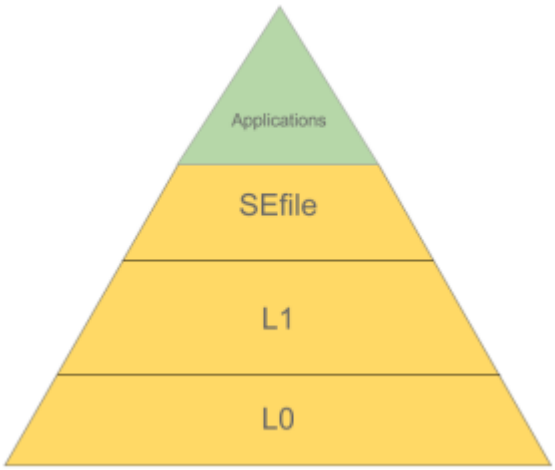

Figure 3 – **SE*file*** hierarchic organization

Conceptually, **SE*file*** 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, **SE*file*** 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.

3.1. Data Confidentiality

One of the most important considerations a Secure File System deals with is the way in which files are encrypted. A Secure File is made up of several encrypted and signed sectors. The first sector is dedicated to the Secure File header, which provides information on the file itself (i.e., length, metadata) and contains a padding if needed, while the other sectors encode file data themselves (Figure 4).

This block structure has the great advantage to allowing data manipulation on subparts of a file by interacting with a subset of its blocks; in this case, there is not the need to decrypt and encrypt the whole file and, in this way, the time overhead is considerably lowered, especially in the common case of limited editing interesting a single block sector.

Figure 4 – Secure File structure

3.1.1. Encryption Algorithm

The Advanced Encryption Standard (AES), also known by its original name Rijndael, is a specification for the encryption of data established by the U.S. National Institute of Standards and Technology (NIST) in 2001[3]. AES has been adopted by the U.S. government and is now used worldwide, becoming a de-facto standard for guaranteeing data confidentiality. It is characterized from being a block cipher, since it is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation of blocks of fixed size, and is fast in both software and hardware.

However, a block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits (i.e., a block). Then, a mode of operation is an algorithm that uses a block cipher to provide an information service such as confidentiality or authenticity. A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.

Most modes require a unique binary sequence, often called an initialization vector (IV), for each encryption operation. The IV should be non-repeating and, for some modes, random as well. The initialization vector is used to ensure distinct ciphertexts are produced even when

[3] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)". Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001.

the same plaintext is encrypted multiple times independently with the same key. Block ciphers have one or more block size(s), but during transformation the block size is always fixed. Block cipher modes operate on whole blocks and require that the last part of the data be padded to a full block if it is smaller than the current block size.

Currently, the APIs, Level L1 and L0, supports only AES as cipher algorithm. **SE*file*** leverages it by using the Counter (CTR) mode of operation.

The simplest of the encryption modes is the Electronic Codebook (ECB) mode. The message is divided into blocks, and each block is encrypted separately.

Counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting successive values of a "counter". The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. Today, CTR mode is widely accepted and any problems are considered a weakness of the underlying block cipher, which is expected to be secure regardless of systemic bias in its input[4].

This said, **SE*file*** uses an encryption scheme as follows. Each sector, except the header, is encrypted using **AES-256-CTR**, meaning that each block cipher depends on an ascending counter which start from a randomly selected initialization vector, generated using ad-hoc functions provided from the crypto engine of any OS.

The header sector, instead, is encrypted using **AES-256-ECB**, to be independent from any initialization vector.

[4] Helger Lipmaa, Phillip Rogaway, and David Wagner. Comments to NIST concerning AES modes of operation: CTR-mode encryption. 2000

3.2. Data authentication

On the other hand, there exists the problem of guaranteeing the integrity of the whole file against malicious attackers: each sector is signed so the integrity and the authenticity of each one can be easily checked.

3.2.1. Algorithms

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS).

In particular, SHA-3 uses the sponge construction, in which data is "absorbed" into the sponge, then the result is "squeezed" out[5]. In the absorbing phase, message blocks are XORed into a subset of the state, which is then transformed. In the "squeeze" phase, output blocks are read from the same subset of the state, alternated with state transformations. The size of the part of the state that is written and read is called the "rate" (often denoted r), and the size of the part that is untouched by input/output is called the "capacity" (often denoted c). The capacity determines the security of the scheme. The maximum-security level is half the capacity.

Finally, a keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. It may be used to simultaneously verify both the data integrity and the authentication of a message, as with any MAC. Any cryptographic hash function, such as MD5 or SHA-3, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA accordingly. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output, and on the size and quality of the key.

An iterative hash function breaks up a message into blocks of a fixed size and iterates over them with a compression function. For example, MD5 and SHA-1 operate on 512-bit blocks. The size of the output of HMAC is the same as that of the underlying hash function (128 or 160 bits in the case of MD5 or SHA-1, respectively), although it can be truncated if desired.

Within **SE*file*** each sector, including the header, is signed using an authenticated signature obtained with **SHA-256-HMAC**, meaning that the signature depends on both the data contained in the sector itself and on a chosen encryption key. To use two different keys to encrypt data and to digest authentication, a feature increasing overall system security, **SE*file*** leverages on the **pbkdf2()** function already implemented within the SDK. This function, provided with a 32 Bytes long *salt* vector (randomly chosen), is used to generate parameters needed for the secure sessions, such as a new key and the number of iteration of the authentication procedure. This mechanism is important to enhance security, because even if one key is unveiled, the second one would be too difficult to obtain.

The procedure enforced within **SE*file*** to ensure data protection and confidentiality is hereby described. Firstly, the file is divided into chucks, sectors containing each exactly 512 Bytes (constant defined as *SEFILE_SECTOR_SIZE*).

Except for the first (the header), each sector is divided into three main fields: data, length and signature. The length field is composed of 2 bytes and stores the number of valid user

[5] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. "The Keccak sponge function family: Specifications summary".

data bytes contained in the data field. The signature field is composed of 32 bytes and stores the result of the authenticated signature, to check whether the sector is corrupted or not and if the data stored in sector were written by an authorized user.

The data field represent the effective payload where the user data are stored, it can be computed as *SEFILE_LOGIC_DATA = SEFILE_SECTOR_SIZE - 2 - 32 (Bytes)*.

The overhead (in Bytes) resulting from this kind of file structure can be easily computed:

$$Overhead = SEFILE\_SECTOR\_SIZE + \\ + \left(\left\lceil \frac{File_{size}}{SEFILE\_LOGIC\_DATA} \right\rceil - 1 \right) \cdot (32 + 2)$$

For example, encrypting a file of 2 GiB with *SEFILE_SECTOR_SIZE* = 512 Bytes will produce an overhead of 152.70 MiB, while for the same file if *SEFILE_SECTOR_SIZE* is set to 4096 Bytes instead its total overhead is 17.15 MiB.

The first sector of the Secure File follows a different structure from the others and is not used to store user data, containing instead several information about the file itself, as follows:

```
typedef struct {
        uint8_t nonce_pbkdf2[32];
        uint8_t nonce_ctr[16];
        int32_t magic;
        int16_t ver;
        int32_t uid;
        int32_t uid_cnt;
        uint8_t fname_len;
} SEFILE_HEADER;
```

The first two vectors of the header are respectively a 32 Bytes long *salt* used for generating a different key to authenticate digest, while the second one is the random initialization vector used as counter for encrypting all the data sectors of the secure file. The *fname_len* field contains the length of the filename which is written right after the header fields.

The *magic* field might be used for representing what type of file it has been encrypted. The *ver* field is used for representing with what version of **SE*file*** it has been encrypted. The *uid* and *uid_cnt* fields, finally, are designed to host information about the user who encrypted the file and its permission. However, all these last features are not supported yet.

All the unused bytes for padding of the header sector, and all the unused bytes obtained when any sector is not filled up to its capacity, are randomly chosen in order to avoid a *known plain-text attack*, an attack model for cryptanalysis where the attacker has access to both the plaintext (called a crib), and its encrypted version (ciphertext). These can be used to reveal further secret information such as secret keys and code books.

3.3. Running the Provided Demo

3.3.1. Secure Text Editor and Secure Image Viewer

The first demo provided deals with a typical case for file editing: reading and writing from and to text files and images.

Both these two projects have been developed in C++ with Qt libraries. They are based on 3 major security classes, in a one-to-one mapping with the 3 most important security operations: the first one manages the security platform to which the user wants to log in, the second one allows the selection of the secure environment through the **secure_update()** function, while the third one manages the opening and creation of files resorting on the **secure_ls()**.

Both these applications were both tested on Windows and Linux[6]. To compile and launch the Secure Text Editor, is sufficient to acquire the "SEfile_TXT" folder and to import the Qt project "SEfile_TXT.pro" stored within. The project can be built in a straightforward manner directly from Qt. Similar conditions hold for the Secure Image Viewer, where the folder of interest is "SEfile_IMG" and the project "SEfile_IMG.pro".

The two applications work similarly.

The Secure Text Editor can open both a plain-text file or a cipher-text file; once the file is opened (no matter what it is) the corresponding encrypted/decrypted version will be created in the same directory. In case the user wants to generate a new file, he can do it by just starting writing on the left box and then clicking on Save both button. It is possible to verify that encrypted files cannot be read properly from regular text editors; conversely, the Secure Text Editor can transparently read any encrypted file (decrypting also the file name) which content has not been altered and is, thus, trusted. Unauthenticated content (i.e., content not corresponding to the file signature) is, instead, discarded.

The Secure Image Viewer, similarly, can open both a plain-text image file or a cipher-text image file; once the file is opened (no matter what it is) the corresponding encrypted/decrypted version will be created in the same directory. At the present, the software supports the three most common image file formats: PNG (Portable Network Graphics), JPG/JPEG (Joint Photographic Experts Group) and BMP (Bitmap image file). It is possible to verify that encrypted images cannot be displayed from regular viewers; conversely, the Secure Image Viewer can transparently read any encrypted file (decrypting also the file name) which content has not been altered and is, thus, trusted. Unauthenticated content (i.e., content not corresponding to the file signature) is, instead, discarded. To test this assumption, is possible, for instance, to open both a plain-text and a ciphered BMP file with a binary editor; while in the first case it is easy to modify Bytes (i.e., pixels of the image) leaving no trace (a regular image viewer will continue displaying the image, and users may not notice data tampering), in the second case the resulting file will not be recognized as valid (being modified from a malicious attacker) and no image viewer, not even the Secure one, will open it.

[6] Tested on Windows 7 x64 and Ubuntu 14.04LTS kernel 3.16

3.3.2. SQLite and DB browser for SQLite

SQLite is a database engine developed in C and freely distributed online[7]. Leveraging on its modularity, the SQlite system has been modified to resort on a custom functionalities wrapper based on **SE*file***, rather than using directly the OS calls.

The starting point of this work was the template offered as example for making a custom VFS interface distributed along with SQLite, version 3.13.0. The outcome has been a secure version of the SQLite, hereinafter referred to as *secureSQLite*.

DB Browser for SQLite is an open source project developed in C++ with Qt libraries and consists of an application that let the user browse a database and perform some basic operations to manage it[8].

The provided demo integrates *secureSQLite* with this application to show the potentialities of this secure library. To achieve such purpose, this demo leverages on Qt Creator as Integrated Development Environment (IDE) and on DB Browser for SQLite version 3.9.0. The demo project includes both the libraries **SE*file*** and *secureSQLite*, with regards to the APIs described in the following Paragraph. With few simple modifications, the DB Browser for SQLite works flawless in a secure fashion, ensuring that information stored in a DB are not accessible from unauthorized users.

To run it, it is needed to acquire and extract the "SECUREsqlitebrowser-3.9.0" archive. The folder contains a Qt project, already set to compile, and produce, a regular (i.e., insecure) version of the DB Browser for SQLite. Instead, to produce the secure version, it is simply needed to add the "CONFIG+=SQLITE_OS_SECURE" define to the project, as in Figure 5[9].

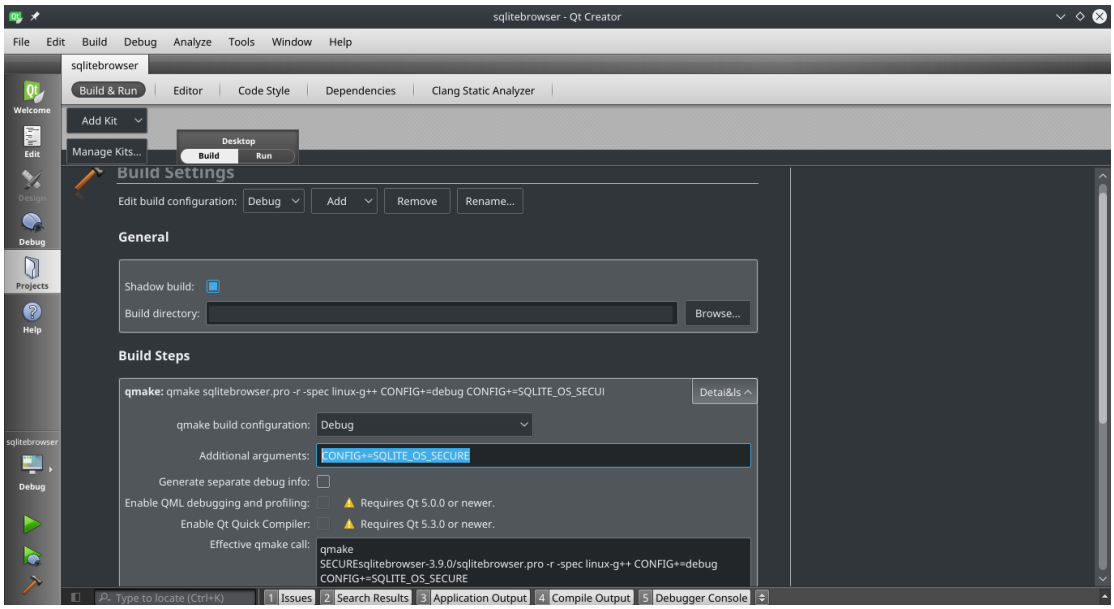

Figure 5 – Qt project interface for compiling the *secureSQLite*

[7] https://sqlite.org/ - All the code external from **SE*file*** is intended not of our property and is released respecting its original license and terms of use

[8] http://sqlitebrowser.org/ - All the code external from **SE*file*** is intended not of our property and is released respecting its original license and terms of use

[9] Tested on Ubuntu 14.04x64 LTS Kernel 3.16. Compilation may require the the compile flag "_GNU_SOURCE".

The executable resulting from the compilation will act almost exactly as its insecure counterpart, with two major differences, nevertheless maintaining the same user interface (Figure 6).

The first one is visible to the user, as he/she has the possibility to select one of the security environments (algorithm and key) supported from the device by using the “Set Environment” function.

The second one is not directly visible to the user, and represents the key aspect of the demo project: every DB resulting from a commit (i.e., “Write Changes” operation) is cyphered and signed up to its file name before being stored, making it unreadable from the regular insecure DB Browser for SQLite. Instead, any DB produced from the regular browser is easily readable, and modifiable, from any binary editor, making it a untrusted for storing DB with private information.

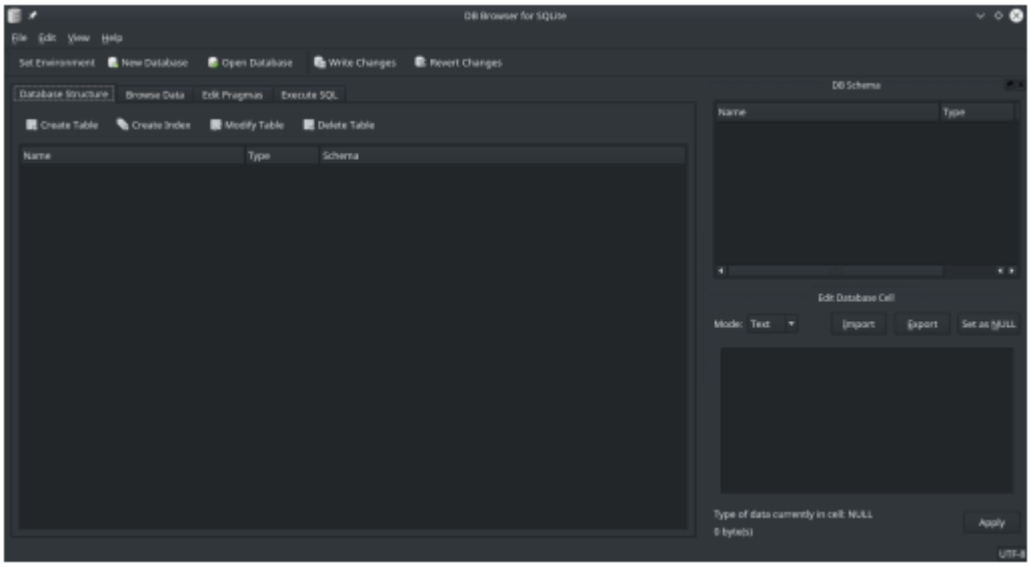

Figure 6 - Overview of DB Browser for *secureSQLite*

3.3.3. The SQLite APIs Commented

Any application leveraging on the *secureSQLite* oversees the initialization and management (up to the release) of all its resources; this is set to not enforce any constraint on the application, which might be, as instance, either based on a command line interface or on a graphical user interface.

The starting point of this work has been the official template offered as example for making a custom VFS interface distributed along with the original version of SQLite. This template is a simple example for creating a working interface for Unix environment that also implement a simple software cache for reducing the number of disk accesses. To force SQLite to use this VFS interface instead of the standard ones, it was implemented a mechanism based on precompiler definitions.

Hereby it is listed a subset of the most important VFS interface functions that have been implemented to develop the provided demo with a brief explanation on their usage and implementations.

SQLITE_API int SQLITE_STDCALL sqlite3_os_init(void)

SQLITE_API int SQLITE_STDCALL sqlite3_os_end(void)

These two functions are respectively used to assign and release the data structure made up by pointers to the rest of VFS interfaces that has been associated with common I/O operations.

static int SecureDirectWrite(SecureFile *p, const void *zBuf, int iAmt, sqlite_int64 iOfst)

This function is used to wrap a write operation, it accepts as parameters a custom file descriptor, the buffer that should be written, the number of bytes to be written and the offset from the start of the file. This function checks if the software cache should be flushed to disk, change the file pointer to iOfst and the issue a secure_write() call.

static int SecureRead(sqlite3_file *pFile, void *zBuf, int iAmt, sqlite_int64 iOfst)

This function is used to wrap a read operation, it accepts as parameters a custom file descriptor, the buffer that should be read, the number of bytes to be read and the offset from the start of the file. This function checks if the software cache should be flushed to disk, change the file pointer to iOfst and the issue a secure_read() call.

static int SecureTruncate(sqlite3_file *pFile, sqlite_int64 size)

This function is used to change the size of the pointed file pFile to size. In this case it simply issues a secure_truncate().

static int SecureSync(sqlite3_file *pFile, int flags)

This function is used to flush OS buffers (and not the software cache) to disk thanks to secure_fsync(). In this case the flags are ignored.

static int SecureFileSize(sqlite3_file *pFile, sqlite_int64 *pSize)

This function after writing the pending software cache to disk, it returns the file's current size thanks to secure_getfilesize(). Since sqlite3_file is highly customizable, the path was added to the file descriptor to be compatible to the **SE*file*** API.

static int SecureOpen(sqlite3_vfs *pVfs, const char *zName, sqlite3_file *pFile, int flags, int *pOutFlags)

This function is used to manage opening/creating of a secure database thanks to secure_open(). In this case pVfs and pOutFlags were ignored, while zName is the path to the

file that should be opened, pFile is the pointer to the file descriptor obtained, and flags is used to determine how the file should be opened.

WARNING on the flags:

- The combination (SEFILE_READ, SEFILE_NEWFILE) is not allowed and therefore fails;
- The combinations (SEFILE_READ, SEFILE_OPEN) and (SEFILE_WRITE, SEFILE_OPEN) work only if the file already exists.

static int SecureDelete(sqlite3_vfs *pVfs, const char *zPath, int dirSync)

This function is used to delete a file pointed by zPath and it was developed as a wrapper to unlink() in Unix and DeleteFile() in Windows, thanks to crypto_filename() function. In this case dirSync parameter is ignored.

static int SecureAccess(sqlite3_vfs *pVfs, const char *zPath, int flags, int *pResOut)

This function should mimic the access() systemcall available in Unix environment, and it will set pResOut to 1 if the file pointed by zPath exists and is readable or readable and writable, otherwise to 0. In this case pVfs and flags were ignored.

static int SecureFullPathname(sqlite3_vfs *pVfs, const char *zPath, int nPathOut, char *zPathOut)

This function is used to retrieve the full path of a secure database pointed by zPath by writing at most nPathOut bytes to zPathOut. In this case pVfs is ignored.

4. SE*link*

SE*link* is a software application that uses the **SE*cube*™** open platform to secure the network traffic. It can encrypt network streams originating from any application, regardless of the application-level protocol.

SE*link* is a reference implementation of an application on top of the Layer1 API for **SE*cube*™**. By using **SE*link*** it is possible to add a secure network layer to any software, without modifying its code base. In other words, the user can employ any of his/her favorite network-enabled software (e.g., web browser, remote desktop viewer) and entrust the customizable security features to the **SE*cube*™** platform, in a way that is completely transparent to the user, allowing him to exploit the benefits of the security functionalities without having deep knowledge about security.

The software does not need to be aware of the presence of **SE*link*** and will function as usual, because **SE*link*** intercepts connections at a lower level.

SE*link* is made up of two macro-components (Figure 7):

- **Client-side software**: it is installed on the host that initiates the connection. It includes a driver, a background service and a graphical user interface. The driver intercepts outgoing connections and redirects them to the service. The service bridges the connection to the destination, applying the encryption layer
- **Server-side software**: it is installed on the host that accepts the connection. It includes a background service and a graphical configuration utility. The service is symmetrical to the client-side service, bridging the encrypted connection to its final destination.

The Client-side components are:

- **SE*link*** driver
- **SE*link*** service
- **SE*link*** GUI.

The driver is needed to intercept all new TCP connections, system-wide, while not modifying any application (Figure 8). The driver redirects all connections to the service, which can decide whether each should be encrypted or not. The graphical user interface is a distinct application too, because Windows services are not intended to create GUI elements within the user's session.

The server-side components are:

- **SE*link*** gateway
- **SE*link*** gateway web UI.

On the server's side a "gateway" application, running as daemon, bridges the secure connections created by a **SE*link*** client host to any server software (Figure 9). This daemon can by configured by directly editing its configuration file, or by using a graphical interface through any web browser.

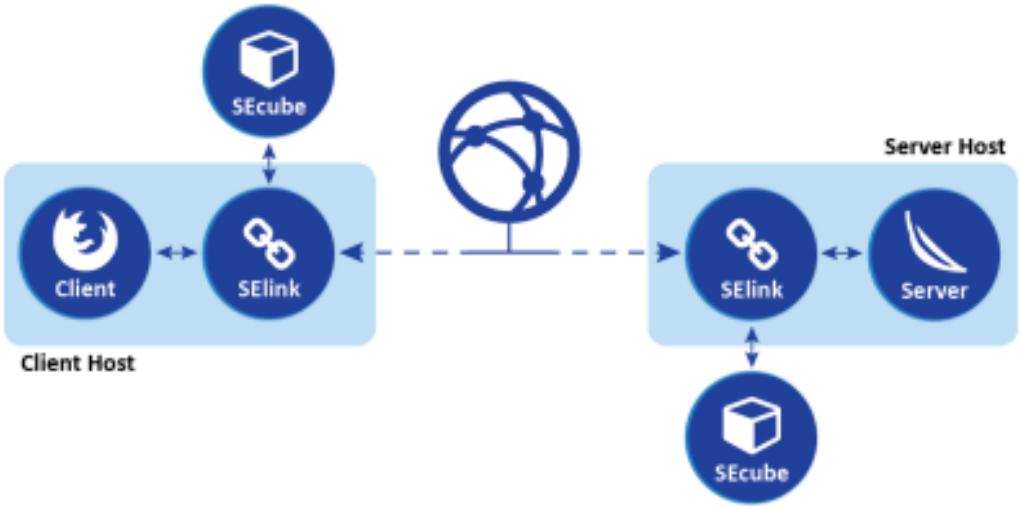

Figure 7 - SE*link* architecture

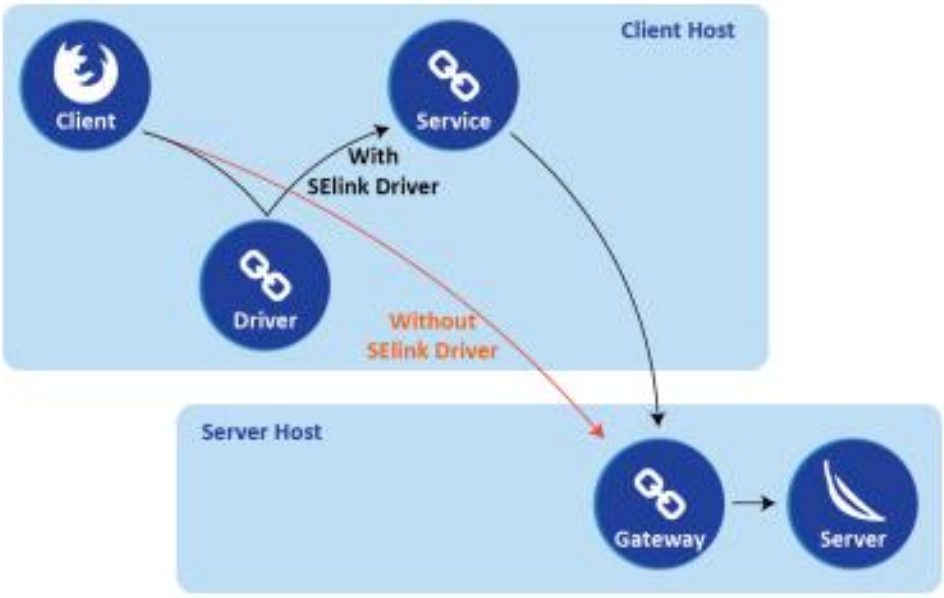

Figure 8 - Connection establishment process; each directed arrow represents a TCP connection request

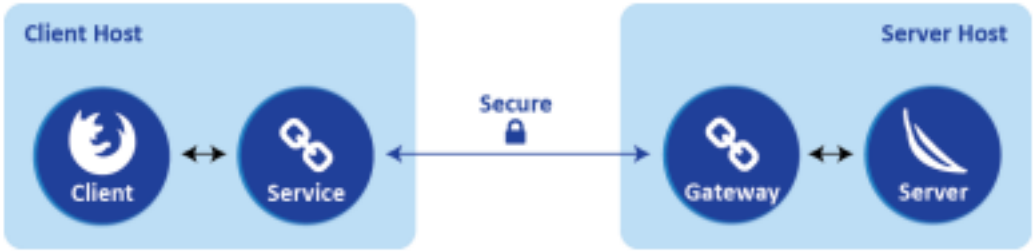

Figure 9 - final connections; each bidirectional arrow represents a TCP connection

4.1. SE*link* driver

The choice of implementing a driver has been taken since it cannot be tampered from other user-mode applications, making it more secure and reliable; moreover, it leaves no traces within the applications' executable memory, so it goes undetected by most software protection frameworks and does not interfere with applications.

After some experiments with other frameworks, the driver was ultimately developed with the Windows Filtering Platform (WFP) API. The superseded Transport Driver Interface (TDI) API was discarded a priori, because of its deprecation. NDIS, despite having all the needed features, would have required more effort to achieve the same result.

The current WFP solution is very lean, consistent with the practice of performing the least needed amount of work within the kernel. In fact, any faulty code within the kernel can cause a system crash, hence the need to restrict kernel-mode code to the minimum necessary amount.

An important reason why the filtering logic has been implemented in user-space is that not all operations can be performed as easily within the kernel. To be more specific, routines within the kernel run at different Interrupt Request Levels (IRQL), depending on their priority. Each of the IRQLs has a mask for interrupts, with higher levels masking more interrupts. At higher IRQLs functions must complete as soon as possible, deferring any time-consuming work.

Also, functions that can cause a page fault, such as operations on pageable memory, must only be performed at the lower levels that do not mask the specific interrupt. Specifically, WFP filter callback functions run at *IRQL = DISPATCH_LEVEL*, and cannot access pageable memory nor perform file I/O.

The **SE*cube*™** platform uses file I/O to communicate with the device; its host API would need to be completely redesigned to use it within a driver. The driver is written in C, as most drivers are. There is no official support to any other language for driver development with the Windows Driver Framework. As a matter of fact, most functions from the C standard library cannot be used either.

For network filtering the API Hooking approach was considered and discarded be- cause of the disadvantages discussed in the previous chapter. LSPs have not been considered for the deprecation issue. Other user-space options involve either modifying applications to add support for **SE*link*** or restrict the applicability of **SE*link*** to applications supporting proxy protocol (e.g. HTTP/HTTPS proxy or SOCKS). Hence the decision to develop a kernel-mode filter.

The NDIS API was considered first, but considerable effort would have been required to keep track of TCP streams from the level at which NDIS filters operate. Windows Filtering Platform (WFP) has been chosen over NDIS because of its sup- port for transport level filtering. Also, WFP is recommended by Microsoft itself over the alternatives.

The driver has the main purpose of redirecting outgoing TCP connections from any application to a local proxy service.

The driver needs to exchange information with the service mainly for the following purposes:

- Sending information about the redirected connection, so that the service can bridge the connection to the intended destination
- Getting the process ID of the service.

Connection information is passed to the service through the redirect context, as stated before. As for the PID, the service sets a WFP Provider Context at startup, which can be read within the driver. Since the provider context cannot be read within the callback functions, because of the IRQL, a periodic timer task running at *IRQL = PASSIVE_LEVEL* polls the provider context at a fixed interval.

4.2. SE*link* service

The **SE***link* service is the main component of the client-side software: all the connections originating from the machine go through the service, which decides which connections should be encrypted and which should not.

The choice of the programming language, C++, is driven by multiple reasons:

- It is mandatory that the service performs well with many connections (hundreds to thousands) because it must handle most of the network traffic of the machine
- The service needs some data structures such as maps, sets, linked lists, which are not part of the C standard library
- The service must interface directly with the Windows API to be able to communicate with the driver
- The service must use a **SE***cube*™ device, whose API is only available for C.

C++ is a multi-paradigm language that seamlessly combines low-level and high-level features. It can directly include C code, perform raw virtual memory access, but is also suited to object-oriented and functional approaches. Plus, following the introduction of the C++11 standard, it has vast standard library that includes a common interface to some OS-specific features such as threading.

The boost library is also included for the following features:

- Command line parameters: used to parse the command line, generate a help message and useful error description messages
- JSON file parsing: for the configuration files
- Filesystem operations: to manipulate paths and access files through a OS-independent interface
- String formatting: used for some log messages
- Logging: to conveniently categorize log messages and possibly redirect them to a log file
- Hashing: for faster matching of an array/string against a set of arrays/strings
- Event-based socket I/O: used to accept connections and react to network events.

Since this application shares much of the logic with its cross-platform server-side counterpart, most of the classes are designed to be cross-platform, and are reused in **SE***link* gateway.

The service is intended to be a high performance and low footprint application; therefore, C++ was chosen for its unique combination of performance and high-level features. Boost further extends C++ with cross-platform implementations of additional features. Notably a high-performance event-driven socket I/O library is included in Boost.

The service oversees encrypting/decrypting and forwarding outgoing connections on the client side.

SE*link* service is configured by means of an external GUI application, which can send some commands to the service to perform operations or retrieve information.

The commands are hereby listed:

- **reload**: reload the filter rules file and update the filter rules
- **status**: query the device and service status
- **discover**: discover all connected devices, returning a list of their paths and serial numbers
- **set_device**: select a device by its serial number and pass its password
- **reset**: disconnect from the device and clear the device configuration.

The service creates a named pipe at startup, on which it listens for connections from the GUI. For each connection to the named pipe, a single request packet is processed and a response packet is returned, then the pipe is disconnected.

A thread is devoted to inter-process communication only. All the I/O operations for the named pipe are handled asynchronously with Windows overlapped I/O functions, and all wait operations, in addition to waiting on I/O completion with a timeout, also wait on a stop event. This makes it possible to immediately stop the inter-process communication thread when the service closes, without resorting to polling.

SE*link* service can register itself as a Windows service, and be managed through the Windows' services interface.

Running a process as a Windows service implies that:

- It runs as the SYSTEM user
- It can be configured to run at startup
- It can be enabled, disabled, started, stopped or deleted from a simple command line interface or from the Windows Management Console.

Summarizing, starting **SE*link*** is as simple as issuing the following commands to the Windows command prompt:

```
@REM start the service
sc start selinksvc
@REM start the driver
net start selinkflt
```

4.3. SE*link* service

An important part of the project is allowing users, with little prior knowledge of cryptography and programming, to use **SE*link*** for the practical purpose of encrypting network traffic. Another point for the GUI is the necessity to provide an interactive dialog window to log into the **SE*link***, and avoid leaving the passphrase in the command line or configuration file.

The GUI could not be included within the service, because there is no conventional way of presenting a GUI to the user while running as the SYSTEM user. A simple GUI, here described briefly, has been developed for the cited reasons.

The Windows Presentation Framework (WPF) is a framework for Windows GUI applications, part of the .NET framework. It was chosen for the development of **SE*link*** GUI because of its overall features and good integration with the operating system. The choice was not restricted to cross-platform framework, because the application is specifically made for the SElink client-side software, running on Windows.

Among the .NET and WPF features that were used there are:

- Windows tray icon functionalities
- JSON parser
- Customizable DataGrid GUI component.

When the application is run, it creates a clickable icon in the tray area, which is used to access the configuration dialogs. There are two main dialogs: the device selection dialog and the filter rules dialog, explained in the next sections (Figure 10).

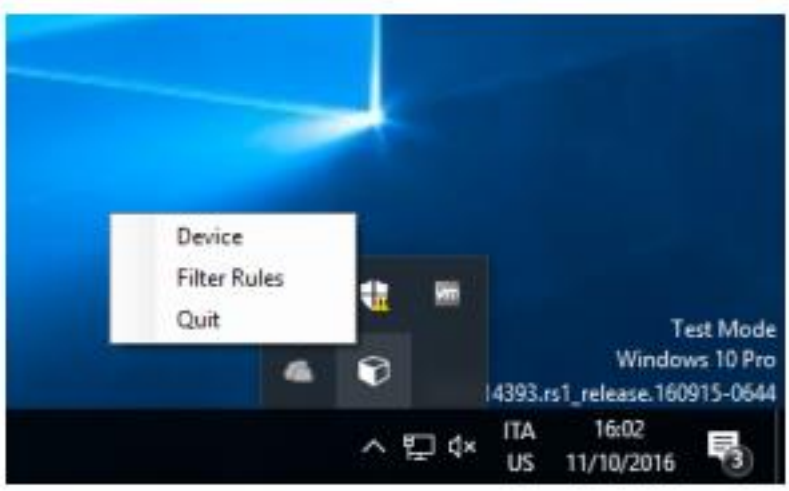

Figure 10 – Tray icon interface

For the service to connect and work properly, the user must select a suitable device (Figure 11).

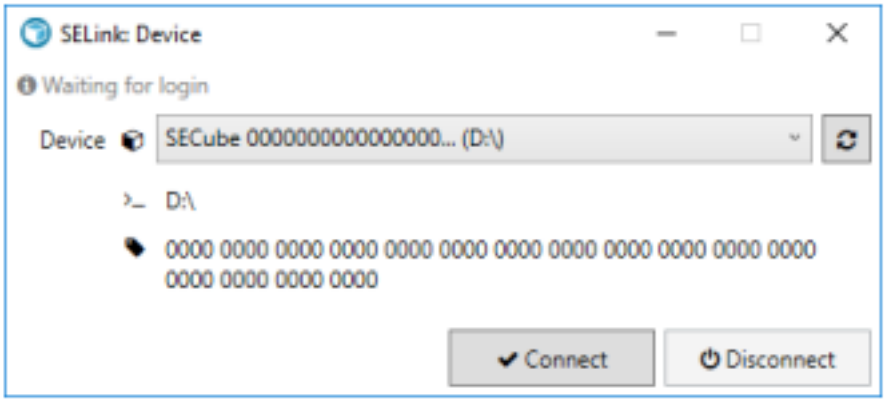

Figure 11 – Device selection interface

A passphrase is required to unlock the device (Figure 12).

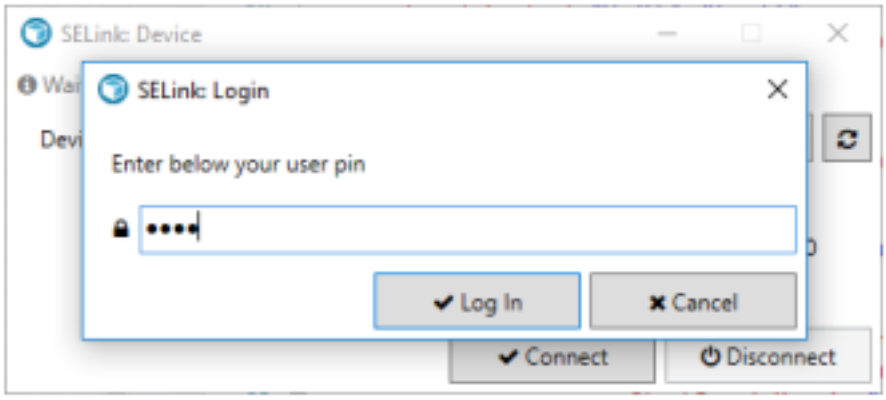

Figure 12 – Unlocking the device

After confirming the passphrase, the outcome of the operation is shown via balloon notification (Figure 13). The GUI does not directly perform any operation on the devices. Instead it uses the service by sending commands through a named pipe.

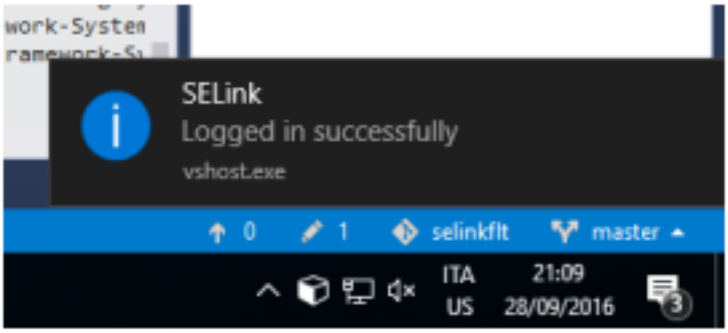

Figure 13 – Login notification

The filter rules window provides an easy way to edit the filter rules configuration file, minimizing the risk of making mistakes or producing an invalid configuration. Based on the Data-

Grid WPF component, it allows adding, removing and editing the filter rules. Rules are processed in order and only the first matching rule is considered.

Therefore, there is the possibility for a rule to cover one of the following rules, which means that any entity matching the second rule also matches the first rule, so the second rule will never be used. The insertion of a masking rule may or may not be intentional, therefore a warning is shown if there are any masked rules, and which is the first masking rule for each masked rule (Figure 14).

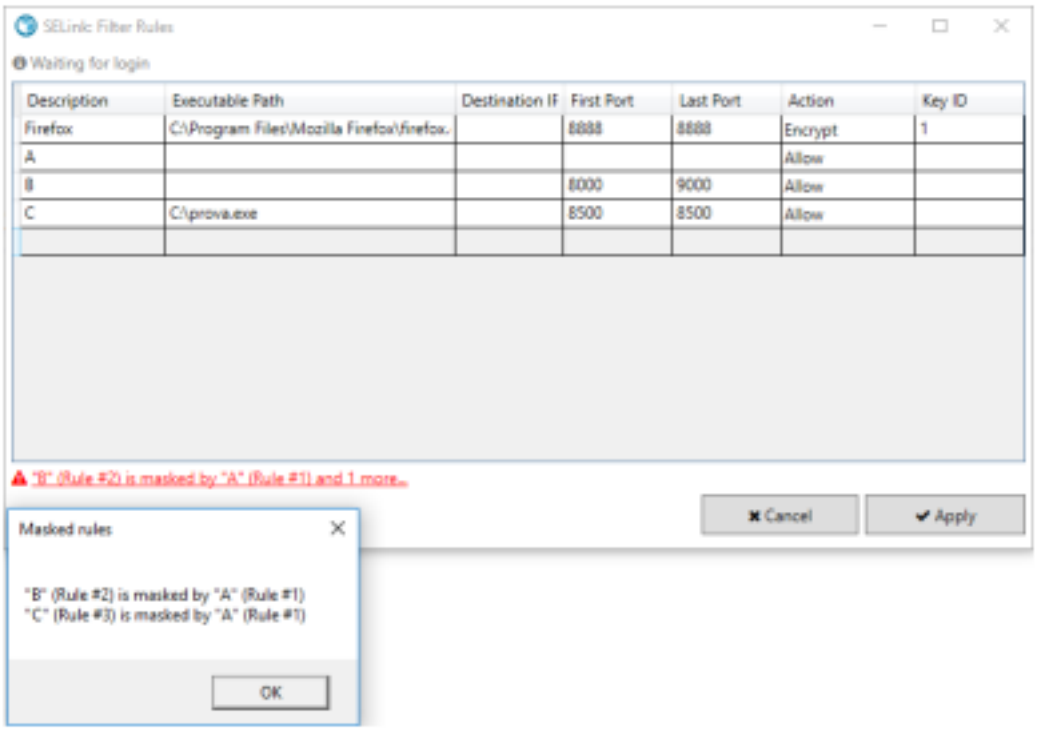

Figure 14 – Filter rules editor

All fields within each rule are validated, to prevent creating an invalid configuration. Each row of the grid shows a relevant error if it does not pass a validation rule (Figure 15).

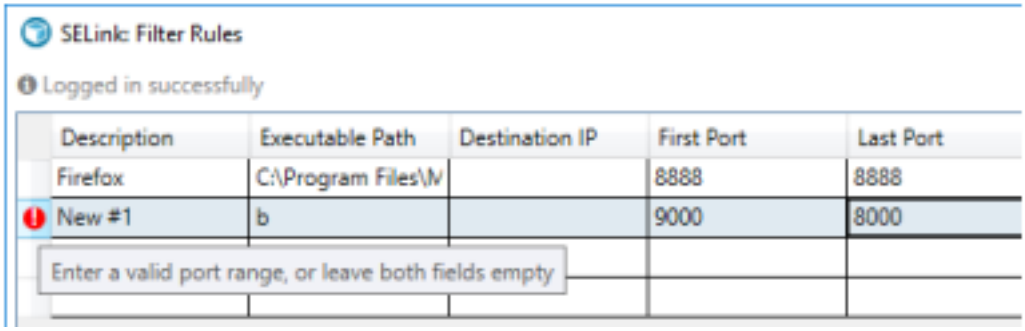

Figure 15 - Figure 16 – Filter rules editor highlights an invalid rule

As soon as the rules are applied, the rules file is written and the service is signaled to reload the file.

4.4. Running the Provided Demo

The main goal of this demo is to show of a Windows software (gateway) that transparently intercepts and encrypts/decrypts TCP streams, and its server-side counterpart, a cross-platform proxy that intercepts and decrypts/encrypts the streams.

This software use the Layer1 API of the **SE*cube*™**open platform to perform the encryption step. The software can encrypt network streams originating from any application transparently, without the need to modify or recompile the application.

In addition, the final product is very be user-friendly and configurable by means of a graphical user interface, both client-side and server-side, and easy to install and capable of running in background silently, requiring minimum user attendance after setup.

To establish a secure communication, a properly configured host running the client software must initiate a connection towards a properly configured host running the server software. In most cases the user will only need to install and configure the client software.

4.4.1. Requirements

Client requirements:

- 64-Bit Windows 10
- **SE*cube*™** device[10].

Server requirements:

- Any of:
  - Windows 7 or newer[11]
  - Linux
- **SE*cube*™** device[8].

4.4.2. The Client Software

The only prerequisite is to install the client software is the Windows C++ Runtime library, freely accessible on-line[12].

There are three components to be installed for the software to operate properly:

1. **SE*link*** driver
2. **SE*link*** service
3. **SE*link*** GUI.

The provided setup executable installs all the components at once.

Since the driver is not signed, Windows 10 needs to run in Test Mode to install the driver.

[10] SElink can be used without a SEcube device, for testing purposes or custom configurations.

[11] Daemon mode not yet supported on Windows.

[12] https://www.microsoft.com/it-it/download/details.aspx?id=48145

To do so:

1. Open an administrator command prompt and execute the following command:
   bcdedit /set TESTSIGNING ON
   Warning: this command disables driver signature check enforcement on Windows.
2. Reboot

Finally, run **selink-setup.exe**.

After the setup procedure, the **SE***link* client software can be easily configured through the GUI application. You can access the device configuration window by left-clicking the tray icon and choosing Device. The status of the service and the device is shown on the top.

To connect a **SE***cube*™ device:

1. Select a suitable device from the drop-down menu. If a device does not show up, press the refresh button on the right.
2. Press the Connect button
3. Insert your user pin and click the Login button
4. The device configuration window should disappear, and a notification should appear shortly after.

To disconnect it, it is simply needed to press the Disconnect button.

You can access the filter rules configuration window by left-clicking the tray icon and choosing Filter rules.

Filter rules are used to manage outgoing connections, to decide which will be encrypted and with which key. A filter rule is made of:

- A condition to select connections based on some parameters:
  - **Executable path**: full path to the source application's executable file
  - **Destination IP**: destination IP address
  - **First port**, **Last port**: destination port range.
- An action to be taken with the matching connections:
  - **Action**: one of *Allow*, *Block* or *Encrypt*
  - **Key**: the key ID to be used when encrypting.

In order for a condition to match a connection, all the parameters must match. An **empty parameter** stands for **any value**.

Within the GUI you may define a list of filter rules, to select different actions for different connections. For example, the user might assign a different encryption key to each application you are going to use. Note that the **order** in which rules appear **is important**: the first matching rule is chosen, regardless of the following rules. If a connection does not match any rule, it is allowed (not encrypted) by default.

You can create a new rule by doing any of the following:

- Right-click on the grid and select Insert after or Insert before
- Fill the last row of the grid

The rules' fields must comply with the following constraints:

- **Description** must be shorter than **64 characters**
- **Destination IP** must be a valid **ipv4** or **ipv6** address
- **First port** and **last port** must describe a **valid port range**
    - first port, last port must be both natural numbers lesser than 65536, or both empty
    - if not empty, first port must be lesser than or equal to last port
- The **key** must be
    - An integer number if the action is *Encrypt*
    - Empty if the action is *Allow* or *Block*

Tip: you can override the default action by adding a rule with empty condition fields at the end of the list.

Drag and drop a row over the desired position to move the corresponding rule. Right-click on a row and select Delete to delete it.

4.4.3. Server side

To install the needed software on server side it is sufficient to:

1. Install g++ and the boost development libraries for your system[13]
2. cd into the **SE*link*** source code directory and build the **SE*link*** gateway

   $ make
3. Install
    - Default installation
        1. **# make install**
           The software and configuration files will be installed in `/opt/selink/`
           It will be configured to run without a SEcube device, using the keys from `/opt/selink/keys.json`
           The systemd unit will be installed in `/usr/lib/systemd/system/selinkgw.service`
    - Custom installation
        - Copy bin/selinkgw and any needed configuration file to a target directory

[13] e.g., **# pacman -S base-devel boost** on Arch Linux

- Customize the "system" unit in "example/selinkgw.service" and copy it to the appropriate location for the system.

The command to start the daemon is: **# systemctl start selinkgw**; to stop it launch: **# systemctl stop selinkgw** .

The configuration is made up of a simple list of port mappings. Each entry contains:

- A description text
- The port on which encrypted connections will be accepted
- The host and port to which connections will be redirected, unencrypted
- The key id to use for encryption.

You may configure the **SE*link*** gateway in two ways:

- Using the web UI
- Using the configuration file.

To use the web UI:

1. Install python3, python3 modules bottle, and jsonschema on your system[14]
2. cd into the gwconfig directory and run the Web UI as root (assumes a default installation)
   # python3 gwconfig.py --use-token
3. Open the link on the terminal output with a web browser

Within the web UI you can add a new rule or delete an existing one.

To add a new rule, press Add and insert the rule. The mappings' fields must comply with the following constraints:

- Listen port and Redirect port must be natural numbers lesser than 65536
- Redirect host must be a valid ipv4 or ipv6 address.

To delete a rule, press the trash icon in the last column of the row to delete. After each modification, please remember to press the Apply button. The new configuration will be saved and applied immediately.

Viceversa, it is possible to edit directly the configuration file as follows:

1. Edit the configuration file with a text editor. Please refer to "example/selinkgw.json" for an example file
2. Signal the daemon: **# systemctl reload selinkgw**

[14] e.g., **# pacman -S python python-bottle python-jsonschema** on Arch Linux

4.4.4. Advanced configuration

The driver can be configured to only filter connections with destination ports within a port range, and allow anything else, regardless of whether the service is running.

The driver configuration is stored in the following registry key: *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Service\selinkflt\Parameters*.

Name	Type	Description
PortFirst	REG_DWORD	First port of filtered port range
PortLast	REG_DWORD	Last port of filtered port range
ServicePort	REG_DWORD	Port on which the service is listening for redirected connections

As a convention on file paths, any path starting with ":" is relative to the executable's path. For example, if the executable is located in "C:\SElink\selinksvc.exe", then ":selinksvc.json" points to "C:\SElink\selinksvc.json".

Any relative path is relative to the current working directory, which depends on how the process was created.

INI files

Command line options for the SElink service and SElink gateway may be specified either by passing them as parameters on the command line, or setting them into dedicated configuration files:

- For **SE***link* service, create a file named "selinksvc.ini" in the same directory as the executable.
- For **SE***link* gateway, create a file named "selinkgw.ini" in the same directory as the executable.

An example of a valid ".ini" configuration file is:

```
provider=soft
keys=:keys.json
```

Please note that only long options (i.e., no short keys) are allowed in the configuration file.

SElink service options

Option	Value	Description
--help, -h	(None)	Show a help message
--log, -l	log file path	Set the location of the log file. Defaults to ":selinksvc.log" when running as service, or none when running in foreground.
--config, -c	configuration file path	Set the location of the filter rules configuration file. Defaults to ":selinksvc.json".
--provider, -p	provider type	Set the provider type. Can be "soft" or "secube".
--keys, -k	path to key collection file	Set the keys. Only required if the provider is of type soft.
--port, -w	port	Set the driver connection redirection port.
--foregrou	(None)	Run in foreground instead of running as a service.

SElink gateway options

Option	Value	Description
--help, -h	(None)	Show a help message
--log, -l	log file path	Set the location of the log file. Defaults to /var/log/selinkgw.log when running as service, or none when running in foreground.
--config, -c	configuration file path	Set the location of the filter rules configuration file. Defaults to :selinkgw.json.
--provider, -p	provider type	Set the provider type. Can be one of soft, secube.
--keys, -k	path to key collection file	Set the keys. Only required if the provider is of type soft.
--serial-number, -s	path to key collection file	Set the keys. Only required if the provider is of type soft.

Option	Value	Description
--pin, -z	path to pin file	The user pin to log into the SEcube device will be read from the specified file. Only required if the provider is of type secube.
--foreground, -f	(None)	Run in foreground instead of running as daemon.

SElink gateway web UI options

Option	Value	Description
--help, -h	(None)	Show a help message
--host	host	Host on which the web UI server will listen
--port	port	Port on which the web UI server will listen
--config, -c	configuration path	Path to gateway configuration file
--pidfile	pidfile path	Path to gateway pidfile
--use-token	(None)	Generate a random token to restrict access to the web UI
--debug	(None)	Enable debug mode

4.4.5. The APIs Commented

SE*link* uses the connection redirection feature in WFP, available for redirecting entire TCP streams to a different destination, possibly for filtering. Filtering in WFP is done by registering callback functions (named Callouts) intercepting the desired operations (e.g., connect, send, receive) at the desired layer.

Within the driver there are two callouts: one for the ipv4 connect redirection layer (*FWPM_LAYER_ALE_CONNECT_REDIRECT_V4*) and one for the ipv6 connect redirection layer (*FWPM_LAYER_ALE_CONNECT_REDIRECT_V6*), which means that all connection requests are intercepted. A filter specification is associated to the callout, restricting the intercepted requests to TCP connections on a user-defined port range.

Resources from the Microsoft website, such the Bind or Connect Redirection feature documentation and the sample driver projects, have been taken as reference for the implementation of the driver.

Any intercepted connection request is redirected to the local proxy, that will possibly encrypt the connection and forward it to the intended destination. First, within the callout, a

redirect context is filled with some data that will be useful for the user-space filter. The redirect context is a WFP feature to attach arbitrary data to the connection request, which can then be retrieved by a different application.

For later applying filter rules, the following parameters are added to the redirect context:

- Original source and destination address and port
- Process ID of the process which generated the request.

```
// Fill the redirect context
ConnectRequest->localRedirectContextSize = REDIRECT_CONTEXT_SIZE;
RedirectContext = (SOCKADDR_STORAGE*)ConnectRequest->
    localRedirectContext;
RedirectContext[0] = ConnectRequest->localAddressAndPort;
RedirectContext[1] = ConnectRequest->remoteAddressAndPort;
RtlZeroMemory(&RedirectContext[2], sizeof(SOCKADDR_STORAGE));
ProcessId = (UINT64)InMetaValues->processId;
RtlCopyMemory(&RedirectContext[2], &ProcessId, sizeof(UINT64));
```

Then, the destination address and port are changed within the request headers, which effectively causes the socket to connect to a local service instead of its original destination. WFP also requires the target process ID to be set for local redirection, so the designated field is filled with the service's PID.

```
// Redirect to localhost
INETADDR_SETLOOPBACK((PSOCKADDR)&(ConnectRequest->remoteAddressAndPort)
    );
INETADDR_SET_PORT((PSOCKADDR)&(ConnectRequest->remoteAddressAndPort),
RtlUshortByteSwap(Globals.ServicePort));
ConnectRequest->localRedirectHandle = Globals.RedirectHandle;
ConnectRequest->localRedirectTargetPID = Globals.RedirectTargetPID;

FwpsApplyModifiedLayerData(ClassifyHandle, (PVOID)ConnectRequest, 0);
```

APPENDIX A - SE*file* APIs

This Section provides a brief overview about the **SE*file*** APIs.

For more details about their implementation, please refer to the Doxygen-based documentation.

uint16_t secure_init(se3_session *s, uint32_t keyID, uint16_t crypto)

uint16_t secure_update(se3_session *s, int32_t keyID, uint16_t crypto)

uint16_t secure_finit()

These functions are used to manage the Environmental variables, since they are not public it has been chosen to manipulate the content of this data with proper functions. This choice has been made to specifically avoid the possibility of letting the user tampers those data. After the board is connected and the user is correctly logged in, the secure_init() should be issued. The parameter se3_session *s contains all the information that let the system acknowledge which board is connected and if the user has successfully logged in. This function may set a default configuration thanks to the L1 provided services: it will be used the first available key for encryption, and the first available algorithm that can manage to encrypt and authenticate data at the same time. Since keys must not be shared outside the device, from the host side, the user may just request to use a key represented by a unique ID (uint32_t keyID).

Once the environment is set, the user is still able to edit these variables by calling the secure_update(). In this case, a default configuration cannot be set, but the user is allowed to edit even just one of the three environmental variables.

Once the user has finished all the operations it is strictly required to call the secure_finit() in order to avoid memory leak. After, a new secure_init() can be invoked.

uint16_t crypto_filename(char *path, char *enc_name)

This function computes the encrypted name of the file specified at position path and writes the result on char *enc_name. The filename is computed using the SHA-256 algorithm, so there is no decryption function to obtain its clear text name unless the header sector is decrypted. Since the service which computes the SHA-256 works with 32 Bytes block, its result is always on 32 bytes, and it is represented as hexadecimal values in ASCII encoding, meaning that for each byte there will be 2 character, resulting in a 64 characters' length.

In any case, this function takes care of parsing path so in enc_name will be copied everything that comes before a "/" or "\" character to compute just the hash of the filename to encrypt.

uint16_t secure_open(char *path, SEFILE_FHANDLE *hFile, int32_t mode, int32_t access)

uint16_t secure_create(char *path, SEFILE_FHANDLE *hFile, int32_t mode)

These two functions, given a filename (as clear text), returns a customized file descriptor to an opened secure file, within the SEFILE_FHANDLE variable. Both functions compute the encrypted filename using crypto_filename() and perform their specific functionality:

- secure_createalways generate a new file deleting its content if it already exists;
- secure_open try to open an existing file starting from its name in clear, if the option SEFILE_NEWFILE is set in int32_t access, a new file is always created and any existing file with the same name overwritten.

In both cases, int32_t mode is used for opening the file in read-only or read-write mode. A real write-only mode has not been implemented since there exists a dedicated secure_write() function.

If a new file is created, both functions create the Header sector for it, they allocate the needed space in memory, populate the header structure with the proper information, encrypt and sign the whole sector (except for the nonce_pbkdf2, as it is needed to check the signature of the header sector itself) before writing it on the storage device. The initialization vectors are randomly generated when a new file is created and then they are stored in proper fields of the file descriptor SEFILE_FHANDLE *hFile.

After executing this two functions the file pointer heads toward the virtual file begin (position 0). The following algorithms demonstrate, respectively, how the secure_create() and the secure_open() works.

Algorithm 1 How a secure file is created

```
function SECURE_CREATE(in path, out hFile, in mode)
    Check Secure Environment
    crypto_filename(path, enc_path)
    OS systemcall to create the object pointed by enc_path according to mode
    Populate header
    Encrypt and authenticate sector
    Write sector to file
    return hFile = file descriptor
end function
```

Algorithm 2 How a secure file is opened

```
function SECURE_OPEN(in path, out hFile, in mode, in access)
    Check Secure Environment
    if access = New File then
        return secure_create(path, hFile, mode)
    end if
    crypto_filename(path, enc_path)
    OS systemcall to open the object pointed by enc_path according to mode
    Decrypt and verify header sector
    Populate hFile with needed data
    return hFile = file descriptor
end function
```

uint16_t secure_read(SEFILE_FHANDLE *hFile, uint8_t *dataOut, uint32_t dataOut_len, uint32_t *bytesRead)

The secure_read function masks the read() operation in Unix environment and the ReadFile() function in Windows, adding all the needed operation related to the secure file management. The number of bytes requested in clear is provided in uint32_t dataOut_len while the actual number of read bytes is stored in bytesRead. In details the operations performed are: starting from the position pointed by the file pointer the function extracts sequentially all the sectors related to the requested portion of data to be read, check for its integrity by looking on the signature, decrypts the sector and concatenate the data to be read (in clear) in the output buffer dataOut given as argument. After that, the file pointer points after the last byte read. A read operation issued requesting a number of bytes that is not aligned to the sector size and is not a multiple of SEFILE_LOGIC_DATA will lead to performance degradation, since it still needs to decrypt the whole sector. The implemented functionality is shown in the following algorithm.

Algorithm 3 How a secure file is read

```
function SECURE_READ(in hFile, out dataOut, in dataOut_len, out bytesRead)
    Check Secure Environment
    Check if the file is not being tampered
    bytesRead = 0
    do
        Read, decrypt and verify signature of one sector
        Append data from decrypted sector to dataOut
        bytesRead = bytesRead + data read
        dataOut_len = dataOut_len − data read
    while dataOut_len > 0
end function
```

uint16_t secure_write(SEFILE_FHANDLE *hFile, uint8_t * dataIn, uint32_t dataIn_len)

The secure_write function masks the write() operation in Unix environment and the WriteFile() function in Windows, adding all the needed operation related to the secure file management. The function write in the file the data passed in clear in the buffer. In particular, the function divides the buffer, received as argument, into sectors, encrypts and signs each sector and write it in the specified position in the file. After this operation, the file pointer points after the last byte written.

In this case, it has been chosen to not return the actual number of written bytes since if the operation fails in writing dataIn_len bytes it would result as an error.

If a secure_write operation is issued requesting to write a number of bytes that is not aligned to the sector size and is not a multiple of SEFILE_LOGIC_DATA, since it still needs to decrypt the whole sector, will lead to performance degradation. The implemented functionality is shown in the following algorithm.

Algorithm 4 How a secure file is written

```
function SECURE_WRITE(in hFile, in dataIn, in dataIn_len)
    Check Secure Environment
    Check if the file is not being tampered
    if File pointer not aligned to sector size then
        Read, decrypt and verify signature of one sector
        Store in buffer sector to be written
    end if
    do
        Append data from dataIn to buffer to be written
        Encrypt and authenticate sector
        Write sector to disk
        dataIn_len = dataIn_len − data in buffer
    while dataIn_len > 0
end function
```

uint16_t secure_getfilesize(char *path, uint32_t *position)

This function is used to get the total logic size of an encrypted file pointed by path and storing its result in position. Logic size will always be smaller than physical size given the introduced overhead.

The implemented algorithm is shown in the following, where N_sector is the total number of sectors, Size_sector is the decided size of each sector (e.g., 512 or 4096 Bytes), overhead is equal to the size of len and signature fields, that is 32 + 2 Bytes = 34 Bytes, and Last sector size is the value stored in len field of the last sector.

Algorithm 5 How a secure file size is computed

function SECURE_GETFILESIZE(in path, out position)
  Check Secure Environment
  Check if the file is not being tampered
  Open file pointed by path using secure_open
  Move file pointer to last sector using OS system call
  if *Total number of sector* = 1 **then**
    return *position* = 0
  end if
  Read, decrypt and verify last sector
  $position = N_{sector} \cdot (Size_{sector} - 1) - N_{sector} \cdot overhead + Last\ sector\ size$
  return *position*
end function

Algorithm 6 How a secure file pointer is moved

function SECURE_SEEK(in hFile, in offset, out position, in whence)
  Check Secure Environment
  Check if the file is not being tampered
  Compute file size using secure_getfilesize
  if *offset* > *file size* **then**
    Move file pointer to last sector using OS system call
    Write *offset* − *file size* '0' at the end of the file
    return *position* = *current file pointer position*
  end if
  Compute file pointer destination according to *whence*
  Move file pointer to destination using OS system call
  return *position* = *destination*
end function

uint16_t secure_seek(SEFILE_FHANDLE *hFile, int32_t offset, int32_t *position, uint8_t whence)

This function moves the file pointer of the specified number of bytes taking care of the effective byte of user data and jumping the bytes related to the overhead introduced by the secure file management (i.e., header sector, signature field and data length).

To mimic the standard OS provided functions, the parameter whence is used to choose if the user wants move the file pointer from the file beginning, from current position, or from its

end. The pointer to position is used to store the logic value where the file pointer is after issuing secure_seek().

In case the destination exceeds the file size, the file is resized by adding a set of zeros sufficient to reach the specified position. This function has proper mechanism to avoid the user to jump into the header sector (which is forbidden in any way).

uint16_t secure_close(SEFILE_FHANDLE *hFile)

This function, given the file descriptor, simply closes the file without additional operations and deallocates all its relative resources.

uint16_t secure_truncate(SEFILE_FHANDLE *hFile, uint32_t size)

This function resizes the file to the specified number of bytes uint32_t size received as argument. It takes care of the sectors management and leave the file pointer to the end of the file (after the last byte of user data).

If the specified file is bigger than the original, sectors are filled with zeros, otherwise data in excess are lost.

The implemented functionality is shown in the following algorithm.

Algorithm 7 How a secure file is truncated

```
function SECURE_SEEK(in hFile, in size)
    Check Secure Environment
    Check if the file is not being tampered
    Compute file size using secure_getfilesize
    if size > file size then
        return secure_seek(hFile, size − file size, do not care, End of file)
    end if
    Compute what will be the last sector after the truncate
    Move file pointer to destination using OS system call
    Read, decrypt and verify sector
    Copy data that should be preserved after truncate
    Truncate file using OS system call
    Write back backup data using secure_write()
end function
```

uint16_t secure_ls(char *path, char *list, uint32_t *list_length)

This function is used to list the content of a directory containing encrypted files and/or directories. The function lists only those files and directories encrypted using the key ID stored in the secure environment, by returning the decrypted name of those and the total length in Bytes of this list.

uint16_t secure_mkdir(char *path)

uint16_t crypt_dirname(char *dirpath, char *encDirname, uint16_t *enc_len)

secure_mkdir masks the mkdir() function in the Unix environment and the CreateDirectory() function in Windows, but it does not implement the whole functionalities of those functions. Since directories are created using a wrapper to the OS system call, it is not possible to achieve a mechanism like the one employed for regular files, so it has been decided to use this encryption scheme, leveraging to crypt_dirname(), just for the directories name: the first 8 characters are the hexadecimal representation in ASCII of the key ID, and the rest is obtained using the AES-256-ECB. The total length of the encrypted name is always returned in enc_len.

The implemented functionality is shown in the following algorithm.

Algorithm 8 How a discover which encrypted file are present in a directory

```
function SECURE_LS(in path, out list, out list_length)
    Check Secure Environment
    do
        Navigate directory pointed by path
        if Object found is a directory then
            Try to decrypt directory name
            if Directory was made with SEfile then
                Add name to list and increment list_length
            end if
        else if Object found is a regular file then
            Open the file and try to decrypt the header
            if Header decrypted succesfully then
                Read the plain text name stored in header
                Append plain text name to list and increment list_length
            end if
        end if
    while All the elements of path have been browsed
end function
```

uint16_t secure_sync(SEFILE_FHANDLE *hFile)

This function is used in case it is needed to be sure that the OS buffers are correctly flushed to the physical file.

APPENDIX B - SE*link* APIs

This Section provides a brief overview about the **SE***link* APIs.

For more details about their implementation, please refer to the Doxygen-based documentation.

void SElink_buffer_init(SElink_buffer* buf)
This function initializes a buffer object to an empty buffer.

void SElink_buffer_ensure(SElink_buffer* buf, size_t newsize)
This function ensures the capacity for **SE***link* buffer.

void SElink_buffer_grow(SElink_buffer* buf, size_t required)
This function ensures capacity for **SE***link* buffer, which is expanded to the desired size (must be a power of 2).

void SElink_buffer_free(SElink_buffer* buf)
This function disposes of the **SE***link* buffer.

void SElink_raw_init(SElink_raw* sr)
This function initializes the **SE***link* raw context.

void SElink_raw_reserve_keys(SElink_raw* sr, size_t nkeys)
This function reserves the memory space for keys buffer.

void SElink_raw_reserve_algorithms(SElink_raw* sr, size_t nalgorithms)
This function reserves the memory space for algorithms buffer.

void SElink_raw_add_key(SElink_raw* sr, uint32_t id, const uint8_t* fingerprint)
This function adds a key to the **SE***link* raw context.

void SElink_raw_add_algorithm(SElink_raw* sr, uint32_t id, const uint8_t* fingerprint)
This function adds an algorithm to the **SE***link* raw context.

void SElink_raw_clear(SElink_raw* sr)
This function disposes of keys and algorithms buffers.

void SElink_raw_d1_write(SElink_raw* sr, size_t* len, uint8_t* data)
This function writes D1 packets.

bool SElink_raw_d1_get_size(SElink_raw* sr, size_t len, const uint8_t* data, size_t* size)
This function retrieves the size of D1 packets.

bool SElink_raw_d1_read(SElink_raw* sr, size_t len, const uint8_t* data)
This function reads D1 packets.

bool SElink_raw_d1_match(SElink_raw* sr, size_t len, const uint8_t* data)
This function finds a match in D1 packets.

void SElink_raw_d2_write(SElink_raw* sr, size_t* len, uint8_t* data)
This function writes D2 packets.

bool SElink_raw_d2_read(SElink_raw* sr, size_t len, const uint8_t* data)
This function reads D2 packets.

void SElink_raw_s1_write(SElink_raw* sr, size_t* len, uint8_t* data)
This function writes S1 packets.

bool SElink_raw_s1_read(SElink_raw* sr, size_t len, const uint8_t* data)
This function reads S1 packets.

bool SElink_raw_s1_match(SElink_raw* sr, size_t len, const uint8_t* data)
This function finds a match in S1 packets.

void SElink_raw_h_write(SElink_raw* sr, uint8_t* header, size_t data_size)
This function writes data packet headers.

void SElink_raw_h_read(SElink_raw* sr, const uint8_t* header, size_t* data_size)
This function reads data packet headers.

void SElink_raw_fingerprint(const uint8_t* salt, size_t len, const uint8_t* data, uint8_t* fingerprint)
This function generates raw fingerprints.

uint16_t SElink_raw_secube_import(SElink_raw* sr, se3_session* s, uint16_t key_size, uint16_t algo_type)
This function imports keys and algorithms from the device.

size_t SElink_raw_packet_size(size_t data_size)
This function retrieves the expected packet size.

void SElink_init(SElink* ctx, se3_session* s)
This function initializes the **SE***link* context.

void SElink_destroy(SElink* ctx)
This function destroys the **SE***link* context.

uint16_t SElink_duplex_write_request(SElink* ctx, SElink_buffer* buf)
This function writes duplex requests.

uint16_t SElink_duplex_get_request_size(SElink* ctx, SElink_buffer* buf, size_t* request_len)
This function retrieves the length of duplex request packets.

uint16_t SElink_duplex_reply(SElink* ctx, SElink_buffer* buf)
This function replies to duplex requests.

uint16_t SElink_duplex_read_response(SElink* ctx, SElink_buffer* buf)
This function reads duplex responses.

uint16_t SElink_simplex_write_invite(SElink* ctx, SElink_buffer* buf)
This function writes simplex invites.

uint16_t SElink_simplex_read_invite(SElink* ctx, SElink_buffer* buf)
This function reads simplex invites.

uint16_t SElink_write(SElink* ctx, size_t data_len, const uint8_t* data, SElink_buffer* buf)
This function writes data packets.

uint16_t SElink_read_header(SElink* ctx, size_t data_len, const uint8_t* data, size_t* remaining_bytes)
This function reads data packets' headers.

uint16_t SElink_read(SElink* ctx, size_t data_len, const uint8_t* data, SElink_buffer* buf)
This function reads data packets.