Politecnico di Torino

Cybersecurity for Embedded Systems (01UDNOV)

An advanced embedded Key Management System

Final Project Report

Master degree in Computer Engineering

Supervisors
Prof. Paolo Prinetto, Nicoló Maunero, Gianluca Roascio

Flavia Caforio (257750)
Samuele Yves Cerini (256813)
Sergio Mazzola (257926)

February 15, 2021

Contents

Introduction

It has been many years since the activities of individuals, organizations and nations have been growingly conducted in the cyberspace [1]. This has been the consequence of a worldwide digitization and widespread of electronic devices, pushing information systems as mediators and controllers of such activities.

The word *cyberspace* has been coined to define the complex resulting from the interaction of people, software and services on the Internet by means of technologies, devices and networks connected to it [2]; the cyberspace itself represents a significantly intricate system, due to it being based on the ensemble of thousands of networks, its stratified nature of softwares and protocols and its heterogeneity in terms of users and devices.

For this reason, along with the huge benefits it determined in terms of communication, productivity, storage of data, it nonetheless posed a new set of vulnerabilities, creating potentials for hostile actions in the cyberspace and resulting in threats to either individual, societal or national interests.

As a result of the research and the action against such brand new threats, the *cybersecurity* field emerged: cybersecurity defines the practice that allows an entity to protect its physical assets and confidentiality, integrity and availability of its information from the threats coming from cyberspace [2]. One of the main concerns of cybersecurity, nowadays very much felt both in the private [3] and public [4] realms, is the protection of privacy and data security, which led to large investments by governments and companies in the field of *cryptography*.

Cryptography is a method for protecting information transmitted through potentially hostile environments by making it non-intelligible to every party not in possession of a *key* [5]. Damages resulting from message alteration, message insertion and message deletion can be avoided too. There are two approaches to cryptography, symmetric cryptography and asymmetric cryptography, mainly discriminated by the way the keys are generated, distributed and used to encrypt the messages, with specific pros and cons.

However, a common term of any practical implementation of whichever cryptographic approach is the need of a secure *Key Management System* (KMS). Key management refers to managing cryptographic keys within a cryptosystem: it deals with generating, exchanging, storing, using and replacing keys as needed at the user level. A flawless and secure key management is the very basis for the security of the entire cryptosystem [6] [7].

The aim of this project has been to study and analyze the concepts mentioned above by implementing from scratch an actual cryptosystem deployed on an open-source security platform, SEcube™ [8]. SEcube™ is a single-chip design that embeds three main cores: an ARM Cortex-M4 `STM32F4` low-power processor, a Common Criteria certified smartcard and a flexible FPGA. The platform has been chosen as an example of device with many security features, both at a physical level and at a software level, to use as a basis for the designed cryptosystem, and also for its affinity to the IoT and embedded systems context, which very much influenced many design choices.

The baseline for this project has been the framework developed during the previous laboratories [9] [10]: the cryptosystem has been designed from scratch to run on the `STM32F429ZI` core; only the bare-metal drivers of the core have been used, without the SEcube™ API. The system is based on a Key Management System able to generate, store, delete and use keys in a secure way, without never disclosing any information about them. Along with all the key management functionalities, the cryptosystem is also provided with two example cryptographic applications which make use of the KMS: the AES-256 encoding algorithm and the HMAC-SHA256 cryptographic hashing algorithm. The device-side project has been implemented on an emulated version of the processor core executed on QEMU [11]. The whole project is based on several custom drivers (e.g. the communication framework to exchange packets with the host, the interface towards the non-volatile memory) which are currently implemented for QEMU, but can be easily adapted to switch to an actual physical board or to another platform thanks to the fully layered nature of the design.

As a part of the project, the software deployed on the device comes with an host-side library, which offers an API to send commands to the cryptographic device and receive its replies as a black box.

The main contribution of the final project described in this report has been to improve the already implemented KMS extending it with the advanced functionalities described in the following:

- extension of the key state field of the keys stored in the KMS database to implement the **management of the keys life cycle**; with this feature, each key has a state determining which activities can be carried out with it (encryption, decryption, signature computation, signature checking); the state of a key can be changed by the host, basing on its decisions (e.g. compromised key due to a leak), or due to the expiration of its cryptoperiod;
- implementation of a Key Agreement Protocol (KAP) for symmetric keys built upon a **Password-based Encrypted Key Exchange** (EKE) based on Exponential Key Exchange; the implemented protocol is used to establish a specific communication flow, orchestrated by the host, between two devices which need to agree on the same key to be used for symmetric encryption purposes; the commonly agreed key is stored in the KMS as any other manually-added key. The KAP library functionalities are able to generate commonly agreed keys with a size of up to 256 bits; the KAP is based on an arithmetical library implementing several number theory functionalities, all based on 32-bit data structures. For the complete functioning of the protocol, a library has also been developed for a custom random number generator driver, exploiting the random data in `/dev/urandom` of the Linux environment;
- extension of the project framework to be able to easily emulate **multiple devices** at the same time and let the host communicate with all of them, basing on its choice;
- extension of the **host-side API** to include the interface towards the new device functionalities to manage the keys life cycle and the Key Agreement Protocol.

The rest of this report is organized as in the following: Section 1 summarizes the main project functionalities and all of its components; Section 2 describes in depth the implementation of the keys life cycle management and how it has been integrated in the already present communication framework and KMS environment; Section 3 describes the same with respect to the newly-introduced Key Agreement Protocol. Finally, Appendix A provides the reader with a documentation of all the functions offered by the host-side API to communicate with the device and exploit its functionalities, while Appendix B outlines the hierarchy of the main project directories and files.

1 Project features

This section summarizes all the features of the project and outlines its architecture and its main components. Several functionalities listed in the following have been already developed during the previous laboratories, so refer to [9] and [10] for deeper details and implementation choices; for the newly introduced features refer instead to Section 2 and Section 3.

The main aim of this project has been to develop a bare-metal cryptosystem for the ARM Cortex-M4-based `STM32F429ZI` MCU; the target platform implementing such cryptosystem, referred in the following as *device*, is generally a low-power embedded device, able to receive commands from a master entity, hereinafter named *host*, and perform the requested operations, communicating back its outcome. The whole project has been built around the following main specifications:

- *security* – apart from the main specifications deriving from the choice of designing a cryptosystem, many other details have been always kept into account to avoid any security flaw: a precise return value standardization for function calls has been established, allowing to trace back to its origin every kind of error or warning occurred during runtime; any stale data from old commands or communications (e.g. old keys in the KMS, old buffers) is wiped; no critical information is allowed to leave the boundaries of the secure device, only very controlled final outcomes are sent back to the host;

- *scalability* – the whole project is extremely layered and device functionalities are generally built on top of APIs towards lower-level drivers, which are the only portion of code which must be adapted when switching to another platform; additionally, the project is very much scalable also from the point of view of the functionalities: everything from the communication framework to the Key Management System has standardized code and data structures easy to understand and integrate with new functionalities;

- *embedded system* – the target platform of the device-side project is a low-power embedded device, with all the hardware and software restrictions of the case; this has also shaped many implementation choices, such has the absence of the use of dynamic memory allocation and the tendency to use more optimized algorithms.

The project, mainly divided into a device-side project and an host-side project, comes with an automatic environment in the form of a `Makefile`, to easily set up the run configuration and execute both the device and the host softwares, establishing the communication among the parties. Multiple devices can be run at the same time, with the host being able to select to which device to send the desired command.

This project also comes with an example wrapper for the host-side API, which allows to test out-of-the-box all the available device functionalities and learn how to thoroughly use them through the offered API. For more information about how to install and run the project, see the `README.md` file in the project root, as in Appendix B.

1.1 Device-side

The backbone of the cryptosystem is a **Key Management System** holding a database able to store a number of keys, along with their metadata, current state and cryptoperiod; the host can generate, store, delete, manage and use keys in a secure way, with them never leaving the secure boundaries of the device. The device is also provided with a **Key Agreement Protocol** integrated in the KMS, which allows a couple of devices communicating by means of the host to agree upon the same shared key.

On top of such key management, two **cryptographic applications** are built to offer to the host some example cryptographic functionalities: the device is indeed able to encrypt and decrypt messages through the AES-256 algorithm and to hash messages or verify their signature with the HMAC-SHA256 algorithm; both functionalities exploit the keys stored in the KMS.

The STM32F429ZI MCU platform on which the device-side project has been deployed is emulated by means of QEMU [11], and the main **low-level drivers** (communication, non-volatile memory, random number generator, timer) have been implemented for this particular scenario, exploiting the functionalities provided by the ARM Semihosting mechanism [12]. Each component, library and driver is anyway extremely layered and designed to be as scalable as possible, so that the whole system can be switched without many difficulties to another platform, including a physical one, by only adapting the lowest-level functions.

In the following, all the developed device functionalities are presented.

1.1.1 Communication framework

The communication framework of the device is used to exchange messages with the host in the form of packets with a pre-defined structure, shown in Figure 1. The device constantly checks the input channel waiting for a new command and, as soon as a new incoming packet is present, the device reads it and executes the requested command, writing the response on the output channel. Apart from the header fields, which are filled during the customization of the command or of the output response, the maximum size of the packets payloads has been set to 7600 bytes, to reflect the limits of the physical USB buffers of the SEcube™ platform.

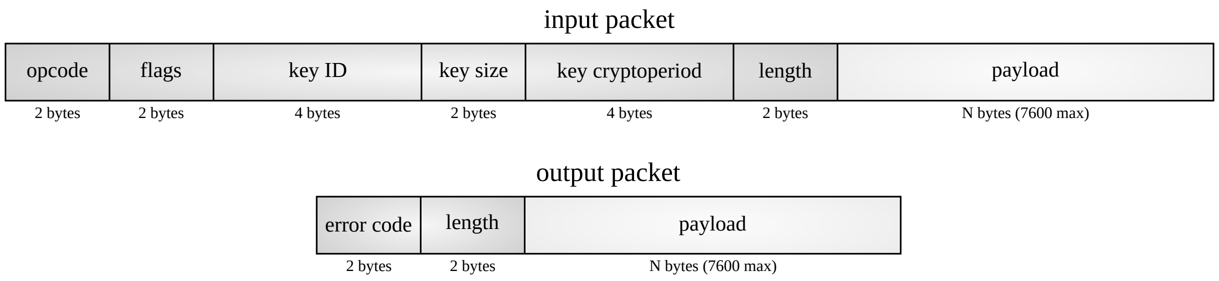

Figure 1: Communication packets structures; input packets are written by the host on the input channel and read by the device, output packets are written by the device on the output channel and read by the host

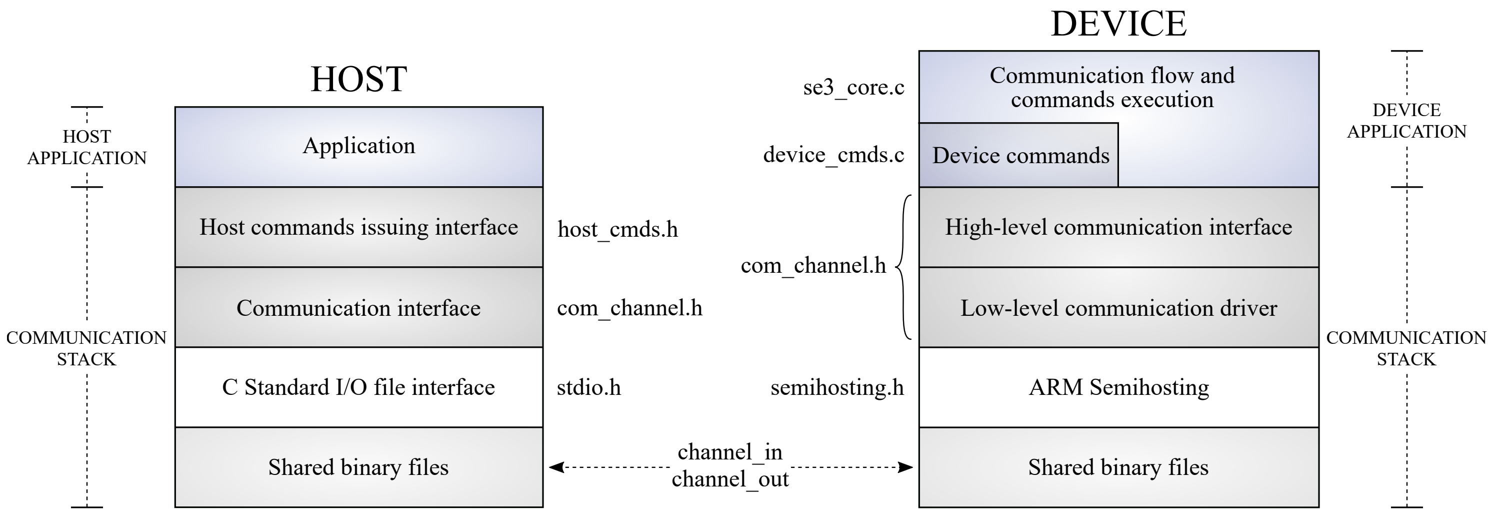

Figure 2: Host-device communication stack.

The communication library com_channel, whose structure is outlined in Figure 2, is mainly composed of three sections: the *low-level drivers*, which interface with ARM Semihosting functionalities reading and writing the binary files shared with the host software; the *high-level interface* composed of several functions accessible by device applications to use the communication functionalities from an higher abstraction level; the *communication flow* implemented in the device main loop, which actually accesses the communication mean to check for the presence of a new command and potentially send an output response. The actual physical communication mean is virtualized by the shared files `channel_in` and `channel_out`.

The communication flow also includes the decoding of the incoming commands from the host and their execution by means of the functions of the `device_cmds` library. See [9] and [10] for further details about the communication framework, including the exact steps of the protocol.

1.1.2 Non-volatile memory

The device is provided with a non-volatile memory driver for those functionalities needing a permanent storage. Due to the device being emulated with QEMU, such driver is currently implemented by means of the ARM Semihosting mechanism exploiting a private binary file.

As the communication library, the interface towards the non-volatile memory, implemented in the library `nv_mem`, is layered and flexible, being composed of: the *low-level drivers*, which interface with ARM Semihosting and the *high-level interface* including several functions to easily access the non-volatile memory independently from drivers implementation.

The non-volatile storage implemented for the device has a size of 1 MB and is byte-addressable. For the sake of simplicity, a static memory address range is dedicated to each one of the applications exploiting the non-volatile memory, as it is usually done with main memory; also, it is duty of the application itself to check that its addresses are in the correct range.

In Table 1, the memory map currently implemented in the system. For more detailed information about the non-volatile memory, see [10].

Table 1: Memory map of the non-volatile memory.

	Base address	Size [bytes]
KMS keys number register	0x00000000	4
KMS database	0x00000004	136000

1.1.3 Random number generator

For the new functionalities needed by the Number theory library and the Key Agreement Protocol a random number generator has been needed. The random number generator library, `rng_custom`, is divided in two main layers: the *low-level driver* of the physical source of randomness, which can be easily adapted to any platform, and an *high-level interface* to let applications request random data in an abstract way.

The actual source of randomness currently employed by the emulated system is the `/dev/urandom` file of the Linux environment, accessed by means of ARM Semihosting functionalities. The high-level interface of the library consists of a function exploiting the low-level drivers to collect 4 random bytes and assemble them into a random 32-bit unsigned integer.

Note that a built-in driver for the random number generator, the library `rng`, is already implemented in the original device-side project; however, some problems arose due to the microcontroller being only emulated on QEMU and not physically run, which is why a custom library for a virtual random number generator has been developed. The original `rng` library might be put in use when switching to a physical platform.

1.1.4 Key Management System

The Key Management System of the device is the backbone of all cryptosystem functionalities: every other cryptographic application depends on it, as well as the security of the whole cryptosystem. The

KMS offers several key management functionalities to the host, including the possibility to generate and add new keys, remove keys by completely wiping their content and metadata, update their cryptoperiod and re-seed their content, change their state. Device cryptographic applications can request keys to the KMS; before returning a key, the KMS checks its state to guarantee it can be safely used for the requested application.

For security reasons, when generating a new key, the host cannot directly select a plain text to use for this purpose: it can only provide the device with a seed which is hashed by means of HMAC-SHA256 algorithm; the digest is the key, which is thus secret and never leaves the secure device boundaries.

The KMS database, stored in the non-volatile memory, is organized as an array of key records, each one with a fixed size of 144 byte and composed of several fields to keep track of the key ID, size and state. The database is kept in a non-volatile memory region considered safe, as the keys are stored only in an encrypted form. The maximum size of a key which can be held in the KMS database is 1024 bits.

The KMS does not disclose any information about the keys it holds; the only information which can be retrieved is the list of the keys IDs in the database. For this reason, it is duty of the host to keep track of keys metadata: it must remember their ID, their size, their state, their expiration time and what they are used for. For implementation details about the KMS, see [10].

As a newly introduced feature, the KMS supports an advanced key state management consisting of several states (pre-active, active, suspended, deactivated, compromised, destroyed) determining which operations can or cannot be performed with the key (encryption, decryption, hashing, signature checking). The current state of a key depends both on its expiration time, determined by its cryptoperiod, which is selected by the host at the creation time of the key, and on the manual state changes that the host can request. Further details about this feature are given in Section 2.

The KMS library is `kms`, and it is based on the non-volatile memory and on the time handler drivers; it does not depend on their actual implementations but it only exploits their high-level interface to read and write the permanent storage and retrieve the current time for cryptoperiod operations.

1.1.5 Time handler

The Key Management System library relies on the `time_handler` library to provide the functionalities related to the automatic temporal key management, which in turn allow to enforce a mandatory key update, limiting the possible negative outcomes in case of key disclosure. At key creation time, the user must indeed provide the KMS with a desired *cryptoperiod* (the relative life duration of the key, expressed in seconds), used to compute the *expiration time* (the actual timestamp specifying from when on the key is to be considered expired) of the key itself.

For the sake of simplicity and consistency, both `cryptoperiod` and `expire_time` variables follow the UNIX epoch standard: both time variables are expressed in seconds on `uint32_t` data type, having `expire_time` computed only upon a request of key activation from the host. The expiration time corresponds to the sum between the cryptoperiod (specified when creating the key) and the current time (expressed in seconds, from the January 1, 1970 date, as specified by the UNIX epoch standard). The time handler library serves exactly this purpose as it allows the device to retrieve the current time as UNIX epoch. The implemented library offers a single interface function `get_time()`, whose internal implementation can be adapted to reflect the actual hardware-dependent time retrieval operation.

Generally a device, being deployed on a microcontroller, can indeed leverage the presence of an internal RTC peripheral to retrieve the actual time. The `get_time()` function hence acts as an implementation-independent interface, masking the actual hardware or software module used to retrieve the current time. This allows to increase the flexibility of the system as only the low-level implementation of the function needs to be changed in case of platform switch, without changing other aspects of the device application . In our implementation, with the device emulated using `QEMU`, the driver used to retrieve the current time consists of a simple function call to ARM Semihosting, which in turn interfaces with the timer of the host machine.

1.1.6 Number theory library

With one of the aims of the project being to implement a Key Agreement Protocol based on Exponential Key Exchange, an arithmetical library implementing several operations from Number theory has been needed.

The library, called `numth_arith`, contains many functions from Number theory implemented over 32-bit unsigned integers; such functions include operations in modulus, on prime numbers and with discrete logarithms, particularly useful for random primes generation and key exponentiation. While the choice of using only 32-bit data structures represent a serious risk for the robustness of the cryptographic functionalities exploiting such library [13], this has to be considered a fully working proof-of-concept, easily extensible to larger and actually secure data structures, which were not used here for complexity reasons.

The arithmetical operations implemented in the described library are listed in the following:

- *modular multiplication*: given a, b and m unsigned 32-bit integers, computes $a \cdot b \bmod m$ by means of an algorithm based on Russian peasant multiplication [14]; the algorithm is adapted to solve the problem to wrap around m, but also to avoid incorrect results due to the intrinsic wrapping around `UINT32_MAX`;
- *modular exponentiation*: given a, b and m unsigned 32-bit integers, computes $a^b \bmod m$, basing on the algorithm in [15];
- *fast primality test*: basing on the algorithm described in [16], this function performs a fast primality test for 32-bit unsigned integers exploiting a statically allocated hash table of 512 bytes;
- *factorization*: factorizes the input 32-bit unsigned integer n and returns the list of factors; due to dynamic memory allocation not being feasible, and not knowing a-priori the number of factors of n, the function also takes as input a maximum array size and, if the number of factors overcomes it, an hard fault is returned;
- *primitive root computation*: given a 32-bit prime number p, this function, based on the algorithm in [17], returns one of its primitive roots; in particular, it computes 16 among the biggest primitive roots of p and randomly selects one to return; primitive roots computation requires the factorization of the Euler's totient of p;

1.1.7 Key Agreement Protocol

Along with the possibility of manually adding a key to individually request cryptographic functionalities to a device, this project introduced a Key agreement protocol (KAP) to let two devices agree on the same shared key of up to 32 bytes, empowering them to set up an encrypted communication channel to exchange confidential messages.

The KAP, implemented in the library `eke_exp`, is a Password-based Encrypted Key Exchange with Exponential Key Exchange based on [18], selected to overcome the security criticalities of the Diffie-Hellman Key Exchange.

Due to the needs of a KAP, this project also introduced the possibility of running multiple devices at the same time, with the host being able to choose to which one to talk. The KAP is indeed implemented taking into account the fact that the devices might be distributed and not able to talk to each other, so that the host is the orchestrator of the protocol flow and the one making the devices interact. The Key agreement protocol is fully described in Section 3, along with all of its features and its implementation details.

1.1.8 AES-256 encryption & decryption

The device implements a cryptographic encryption and decryption compliant to the AES-256 standard [19] in the library `aes256`. The Advanced Encryption Standard (AES) is a standard cryptographic algorithm used to protect information, making it non-intelligible to entities not possessing the key.

The algorithm uses 256-bit keys and it is based on a symmetrical cipher block that can both encrypt and decrypt; the cipher works on 16 bytes blocks (4 words of 4 bytes each); thus, when the plaintext size is not a multiple of 16 bytes, some padding is needed, implemented here with zeros. The output size of the implemented AES-256 algorithm is always a multiple of 16 bytes, with the minimum size being 16 bytes.

Two operation mode can be selected for the encryption and decryption of messages: Electronic Codebook (ECB) mode and Cipher Block Chaining (CBC) mode, with the second being stronger but more expensive. More details about AES-256 and its implementation in the device are given in [9].

1.1.9 HMAC-SHA256 hashing & signature checking

An HMAC (hash-based message authentication code) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. As with any MAC, it may be used to simultaneously verify both the data integrity and the authenticity of a message. HMAC algorithms do not encrypt the message; instead, the message (encrypted or not) must be sent alongside the HMAC hash: parties with the secret key will hash the message again themselves and, if it is authentic, the received and computed hashes will match [20].

The device implements the HMAC-SHA256 algorithm, an HMAC algorithm based on the SHA-256 cryptographic hash function, and it is able to both generate digests of input messages and check the integrity of a message by comparing its digest to the received signature; HMAC-SHA256 has a fixed output size of 32 bytes [21].

SHA-256 is part of the Secure Hash Algorithm - 2 (SHA-2) set of cryptographic hash functions designed by the United States National Security Agency (NSA) [22]. SHA-256 takes as input an arbitrary long message and produces as output a 256-bit long digest signature; it is implemented by the library `sha256`. On top of the SHA-256 algorithm is built the `hmac_sha256` library, implementing HMAC-SHA256. While HMAC-SHA256 accepts as input an long messages and keys, for this implementation their maximum length is limited to 7600 bytes, the maximum length of the input packets payload set by the communication framework. More details about SHA-256 and HMAC-SHA256 are available in [9].

1.2 Host-side

The host-side project essentially consists of an API library meant to be used by a general-purpose machine to interface to all the functionalities implemented on the device. The host-side API, entirely documented in Appendix A, makes possible the use of a device as a black box, only providing the specified input data and output data structures for each command.

As already mentioned, the host-side project also includes an example wrapper for the host API able to thoroughly play with all of the functionalities offered by the device. The designed software provides the user with a textual interface to select which command to request to the device, among all the available ones, and to which of the running devices to send it; the command can then be customized basing on the needed fields (e.g. commands flags, ID of the key to use if cryptographic functionalities are requested) and the payload can be entered (e.g. an hexadecimal string to decrypt, a plaintext message to encrypt, a seed to generate a key).

1.2.1 Communication framework

As in Figure 2, the host interface through the communication channel with the device is implemented by the `com_channel` library. The library is based on the C Standard I/O file interface, which could be considered the actual driver of the communication mean, and implements several high-level functions to let commands API access the communication framework in an implementation-independent way.

The C Standard I/O file interface is used to read and write the binary files shared with the device as communication mean, by means of ARM Semihosting. Should the system switch to a physical platform, only the functions in `com_channel` would need to be adapted to the new driver.

1.2.2 Commands API

The host-side library `host_cmds` is built on top of the host-side communication interface and provides the host applications with an API towards all of the device functionalities. The API is fully documented in Appendix A.

2 Key Management System

The Key Management System developed in this project is heavily based upon the KMS already implemented in [10]. Therefore, many of the functionalities have been kept unchanged or just slightly modified to accommodate the new ones. This has been possible without major hassle thanks to the general organization of the main implementation, whose architecture was designed to be as modular and scalable as possible, to easily extend it.

In the following, a brief review of what has already been done is presented, introducing the additional minor modifications while particularly focusing on the main features and implementation details of the new functionalities introduced with this project.

2.1 Specifications overview

As already mentioned, many additional requirements have been integrated in the already existing system developed in [10]. With respect to our previous implementation [10], the major addition consists in the management of keys lifetime: a complete KMS like the one implemented with this project provides the required foundation for a secure generation, storage, distribution, use and destruction of keys, as it allows to manage the entire key lifecycle, from its creation to its destruction. Every state and all of the transitions implemented among key states are compliant to the SEcube™ software documentation [23] and its related state diagram (Figure 3)

Therefore, for this project, key fields are not only *empty* or *valid* anymore. Instead, there are now multiple reachable states for each valid key, that can limit the field of applications in which it can be used. The presence of these additional states allows the host to manage each key present in the KMS database with an improved granularity, limiting their exposure and reducing the possible consequences in case of device tampering and key exposure.

According to the given requirements, a key can assume the following states:

- `Pre-Active` – this is the default state for every new key, one of the most restrictive ones, as the key cannot be used to encrypt/decrypt nor creating/checking HMAC signatures. At this step, the key's cryptoperiod has been defined but it will not be considered. This state, once left by the key, cannot be reached anymore;

- `Active` – in this state the key can be used to perform all the possible available operations (encryption/decryption and HMAC signature creation/check). The activation of a key is specifically requested by the host (`cmd_activate_key()`). Upon activation, the cryptoperiod associated to the key is used to compute the actual expiration time: whenever the current time reaches the expiration timestamp, the key is automatically moved to the `deactivated` state;

- `Suspended` – the host can request to suspend an active key, limiting its applications only to AES decryption and HMAC signatures check. In this state the key can still expire and be automatically moved to the `deactivated` state (or manually deactivated by the host with `cdm_deactivate_key()`). The key can be moved back to the `active` state as long as it is not expired, upon specific host request (`cmd_activate_key()`);

- `Deactivated` – a key is put into the `deactivated` state whenever its expire time has passed or upon specific request from the host (`cmd_deactivate_key()`). Further implementation details are given in Section 2.4. In this state, a key can only be used for decryption or HMAC signature check;

- `Compromised` – as for a `deactivated` key, encryption and HMAC signature creation are not possible, but the key can still be used to decrypt or check the HMAC signature of old files. For security reasons, the host can request to put a key in the `compromised` state at a any time, without restrictions. In our specific implementation, a key is also automatically set to this state as a precaution whenever a key is found to reside in an unknown state or whenever tampering or glitching activity is suspected to be going on;

- `Destroyed` – in this state only the metadata associated to the key is kept in the memory, hence not allowing for any operation: the data that is left encrypted with a destroyed key cannot be recovered anymore. For security reasons, the host can request the deactivation of a key at any time and without restrictions. Note that a key in the destroyed state is still stored encrypted in the KMS database, even if not usable anymore; the complete deletion of a key content and its metadata happens only if specifically requested by the host with `cmd_remove_key()`.

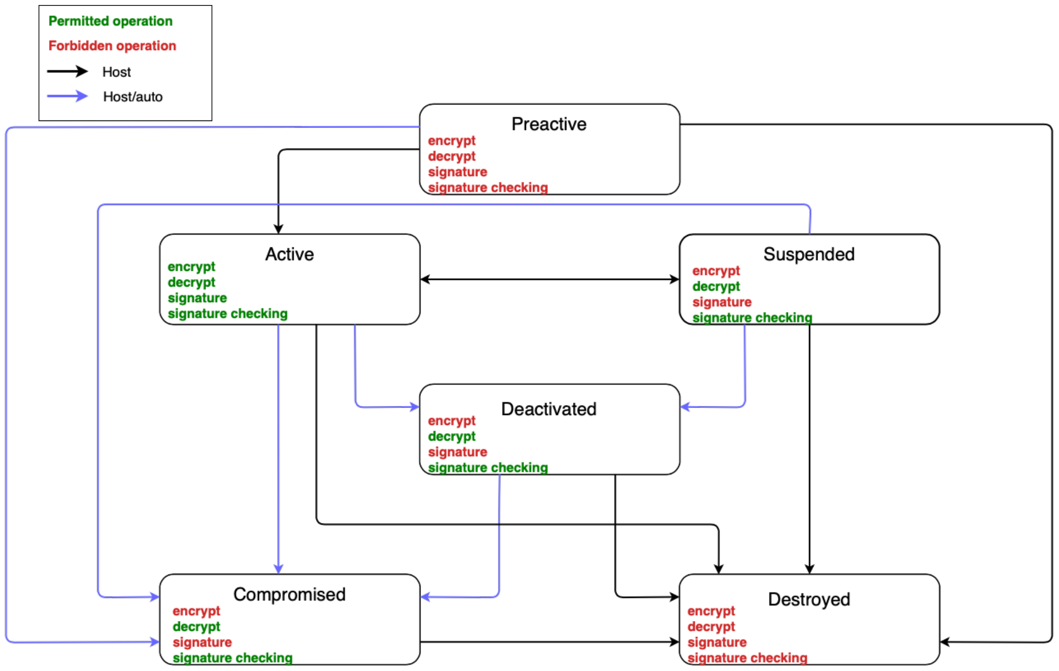

Figure 3: Key state transition diagram: the black arrows denote transitions that are specifically requested by the host. The blue arrows denote transitions that can be both requested by the host or taken automatically by the device upon some triggering condition. For each state, permitted and forbidden operations are also highlighted.

2.2 Features

The KMS main data structure has been kept, in its core functionality, unchanged. This means that the data structure representing each key in the non-volatile memory follows the same rationale described in the previous version of the implementation: the key database is still organized as an array of `struct kms_key` records, where each record occupies a fixed memory space (to avoid complex problems such as fragmentation of memory space and different-size-entries scanning). The maximum size of each key is still of 128 bytes (i.e. 1024 bits), with each database record taking 144 bytes (key content and metadata fields). This implementation adds two additional fields to the key structure:

- `cryptoperiod` – an unsigned integer variable on 4 bytes, representing the cryptoperiod of the key (i.e. its lifetime), expressed in seconds (UNIX epoch). Among many other advantages, the use of a cryptoperiod allows to limit the time period during which information may be compromised by inadvertent disclosure of keys to unauthorized entities;

- `expire_time` – an unsigned integer variable on 4 bytes, representing the expiration date of the key (i.e. `activation_time` + `cryptoperiod`), expressed in seconds (UNIX epoch);

The field `state`, although similar to the one used in the previous version, now has to accommodate an increased number of key states, following the given requirements. The final key structure is illustrated in Figure 4.

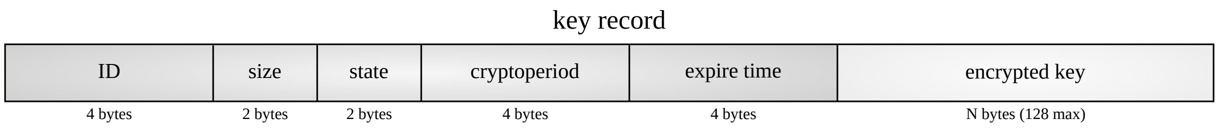

Figure 4: Fields of a key record, representing each key with its metadata, in the KMS database.

The main functionalities, previously implemented and described in [10], include: KMS initialization, new key addition, update of an existing key, removal of an existing key, key retrieval and listing of all valid keys.

Some additions and modifications have been made to adapt the host API to the new requests: now both the `cmd_add_key()` and the `cmd_update_key()` operations require to specify a cryptoperiod to be linked to the key itself. We will properly illustrate the new implementations in Section 2.4.

2.3 Security details

2.3.1 Threat model definition

Understanding the threats that the designed system could be subjected to is a starting point for deciding on what countermeasures are needed to protect it. We, as designers, have to imagine a set of possible scenarios the system could end with, concentrating on the ones that can lead to problematic outcomes and unexpected behaviors. In the case of a KMS, similarly to the case of a physical safe, the most problematic outcome is the one in which the physical memory content (i.e the keys) is disclosed to the external environment, as a consequence of a malfunctioning or a malicious attack [24]. In fact, disclosed keys can lead to a succession of potentially catastrophic consequences, starting from the now-unencrypted communications that the unaware user of the system is going to inevitably experience.

In the specific case of this project, it is meaningful to assume that a KMS, being such a central and crucial component in a security-oriented embedded system, will be also the main target and interest of a malicious attacker. Thus, a correct implementation must consider security (and specifically, non-disclosure of private information) as a top priority and design parameter since the early stages of development.

From a point of view of the causes of these unwanted outcomes, the designers must take into account all the possible scenarios the device may encounter in its entire lifetime. Specifically, given the generic nature of an embedded system (small dimensions, extreme portability, possibly remotely connected), it is safe to consider all the cases in which the system could be subjected to external physical attacks or tamper attempts. Being the main issue defined, the designer proceeds in the definition of the threat model, possibly researching the generic *attacker persona* and the tools used to carry certain attack procedures.

Considering, as an example use-case of the employed platform, the existing products already developed using SEcube™ [25] [26], it is not unlikely that the device may fall into malicious hands, for example by being left unattended by the customer for a certain period of time or by directly being stolen. The worst possible attack is the second one as the attacker has full access to the device, with nearly-unlimited time to understand it, disassemble it and to attach external (and ideally professional) equipment to it, with the final goal of retrieving the content of the memory.

Hence, given the extremely good conditions the attacker may leverage, both classes of non-invasive and invasive attacks are still achievable. For the non-invasive attacks we imagined the attacker being in possession of specialized hardware to accomplish side-channel attacks, such as a *ChipWhisperer* board [27] (which costs, at the time of writing this report, as low as 250$) and a commercial-grade laptop. Such relatively inexpensive setup can be used to analyze the power consumption of the device, implementing the many existing techniques (for instance, see [28] and [29]). *ChipWhisperer*, despite being a relatively new product, has helped democratizing even more the exploitation of side-channel attacks (like power analysis and timing attacks) and fault-based attacks (like power and clock glitching). Some real world demonstrations on products and devices similar to the SEcube™ USB Token [25] (like cryptocurrrency hardware wallets) have already been presented in hacker conferences (see [30] and [31]) and even used in CTFs competitions [32].

It is therefore more than reasonable to bring this kind of product and attack methodology into our consideration while searching for a threat model for our KMS implementation.

Additionally, assuming a best case scenario where the implementation is hardened as much as possible against the aforementioned threat model (but still not *unbreakable*) the careful designer must also take into account the critical case where some of the countermeasures for the above problem have been circumvented. In other words, error prevention mechanism may not suffice, due to both designer inexperience (or bugs in the implementation) or to future improvements on attack methodologies and tools used to carry those that were unknown or unavailable at the time the system was developed. Therefore, when designing such critical components, it is crucial to consider, in parallel with error prevention mechanism, also error detection and error tolerance mechanisms.

2.3.2 Secure implementation

In this section, the specific implementation details adopted to overcome the threat model and attacks observations are discussed.

The given requirements and specifications discussed in Section 2.1 are already helping forging a system with the aforementioned error detection and error tolerance mechanisms. In fact, the presence in the KMS state diagram of the `destroyed` and the `compromised` states anticipates (and solves) the possibility of a negative outcome due to a system internal error or an external attack, by de facto implementing an error detection and recovery mechanism. Both these states are enforceable at any time from the host in case some malicious activity is perceived by the user (or for pure precautionary measures). The `compromised` state can be also enforced directly by the device (we will elaborate later on what events can trigger this scenario), preventing any possible exploitable delay between the sending of the error message (*device* side), its reception (*host* side) and its comprehension (*user* side). Indeed, this automatic mechanism and the timing advantage it brings can preserve the integrity, authenticity and confidentiality of user communications.

Additional precautions have also been taken already in the previous implementation of the KMS by deciding to not to store private keys in the memory of the device, which are indeed replaced by a collection of encrypted hashes. This extra step, as explained in the related report [10], requires the host, when asking the device to create a new key, to send a seed to it (which is assumed to be generated using a random generator in order to avoid sending multiple times the same seed). The device, once the seed is received, firstly computes its hash, it encrypts it and finally stores it in the non-volatile memory. This key encryption mechanism allows to safeguard the seeds from use even in the unfortunate event of an attacker being able to disclose the encrypted hashes. As a matter of fact, the attacker would not be able to decrypt them back given the lack of the encryption key (supposed to be unique, hence limiting its range of applicability to a single device being produced). In this scenario, we suppose the attacker being unable to retrieve the device-related encryption key: in the remote event this key is disclosed the attacker would be able to decrypt the encrypted hashes, fulfilling its initial intent and potentially voiding the authenticity, integrity and confidentiality of the main user communications. To avoid this remote scenario, future updates to the KMS may enforce the use of the EAL5+ certified Smart Card embedded in the SEcube™ chip just to store this device-unique encryption key, providing an additional level of security. On the other hand, even in the aforementioned worst case scenario (disclosure of both encrypted hashes and encryption key), the attacker would be unable to retrieve the seeds from their hashes, due to the very nature of the SHA-245 hash function used. We considered the above implementation hardened enough against attacks involving a complete dump of the Flash memory: although not perfect and still improvable, these additional precautions may stop or delay a successful outcome of an attack, giving the user the time to recreate a completely new keyring and to start a new series of communications the attacker cannot read unless a new attack on the new device is carried on.

Following the same principle of complicating as much as possible the efforts of the attacker (and limiting the quality of the results obtained), the use of a cryptoperiod as required by the main specifications fulfills exactly this kind of goal. By reducing the temporal attack surface of the single key, a good KMS enforces the periodic substitution of a certain key value, allowing to limit the possible negative outcomes in case of disclosure of the secret. Even in the worst case scenario where the user is unaware that the secrets have been disclosed by an attacker, the lifetime (cryptoperiod) associated to each single key mandatorily requires the user to change the key after a certain time, allowing the user itself to start a new encrypted communication after a reduced amount of time from the attack and disclosure time instant. For instance, a cryptoperiod of 24 hours limits the use of the key itself to 24 hours: after the expiration of this lifetime a new key must be produced by the user, hence limiting to 24 hours (in the worst case scenario for the user) the possible use of that key by an attacker. This clearly reduces the amount of documents the attacker may happen to exchange using the disclosed old key, possibly vanishing all the efforts spent to obtain it.

By recalling the attacker's persona definition detailed in Section 2.3, we now present the additional checks the device makes in order to avoid the most common low-cost glitching attacks (supply voltage glitches, clock glitches), as classified in [33]. Glitching attacks on real devices have demonstrated to be feasible even with extremely economical hardware [34] [35] [36]. Often used to skip password checks or to glitch read-write protection mechanisms of microcontrollers, glitch attacks can affect the return values of `C` functions. If an attacker is able to find the correct moment in time a certain function returns a specific value (by observing the power consumption traces, for instance), it can force a glitch in the power supply or in the clock lines to modify the register content being handled (containing the return value). If the function in use greatly counts on the returned variable to determine the success (or failure) of a certain operation, a glitch in the right moment able to change the value to an unexpected (or uncontrolled) one can lead to catastrophic outcomes.

In detail, if an attacker is able to affect the return values of the `C` functions used to confirm (or negate) a transition affecting the key current state, the key itself may reach a not-to-be-reached KMS state that, in the worst case, brings the key in a more privileged status. As an example, an attacker may attempt to glitch the state of a key, bringing it from the `destroyed` state (where no operation is possible) to the `active` state (where all operations are possible), de facto enabling it again for operations that were no more possible. To mitigate the possibility of this scenario we hardened the return value catch mechanism, avoiding leaving possible return codes being not checked, basically evaluating all possible `if-then-else` branches, without leaving any dangling control. For instance, let us assume a callee function returns to the caller the following possible values:

- `1` – indicating `success`;
- `0` – indicating a `warning` (handled gracefully);
- `-1` – indicating an `error` (hard fault);

Although the return values being unambiguous, a poorly written `if-then-else` error-catching clause in the caller function may simplify the check of the `success` case to a mere `if (ret > 0)`, i.e. *if any possible positive value*). This approximation is exactly what may lead the attacker to a successful glitching attack: by being able to glitch the value of the register used to store the return variable, the attacker has a huge range of possible positive values the register content may assume after the glitch: assuming the register content being on 32-bits signed (`int`), a glitch may enforce any positive value in the range between 0 and $2^{31} - 1$. In other words, the attacker as a high probability of being able to change the return value into a positive one, changing the original operation outcome and glitching the key state. On the other hand, a correct `if then else` clause checks for the exact return value (e.g. `if (ret == 1)`): this means that, to be successful, an attacker must glitch the device and enforce exactly the value `1`, a much more difficult (if not statistically unfeasible) case to carry on. Additional countermeasures may require to check specific (and more complex) return values, using special hexadecimal constants that are considered to be difficult to enforce due to their specific bit sequence (on a 8-bit architecture, `0xAA` and `0x55` could be an example). In case this kind of activity is detected by the system the key is automatically moved to the `compromised` state, de facto implementing the automatic mechanism mentioned above.

For the side-channel passive class of attacks, on the other hand, we simply assumed the use of the SEcube™ platform would suffice, given its BGA package, which firstly limits the access to the external pins of the processor and finally should theoretically add enough noise to the power traces through its `SiP` (System-in-Package) internal structure organization, to keep any Correlation Power Analysis and Differential Power Analysis attack difficult to carry on. Additional countermeasures on the code implementation can be added like the insertion of random time intervals during the execution of cryptographic functions. The goal, in this case, is to destroy the phase alignment between the several traces captured and needed to carry on statistical-based attacks like the aforementioned ones. A good list of all the possible countermeasures against power analysis are given in [28].

2.4 Implementation details

All the main KMS functionalities, described in Section 2.2, are implemented by the functions briefly analyzed in the following, specifically focusing on the new features. For more details on the previous implementation, please refer to [10].

- `kms_init` – initialization of the Key Management System at boot up and managing of the non-volatile memory;
- `kms_add_key` – this function, internally called by the `cmd_add_key()`, allows to add a new key record with user defined parameters (`key_id`, `key_size`, `cryptoperiod`). The default state at creation time is `Pre-Active` state. The key content is encrypted before being stored in the non-volatile memory, as we described in the previous sections;
- `kms_remove_key` – this function, internally called by the `cmd_remove_key()`, allows to remove a key from the KMS database and from the non-volatile memory. The content of the `C` structure in memory is completely overwritten with `0`'s, to avoid any confidential information being left in the memory;
- `kms_update_key` – this function, internally called by the `cmd_update_key()`, allows to update an already existing key (that may have been compromised, for example) without having to destroy it first and adding a new one with the same ID. After reseeding, the new key keeps the size of the previous one, now destroyed: this simplifies the key management on the host side and helps avoiding unfortunate scenarios where the host may be compromised and the various key sizes being disclosed. The only new information required by the host is the new cryptoperiod. The `expire_time` field of the updated key is set back to `0` (meaning that no expiration date has been computed yet) and the state will be set to the `preactive` one;
- `kms_get_key` – called by multiple commands in `device_cmds.c`, this function looks for the given `key_id` in the non-volatile memory using the `kms_search_key`, finally returning the entire structure (key value and related metadata) if the key is present. This function also checks, by calling the `kms_check_expire_time` function, if the requested key is expired or not according to its cryptoperiod. The key content is decrypted before being returned;
- `kms_list_key` – this function, internally called by the `cmd_list_keys()`, allows to scan the entire non-volatile memory, returning a collection of the `key_id`s found. No other information is disclosed to the host;
- `kms_change_status_key` – called by the commands like `cmd_activate_key()` (and similar ones), this function is in charge of changing the current status of the key, hence operating a state transition. Supported by the functions `key_state_transition()` and `key_state_transition_check()`, this function implements the `if then else` error-catching clause we discussed in section 2.3.2. Therefore, this mechanism implements a secure handling of the key's state transition, informing

the host about the presence of an illegal (i.e. non-existent in the state graph) state of the key or of the presence of glitching activity in the return codes of the called functions. Both scenarios are indicators of tampering activity or internal fault of the system: in both cases the key is automatically moved to the `compromised` state as a precaution, before informing the host of the presence of suspicious activity;

- `kms_add_plain_key` – similar to the above `kms_add_key()`, but in this case the actual key is not generated from a seed but used as provided directly from the input argument.

Those functions are supported by nine other accessory functions:

- `kms_search_key` – scans the whole non-volatile memory as a fixed-records database, reading a `struct kms_key` at each iteration and checking whether it matches the specified key ID; if so, the index of the matching key record is returned; the function stops when either `keys_num` valid keys are traversed or the memory finishes;
- `kms_check_expire_time` – checks whether a key is expired or not. This function is used in all those operations that require to retrieve a key from the KMS and use it (for instance, an AES encryption/decryption or an HMAC signature computation). This function returns a value that depends on the key's expiration state, notifying the caller function if the operation can be concluded normally or not. In case the actual timestamp is greater than the expiration time (computed when activating the key), the key is moved automatically by the device to the `deactivated` state;
- `kms_search_empty_record` – scans the non-volatile memory as a fixed-records database, reading a `struct kms_key` at each iteration in search of an empty record (i.e. a record with `state` field equal to `KMS_KEY_EMPTY`); if found before the end of the storage, its index is returned;
- `kms_keys_num_update` – basing on the mode defined by its input, increases or decreases the global variable `keys_num` and updates its value in the non-volatile memory;
- `generate_key` – generates a key of the desired size starting from a seed of an arbitrary length by subdividing it and hashing its sub-seeds; the result is then encrypted and returned to the caller ready to be stored in the KMS; the keys for HMAC-SHA256 and AES-256 are hardcoded in `kms.h` and are assumed to be unique for each device produced;
- `key_state_transition_check` – checks if a requested transition is permitted or not following the diagram in Figure 3. This function returns a `1` if the transition from the initial state to the final one is allowed and a `0` if it's not possible. If the requested destination state does not exist (also calles an "illegal" state) then the functions returns the value `-2`;
- `key_state_transition` – this function is responsible for the actual change of the key state. It calls the `key_state_transition_check` function which is used to enforce the state diagram and to check if the transition is possible or not. Secondly, it handles the possible return errors obtained, forwarding them to the caller function, which will precautionary move the key to the `compromised` state in case of suspicious activity;
- `can_encrypt` – this function checks if the given state (received as argument) allows the operations of AES encryption and the generation of a new HMAC-SHA256 signature. Like the above functions, it also informs the caller function if the given state is present or not in the state graph (hence declaring it as "illegal");
- `can_decrypt` – similarly to the previous function, this one checks if a key in that particular state can complete an AES decryption operation or do an HMAC signature check. Like the above functions, it also informs the caller function if the given state is present or not in the state graph (hence declaring it as "illegal").´

2.5 Communication framework integration

As already mentioned, in order to integrate the KMS in the already developed communication framework, two fields are needed in the *input packet* structure: `key_id` and `key_size` (see Figure 1).

The cryptographic commands to request an AES-256 encryption or decryption and an HMAC-SHA256 signature computation or checking can select the key to use by filling the *key ID* field: the requested key will be searched in the KMS database and, if found, it will be retrieved, decrypted and passed to the cryptographic function, otherwise an error is returned to the host. The key is also checked to have a state compliant to its actual utilization for the request purpose, as detailed in Section 2.1. Also, as far as the AES-256 algorithms are concerned, the key is checked to have a size of 32 bytes, otherwise an error is returned too.

The commands to manage the KMS (add, update, remove key) can point the desired key by means of the *key ID* field; the `cmd_add_key()` and `cmd_update_key()` commands also uses the input payload to send a seed to the key generation procedure, whose length in bytes is specified in the usual `length` field of the input packet header. Finally, the command to add a new key also exploits the *key size* field to specify the desired size in bytes for the new key to be added in the KMS database.

3 Key Agreement Protocol

In cryptography, a Key Agreement Protocol (KAP) is a protocol through which two or more parties can agree on a key in such a way that both influence the outcome. If properly done, this precludes undesired third parties from forcing a key choice on the agreeing parties.

In the system designed for this project, a KAP is used to solve the problem of *keys distribution* in *symmetric key cryptography*: both parties involved in a communication of a secret message must indeed possess the same key, which must be exchanged prior to using any encryption; this might be necessary even in the case when no already existing secure encryption channels are there between the parties, and the key must be agreed upon on an insecure channel, where malicious activity might be carried out.

The first Key Agreement Protocol was developed by Whitfield Diffie and Martin Hellman in 1976, and it is known as Diffie-Hellman Key Exchange [37]. Since then, with the advancements in technology and in cryptoanalysis, Diffie-Hellman protocol has been proven to be subject to man-in-the-middle attacks [38], and thus it does not represent anymore a valid choice for key agreement on insecure channels.

With the aim to obtain a more robust alternative to Diffie-Hellman protocol, the KAP implemented in this project takes inspiration from a protocol based on the combination of asymmetric (public-key) and symmetric (secret-key) cryptography (i.e. a secret key used to encrypt a randomly-generated public key) that allows two parties sharing a common password to exchange confidential and authenticated information, such as a shared session key, over an insecure network [18]. The approach is known as Encrypted Key Exchange (EKE) and solves the flaws of off-line dictionary attacks of common password-based KAPs, man-in-the-middle attacks and replay attacks.

3.1 Features

The implemented Key Agreement Protocol can be defined as a **Password-based Encrypted Key Exchange** implemented with **Exponential Key Exchange**. The protocol has been implemented assuming the centrality of the host, which is the entity orchestrating the interaction between the two devices which want to agree on the shared key. The device are not aware of the protocol flow themselves; they instead only receive commands from the host, with the required data attached, and send their reply, which is forwarded by the host to the other device taking part in the agreement.

The KAP is fully integrated in the already existing Key Management System, and it represents an additional option to add a new key to the KMS database, along with the usual possibility to generate a local, private key useful only to request encryption, decryption or hashing of messages to the same individual device. The difference of the keys generated with the help of the KAP is that they are shared with another device, and thus exchange of encrypted messages with other devices, once again orchestrated by the host, is made possible.

The main aim of the KAP is to let two devices agree on a **shared secret**, which is then hashed with HMAC-SHA256 using the same private key hard-coded in all devices to create a secret key equal for both. Due to the shared key being the output of HMAC-SHA256 algorithm, its length can be of at most 32 bytes. The size of the key to be generated can be selected by the host, along with the ID that the key has to assume in the KMS database and its cryptoperiod; as the usual KMS functionality to add new keys, the generated shared key is stored in the KMS in the *pre-active* state, which then needs an explicit activation before any cryptographic use.

3.1.1 Protocol specification

The protocol is articulated over six steps, executed in an alternated way by two devices interacting with each other by means of the host. In the following, the exact steps of the protocol are described; the two devices taking part in the interaction are named *device A* and *device B* without any loss of generality, since every device implements the functions to execute all of these steps, and could thus be the party A or B of the protocol, depending on who executes the first step. The protocol flow is also sketched in Figure 5.

1. **Step #1 (device A)** – A generates new β and α parameters with the characteristics in Section 3.1.2, along with a random number R_A; then it computes its local exponentiation $(\alpha^{R_A} \bmod \beta)$ and encrypts it with the password P, a fixed secret key hard-coded in all devices. Then it sends to A
$$\beta,\ \alpha,\ \mathrm{Enc}_P[\alpha^{R_A} \bmod \beta]$$
2. **Step #2 (device B)** – B receives β, α and $(\alpha^{R_A} \bmod \beta)$, picks a random number R_B and computes its local exponentiation $(\alpha^{R_B} \bmod \beta)$; then, it is also able to compute the shared exponentiation as
$$((\alpha^{R_A} \bmod \beta)^{R_B} \bmod \beta) = (\alpha^{R_A \cdot R_B} \bmod \beta)$$
which is the *shared secret*. The shared secret is hashed with HMAC-SHA256 (using another fixed secret key H hard-coded in all devices) to obtain a shared secret key K of the desired size (at most 32 bytes). Finally, B generates a random challenge Chlg_B and sends back to A the encrypted data
$$\mathrm{Enc}_P[\alpha^{R_B} \bmod \beta],\ \mathrm{Enc}_K[\mathrm{Chlg}_B]$$
3. **Step #3 (device A)** – A receives $(\alpha^{R_B} \bmod \beta)$, computes the shared secret as
$$((\alpha^{R_B} \bmod \beta)^{R_A} \bmod \beta) = (\alpha^{R_B \cdot R_A} \bmod \beta)$$
and hashes it with H to obtain K; now it is able to decrypt Chlg_B and solve it. A then generates a random challenge Chlg_A and sends to B
$$\mathrm{Enc}_K[\mathrm{Solution}(\mathrm{Chlg}_B),\ \mathrm{Chlg}_A]$$
4. **Step #4 (device B)** – B receives $\mathrm{Solution}(\mathrm{Chlg}_B)$ and Chlg_A, verifies that $\mathrm{Solution}(\mathrm{Chlg}_B)$ is correct and, if so, generates the reply to Chlg_A; then it sends to A
$$\mathrm{Enc}_K[\mathrm{Solution}(\mathrm{Chlg}_A)]$$
5. **Step #5 (device A)** – A receives $\mathrm{Solution}(\mathrm{Chlg}_A)$ and, if it is correct, the agreement concluded successfully and it can store the newly generated key K in the KMS database with the specified ID, size and cryptoperiod, in the pre-active state. When this step concludes successfully, A sends a positive acknowledgment to the host, which forwards it to B in the form of the next step command request.
6. **Step #6 (device B)** – when this step is launched on B it is interpreted as a positive acknowledgment from A, so that B also stores the key K in its KMS database with the given ID, size and cryptoperiod, in the pre-active state.

The protocol may encounter an issue during any of the listed steps; this may happen due to:

- no primitive roots for β have been found during the computation of α (see Section 3.1.2);
- too big requested key size for K, larger than 32 bytes;
- the challenge solution of either A or B is incorrect due to one of them wrongly generating the shared key K or wrongly applying the solution;
- selected key ID already present in the KMS database, or no space left in the database;
- other hard faults depending on the libraries implementation, discussed in Section 3.2.

In such cases, the protocol can be reset and restarted from the beginning. In addition to the functions implementing each one of these steps, the EKE library is indeed provided also with a function to reset the state of the protocol, which can be issued at any time by the host. The function is also useful to avoid any leak of stale data from the buffers used inside the device.

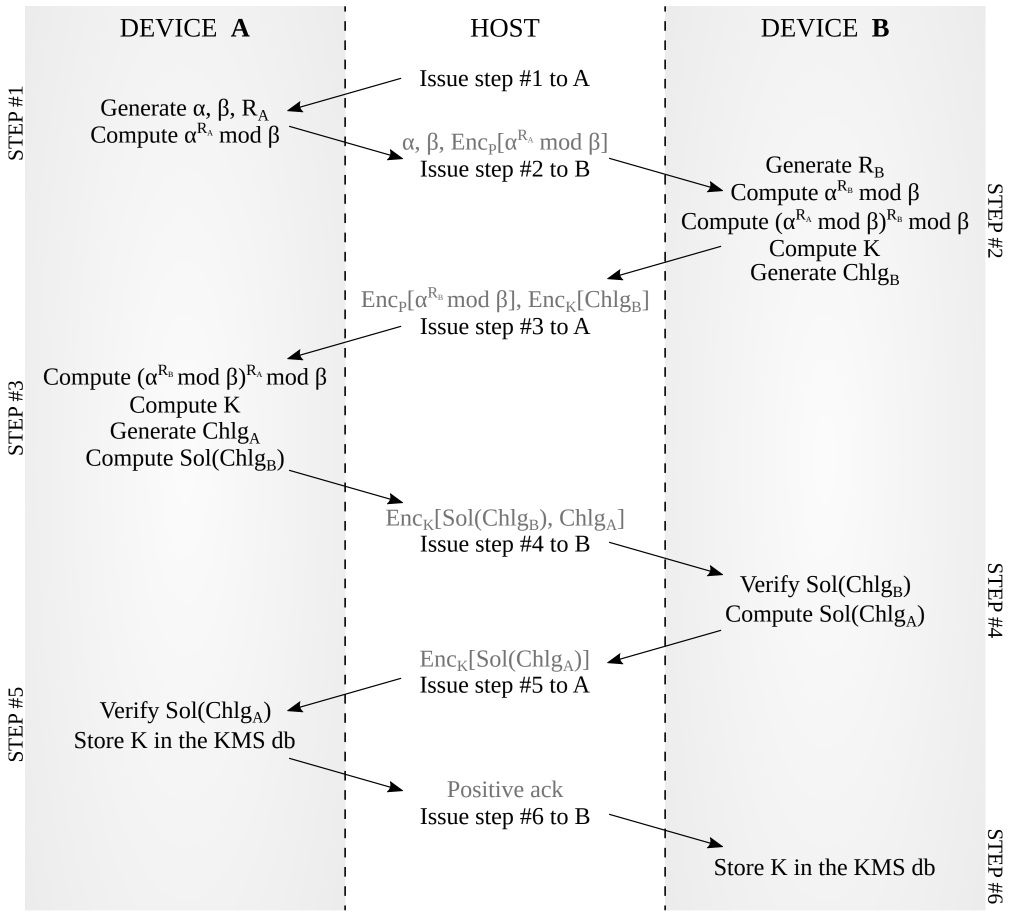

Figure 5: Schematic flow of the implemented Key Agreement Protocol; the writings highlighted in gray under the *host* column represent the data output of a protocol step executed by a device, sent to the host and forwarded to the other device while issuing the following protocol step.

3.1.2 Security features

The **main vulnerabilities of KAPs** on insecure networks are overcome by means of two features:

- every quantity exchanged between the two parties is encrypted using a fixed, previously shared secret password P: this prevents **man-in-the-middle** attacks, because an intruder cannot sit in the middle; also attempts to **guess** P **off-line** are useless because a successful attack would only retrieve α^{R_A} mod β or α^{R_B} mod β, with the quantity $\alpha^{R_A \cdot R_B}$ mod β remaining unknown due to the two random numbers R_A and R_B never being sent (not even encrypted) on the network;

- after the shared secret key K is generated, the parties exchange random challenges to verify the outcome of the agreement: this protects against **replay attacks**, which exploit old, stale data which may persist in the communication channel; since the challenges are random, the solution is different every time and old messages are useless.

Also, according to [18], the KAP can be protected against **cut-and-paste** attacks, even with smaller sizes of the parameters, if different β and α are picked at each session (to prevent precalculation) and with the following specifications:

- β must be a prime number;

- β must be large enough, to protect against **precalculation of tables**;

- $\phi(\beta) = \beta - 1$ must have at least one large factor, to guard against **Pohlig and Hellman's algorithm**;

- α must be a primitive root of the field $GF(\beta)$.

Satisfying the previous conditions, β and α can also be sent in clear text without any risk.

The strength of the algorithm used to encrypt the data sent over the insecure network with P and K is also of main relevance for the KAP security; the one employed in the devices is AES-256.

3.2 Implementation details

The library implementing the protocol, named `eke_exp`, is based on the functionalities mainly offered by three other libraries: the Number theory library (Section 1.1.6), the custom Random number generator driver (Section 1.1.3) and the Key management system library (Section 1.1.4).

Despite the literature recommending a size of at least 200 bits for the parameters employed in the protocol and for the data structures used for modular operations [18], the KAP in this project is implemented over 32-bit data structures, in turn based on the 32-bit modular operations offered by the Number theory library, as a proof-of-concept for the protocol working principle. At the cost of higher complexity and resources utilization, the protocol can be easily extended to be more secure in terms of parameters size.

Cryptographic operations are also employed by the KAP: in particular, the AES-256 algorithm (Section 1.1.8) is used for the encryption of the data exchanged between the devices taking part to the key agreement. To encrypt the initial messages (i.e. the results of the local exponentiations) a 32-byte password P is used; P is a secret key hard-coded in the KAP library and thus equal for all the devices. HMAC-SHA256 algorithm (Section 1.1.9) is instead used to hash the shared secret and obtain K, in combination with a secret 64-byte key H hard-coded in the same way in the KAP library, and thus equal for all the devices.

3.2.1 KAP functionalities implementation

In terms of actual implementation of the KAP functionalities, the functions implemented by the library are:

- `eke_init()` – the initialization function, simply calling `eke_kap_reset` to initialize the KAP by setting its global variables to zero;

- from `eke_kap_step1_a()` to `eke_kap_step6_b()` – the 6 functions implementing the steps of the protocol; note that `eke_kap_step1_a()` is the first function executed by A, `eke_kap_step2_b()` is the first function executed by B, `eke_kap_step3_a()` is the second function executed by A, and so on.

- `eke_kap_reset()` – some values are needed on the same device among different steps of the protocol (i.e. among different function calls, for example the same β is used both in `eke_kap_step1_a()` to compute the local exponentiation and in `eke_kap_step3_a()` to compute the shared secret), thus they must be retained for some time; to solve this, some global variables are used as buffers. This function resets them to zero; it can be launched at any time by the host as a command, and it is also run at device start-up (during KAP initialization) and at the beginning and end of each agreement session (i.e. during the first and last steps of the protocol);

- `generate_beta_alpha()` – generates new values for β and α parameters following the specifications in 3.1.2; in particular, a large enough random prime must be picked for β, which in this implementation is considered such if $> 2^{26}$. There are 199322412 prime numbers in the range $[2^{26}, 2^{32} - 1]$, i.e. there is approximately a 4.7% chance of hitting a prime number from a random 32-bit number larger than 2^{26}. Since the random prime is researched by means of a `while` loop in which a random `uint32_t` is first generated and then tested for its primality, the primality test is repeated on average 21.2 times when a new β is requested. If the random value is prime, it is also checked to have at least one large enough factor, considered here as $> 2^{25}$. α is then simply generated by computing a large primitive root of β. Even with the random prime generation being a bottleneck in terms of computational complexity, the performed tests proved that the employed fast primality test makes this operation very quick and still perfectly feasible for a real application of the device;

- `solve_challenge()` – solves the challenge represented by the input 32-bit value; the challenge is just a control value received by the other party, which must be transformed through this function implementing a secret operation and sent back to the party which generated it, which in turn applies the same function and verifies the correctness of the solution. The secret operation hard-coded in each device for the challenge solution is $(challenge \oplus \texttt{0x62F870A0}) + \texttt{0x00A02ADE}$.

The custom Random number generator is employed several time in the functions above, with its interface function `rng_get_random32()` to simply fill a `uint32_t` variable with 32 random bits, to generate: the local random 32-bit numbers R_A and R_B, the random 32-bit challenges Chlg_A and Chlg_B, and the random prime number β. In particular, to generate a random prime, `rng_get_random32()` is used in a loop in combination with the fast primality test `is_prime()` from the Number theory library, to randomly generate 32-bit values until a prime number is hit.

A note has to be made about a new function introduced in the KMS library `kms`: the usual function to add a new key in the KMS is `kms_add_key()`, and it hashes a seed to generate the actual key to be stored in the database. The KAP library instead needed instead a function able to store a key in the KMS database passed as-is in the input arguments, due to it begin already generated and agreed by the two devices; this function is `kms_add_plain_key()` and works in the same exact way of `kms_add_key()`, but it directly takes as input the content of the key, instead of a seed to be hashed.

3.2.2 Errors, warnings and return codes

Several errors and warnings can be raised by the device and returned to the host, basing on the functionalities implemented by the KAP library and in the libraries on which it relies. In the following, all possible issues which might occur during the KAP flow are listed:

- no primitive roots can be found for the generated β prime number during α computation (*warning*, can be handled with a failure message to the host, which can restart the protocol to try with another β value);
- the key size requested by the host is too large, bigger than the 32-byte maximum imposed by the HMAC-SHA256 output (*warning*, can be handled with a failure message to the host, which can request again such protocol step with a smaller key size);
- the solution to the challenge from A computed by B, or vice-versa, is not correct (*warning*, something went wrong and the devices did not correctly agree on the same key, or one of them guessed a wrong challenge solution, which might be symptom of an attack; the host can just re-run the KAP);
- the KMS database in the non-volatile memory run out of space to store a new key (*warning*);
- a key with the ID specified by the host already exists in the KMS database (*warning*, the host can just pick a different ID);
- the custom Random number generator returned an hard fault (*hard fault*, the device execution is aborted); this may happen, basing on the Random number generator implementation, due to ARM Semihosting not being able to open the `/dev/urandom` file;
- the computation of β factors run out of memory: due to the heap not being usable, a space of at most 50 `uint32_t` is statically allocated to such factorization; if the number of factors exceeds 50, an hard fault is returned (*hard fault*, the device execution is aborted and needs to be restarted, then the host can relaunch the KAP to try with a different β);

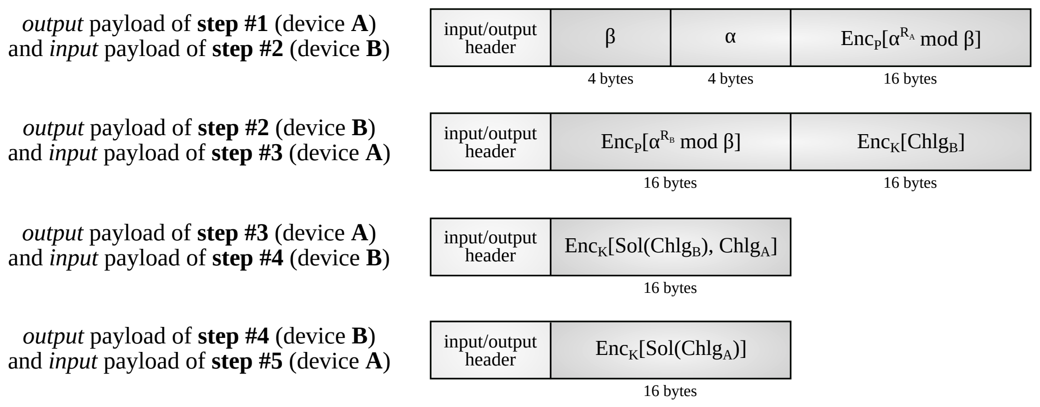
output payload of step #1 (device A) and input payload of step #2 (device B)
input/output header
β
α
Enc_P[α^{R_A} mod β]
4 bytes
4 bytes
16 bytes
output payload of step #2 (device B) and input payload of step #3 (device A)
input/output header
Enc_P[α^{R_B} mod β]
Enc_K[Chlg_B]
16 bytes
16 bytes
output payload of step #3 (device A) and input payload of step #4 (device B)
input/output header
Enc_K[Sol(Chlg_B), Chlg_A]
16 bytes
output payload of step #4 (device B) and input payload of step #5 (device A)
input/output header
Enc_K[Sol(Chlg_A)]
16 bytes

Figure 6: Input and output payloads structures for device commands implementing the KAP steps: the implementation of each command in the `device_cmds` library is able to decode its own input payload structure and encode the parameters to produce its related output payload. For brevity, the input and output packets header are not detailed in the figure.

- any of the possible KMS hard fault occurred, e.g. non-volatile memory, timer peripheral, total keys number update (*hard fault*, device execution is aborted);

As all the other functionalities implemented in the device libraries, the return code of the functions of `eke_exp` library is standardized: a return value `>0` represents a success with some output details (e.g. its size); a return value of `0` can represent either a success without anything else to communicate or a soft failure, depending on the function; a return value of `-1` always represents an hard fault which cannot be handled, making the device go into an infinite loop, while any other return value `<0` represents a soft error, returning a warning to the caller which must be handled in a specific way.

Thanks to the fact that all functions in the project implement such codification of return values, any issue caused at any level of the function calls hierarchy returns a success, a warning or an hard fault to its caller, until the upper level is reached, which also returns the message to the host. Also, at each function call level, a message, a warning or an error is printed on the trace while returning from the function, which provides the user with an exact trace back of the issue in terms of function calls. Such prints on the trace can be enabled with the define `__VERBOSE` in `eke_exp.c`, as in all the other libraries.

3.2.3 Communication framework integration

Seven actual commands, consisting in the six protocol steps in addition to the KAP reset command, have been added to the ones that can be requested for execution by an host to a device; all of them are described in details in Appendix A.

As already mentioned, the two devices agreeing on the shared key K do not directly communicate with each other and they are not aware of the current state of the KAP; the host is the only one aware of this and it orchestrates the communication between the devices.

In order for the device to exploit the input and output packets of the communication framework, whose structure is sketched in Figure 1, each command of the KAP defines its own fixed structure of the packet payload. Note that only the payload is used to exchange the key agreement data, and no new static fields have been added to the packet data structures: this would have indeed generated to many new fields increasing the packet size of any communication between device and host, for something that only KAP commands would have used. This choice thus pays the price of higher complexity for specific encoding and decoding of KAP data in the payloads of KAP commands, with the aim to avoid to increase the size of all exchanged packets, for any command.

All the devices are aware of the structure of the payload of any KAP command, and, basing on that, they are able to decode the related input payload to extract the parameters sent by the other device, and then encode their outputs for the output payload to forward to the next step. The host API is also aware of the structure of the KAP commands input and output payloads and thus only need the host application to provide them with the parameters (e.g. β, α, $\mathrm{Enc}_P[\alpha^{R_A} \bmod \beta]$, ...), which will be automatically encoded in a payload. The structures of the payloads for each KAP command are shown in Figure 6.

References

[1] HO Hundley and Robert H. Anderson. Emerging challenge: Security and safety in cyberspace. *IEEE Technology and Society Magazine*, 14(4):19–28, 1995.

[2] *International Standard: Information technology. Security techniques. Guidelines for information cybersecurity. ISO/IEC 27032.* ISO/IEC, 2012.

[3] Mike Isaac and Sheer Frenkel. Facebook Security Breach Exposes Accounts of 50 Million Users, *The New York Times*. Available at `https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html`, September 2018. Accessed 30 January 2021.

[4] David E. Sanger. Russian Hackers Broke Into Federal Agencies, U.S. Officials Suspect, *The New York Times*. Available at `https://www.nytimes.com/2020/12/13/us/politics/russian-hackers-us-government-treasury-commerce.html`, December 2020. Accessed 30 January 2021.

[5] Carl H Meyer. Cryptography-a state of the art review. In *Proceedings. VLSI and Computer Peripherals. COMPEURO 89*, pages 4–150. IEEE, 1989.

[6] Dawn M. Turner. What is Key Management? A CISO Perspective, *Cryptomathic*. Available at `https://www.cryptomathic.com/news-events/blog/what-is-key-management-a-ciso-perspective`, February 2016. Accessed 30 January 2021.

[7] Elaine Barker, Miles Smid, Dennis Branstad, and Santos Chokhani. Nist special publication 800-130: A framework for designing cryptographic key management systems. *National Institute of Standards and Technology Report*, 2013.

[8] Antonio Varriale, Elena Ioana Vatajelu, Giorgio Di Natale, Paolo Prinetto, Pascal Trotta, and Tiziana Margaria. Secube™: An open-source security platform in a single soc. In *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2016.

[9] Flavia Caforio, Samuele Y. Cerini, and Sergio Mazzola. Laboratory #2 report, *Cybersecurity for Embedded Systems*, Politecnico di Torino, June 2020.

[10] Flavia Caforio, Samuele Y. Cerini, and Sergio Mazzola. Laboratory #3 report, *Cybersecurity for Embedded Systems*, Politecnico di Torino, July 2020.

[11] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005.

[12] Semihosting, *ARM Compiler toolchain - Developing Software for ARM Processors.* Available at `https://developer.arm.com/documentation/dui0471/g/Semihosting`. Accessed 30 January 2021.

[13] Ieee standard specification for password-based public-key cryptographic techniques. *IEEE Std 1363.2-2008*, pages 1–140, 2009.

[14] Russian peasant multiplication, *Cut The Knot.* Available at `http://www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml`. Accessed 30 January 2021.

[15] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* john wiley & sons, 2007.

[16] Michal Forišek and Jakub Jancina. Fast primality testing for integers that fit into a machine word. 2015.

[17] Primitive root, *CP-Algorithms.* Available at `https://cp-algorithms.com/algebra/primitive-root.html`. Accessed 30 January 2021.

[18] Steven Michael Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. 1992.

[19] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

[20] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication, 1997.

[21] S Kelly and S Frankel. Using hmac-sha-256, hmac-sha-384, and hmac-sha-512 with ipsec. Technical report, RFC 4868, May, 2007.

[22] Donald Eastlake and Tony Hansen. Us secure hash algorithms (sha and sha-based hmac and hkdf). Technical report, RFC 6234, May, 2011.

[23] Matteo Fornero, Nicolò Maunero, Paolo Prinetto, Gianluca Roascio, and Antonio Varriale. *SEcube™ Open Source SDK 1.5.0.* Accessed 3 October 2020.

[24] Elaine Barker and Quynh Dang. Nist special publication 800-57 part 1, revision 4. *NIST, Tech. Rep*, 16, 2016.

[25] Multi-purpose usb 2.0 token based on secube™ chip. Available at `https://www.secube.eu/products/usecube-bundle-including-5-usecube-tokens-and-1-devkit/`. Accessed 12 February 2021.

[26] Rugged 8" tablet powered by secube™ chip for use in toughest, critical environments. Available at `https://www.secube.eu/products/secube-bundle-including-10-secube-chips-and-1-devkit/`. Accessed 12 February 2021.

[27] Colin O'Flynn and Zhizhang David Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 243–260. Springer, 2014.

[28] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[29] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[30] Josh Datko, Chris Quartier, and Kirill Belyayev. Breaking bitcoin hardware wallets. *DEF CON 2017*, 2017.

[31] 35c3 - wallet.fail - hacking the most popular cryptocurrency hardware wallets. Available at `https://www.youtube.com/watch?v=Y1OBIGslgGM`. Accessed 12 February 2021.

[32] Breaking aes with chipwhisperer - piece of scake (side channel analysis 100). Available at `https://www.youtube.com/watch?v=FktI4qSjzaE`. Accessed 12 February 2021.

[33] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[34] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptol. ePrint Arch.*, 2016:810, 2016.

[35] Clock glitch attack examples - bypassing password check. Available at `https://www.youtube.com/watch?v=Ruphw9-8JWE`. Accessed 12 February 2021.

[36] Hardware power glitch attack (fault injection) - rhme2 fiesta (fi 100). Available at `https://www.youtube.com/watch?v=6Pf3pY3GxBM`. Accessed 12 February 2021.

[37] Martin Hellman. An overview of public key cryptography. *IEEE Communications Society Magazine*, 16(6):24–32, 1978.

[38] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.

A Commands API

In the following all the device commands whose execution can be requested by the host are documented; since each command functionality implemented on the device corresponds to a unique given command in the `host_cmds` library, the functionalities are reported here with the name of the corresponding function which the host must call. Also, all the necessary parameters that the host has to provide are described.

All commands functions launched from the host have the following return value format:

- `-1` if the communication with the device did not conclude successfully due to some error, and the `response` array content is meaningless;
- `OUT_ERR_EMPTY = 0` if the output header is empty, meaning that no response was received from the device within the response timeout; the `response` array content is meaningless;
- `OUT_ERR_NOERR = 1` if the command was successfully executed and concluded by the device, and it correctly sent its reply back to the host; the output in the `response` array is meaningful;
- `OUT_ERR_ERROR = 2` if the communication with the device concluded correctly but the command executed on the device returned an error or a warning, meaning that the output `response` contains an error message string rather than the expected content.

1. Ping

int **cmd_test_ping**(uint8_t device_id, uint8_t *response, uint16_t *response_len_p)

Test command to check if the device is alive; it does not send any payload to the device. The string `Pong` should be received if everything works well in the communication. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

2. Test payload

int **cmd_test_payload**(uint8_t device_id, uint8_t *response, uint16_t *response_len_p)

Test command to check if the exchange of payloads between host and device works well; it sends a default string `Test dummy input payload` to the device. The device checks the correct reception of the default string and respond with a string `Test dummy output payload`, which should be received by the host if everything works well in the communication. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

3. AES-256 encryption

int cmd_encrypt(uint8_t device_id, uint8_t *payload, uint16_t payload_len, aes_mode_t aes_mode, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Encrypt a plain text with the AES-256 encryption algorithm, with ECB or CBC operating mode, exploiting a 32-byte key stored in the device KMS database. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `payload` array of `PAYLOAD_BUF_IN_SIZE` elements of `uint8_t` type containing the plain text;
- a `uint16_t payload_len` variable, specifying the plain text length in bytes;
- an `aes_mode` input specifying the operating mode, which can be `AES_ECB_MODE` or `AES_CBC_MODE`;
- a `key_id` integer specifying the ID of the 32-byte key in the KMS to use for encryption;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the cipher text;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

4. AES-256 decryption

int cmd_decrypt(uint8_t device_id, uint8_t *payload, uint16_t payload_len, aes_mode_t aes_mode, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Decrypt a cipher text encrypted with the AES-256 encryption algorithm, with ECB or CBC operating mode, exploiting a 32-byte key stored in the device KMS database. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `payload` array of `PAYLOAD_BUF_IN_SIZE` elements of `uint8_t` type containing the cipher text;
- a `uint16_t payload_len` variable, specifying the cipher text length in bytes;
- an `aes_mode` input specifying the operating mode, which can be `AES_ECB_MODE` or `AES_CBC_MODE`;
- a `key_id` integer specifying the ID of the 32-byte key in the KMS to use for decryption;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the plain text;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

5. HMAC-SHA256 hashing

int cmd_hmac_sign(uint8_t device_id, uint8_t *payload, uint16_t payload_len, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Hash a message by means of the HMAC-SHA256 message authentication code based on the SHA-256 cryptographic hash function, to obtain a 32-byte fixed-size message digest; a key of an arbitrary length stored in the device KMS database is used. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `payload` array of `PAYLOAD_BUF_IN_SIZE` elements of `uint8_t` type containing the message to hash;
- a `uint16_t payload_len` variable, specifying the message length in bytes;

- a `key_id` integer specifying the ID of the key in the KMS to use for hashing;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the message digest;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

6. HMAC-SHA256 checking

int cmd_hmac_check(uint8_t device_id, uint8_t *payload, uint16_t payload_len, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Given a payload message and an attached signature, check if the digest is equal to the expected one, hence proving the authenticity and the integrity of the message itself. The digest to be compared to the received one is computed by means of the HMAC-SHA256 message authentication code based on the SHA-256 cryptographic hash function. A key of an arbitrary length stored in the device KMS database is used. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `payload` array of `PAYLOAD_BUF_IN_SIZE` elements of `uint8_t` type with the following structure: the first 32 bytes of the payload contain the signature with which the message came; the following `payload_len` − 32 bytes contain the message itself;
- a `uint16_t payload_len` variable, specifying the whole payload length in bytes (signature + message);
- a `key_id` integer specifying the ID of the key in the KMS to use for hashing;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the message digest;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

7. Add key to KMS

int cmd_add_key(uint8_t device_id, uint8_t *payload, uint16_t payload_len, uint32_t key_id, uint16_t key_size, uint32_t key_cryptoperiod, uint8_t *response, uint16_t *response_len_p)

Generate a new key of the desired size and with the specified cryptoperiod by hashing with HMAC-SHA256 the user-defined seed; the generated key is encrypted and stored in the KMS database with the given unique ID. The newly added key assumes the *pre-active* state, and needs an explicit activation in order to be used after its generation. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `payload` array of `PAYLOAD_BUF_IN_SIZE` elements of `uint8_t` type containing the seed for the key generation;
- a `uint16_t payload_len` variable, specifying the seed length in bytes;
- a `key_id` integer specifying the ID to assign to the new key in the KMS;
- a `key_size` integer specifying the desired size of the new key to be generated;
- a `key_cryptoperiod` integer specifying the lifetime (in seconds) of the key;

- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the command response, specifying the operation outcome;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

8. Remove key from KMS

int cmd_remove_key(uint8_t device_id, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Remove the key specified by the given ID from the KMS database, entirely wiping all of its traces. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID of the key to remove from KMS;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the command response, specifying the operation outcome;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

9. Update key in KMS

int cmd_update_key(uint8_t device_id, uint8_t *payload, uint16_t payload_len, uint32_t key_id, uint32_t key_cryptoperiod, uint8_t *response, uint16_t *response_len_p)

Update the already existing key specified by the given ID in the KMS database; the new key has the same size of the previous one but its content is re-seeded with a new seed sent by the host. In addition, a new cryptoperiod must be specified. A re-seeded key falls back on the *pre-active* state, no matter its previous state. Needed arguments:

- a `device_id` specifying the device to which the command must be sent;
- a `payload` array of `PAYLOAD_BUF_IN_SIZE` elements of `uint8_t` type containing the seed for the key re-seed;
- a `uint16_t payload_len` variable, specifying the seed length in bytes;
- a `key_id` integer specifying the ID of the key in the KMS to be updated;
- a `key_cryptoperiod` integer specifying the lifetime (in seconds) of the key;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the command response, specifying the operation outcome;
- the address `response_len_p` of a `uint16_t response_len` variable, to store the response length in bytes.

10. List keys in KMS

int cmd_list_keys(uint8_t device_id, uint8_t *response, uint16_t *response_len_p)

List the IDs of all valid key records in the KMS database (i.e. records with the state field not set to *empty*); no other information about the key is disclosed for security reasons: it is due of the host to keep track of the characteristics and the state of the keys known to it. If the command is successful (i.e. `OUT_ERR_NOERR` returned), `response` must be interpreted as an array of `response_len`/4 elements of `uint32_t` type (i.e. a list of keys IDs).

- a `device_id` specifying the device to which the command must be sent;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

11. Activate key in KMS

int cmd_activate_key(uint8_t device_id, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Activate an existing key, specified by its key ID (i.e. transition to the key state *active*). The specific key state graph is checked upon completion of the command: if the transition from the current state of the key is not allowed, the host will be notified with an error; no additional information about the keys are disclosed. The device also enforces additional checks against tampering and glitching activities affecting the internal code implementation, notifying the host of potentially suspicious activity.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID of the key in the KMS to be updated;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

12. Suspend key in KMS

int cmd_suspend_key(uint8_t device_id, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Suspend an existing key, specified by its key ID (i.e. transition to the key state *suspended*).The specific key state graph is checked upon completion of the command: if the transition from the current state of the key is not allowed, the host will be notified with an error; no additional information about the keys are disclosed. The device also enforces additional checks against tampering and glitching activities affecting the internal code implementation, notifying the host of potentially suspicious activity.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID of the key in the KMS to be updated;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

13. Deactivate key in KMS

int cmd_deactivate_key(uint8_t device_id, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Deactivate an existing key, specified by its key ID (i.e. transition to the key state *deactivated*). The specific key state graph is checked upon completion of the command: if the transition from the current state of the key is not allowed, the host will be notified with an error; no additional information about the keys are disclosed. The device also enforces additional checks against tampering and glitching activities affecting the internal code implementation, notifying the host of potentially suspicious activity.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID of the key in the KMS to be updated;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

14. Compromise key in KMS

int cmd_compromise_key(uint8_t device_id, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Flag an existing key, specified by its key ID, as compromised (i.e. transition to the key state *compromised*). For security reasons, this operation is always allowed, no matter the current state of the key. The device also enforces additional checks against tampering and glitching activities affecting the internal code implementation, notifying the host of potentially suspicious activity.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID of the key in the KMS to be updated;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

15. Destroy key in KMS

int cmd_destroy_key(uint8_t device_id, uint32_t key_id, uint8_t *response, uint16_t *response_len_p)

Flag an existing key, specified by its key ID, as destroyed (i.e. transition to the key state *destroyed*). The content of the key and its other metadata are not affected by this command. For security reasons, this operation is always allowed, no matter the current state of the key. The device also enforces additional checks against tampering and glitching activities affecting the internal code implementation, notifying the host of potentially suspicious activity.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID of the key in the KMS to be updated;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

16. Key Agreement Protocol - Step #1 (device A)

int cmd_kap_1a(uint8_t device_id, uint32_t *beta_p, uint32_t *alpha_p, uint8_t *enc_pow_a, uint8_t *response, uint16_t *response_len_p)

Launch step #1 of the Key Agreement Protocol, meant to be executed on device A. Device A generates new β and α parameters, returning them in plain text to the host; it also computes its local exponentiation, encrypts it with an hard-cored password P common to all devices and returns the cipher text to the host.

- a `device_id` specifying the device to which the command must be sent;
- the address of a `uint32_t beta` variable, to return the newly generated β parameter;
- the address of a `uint32_t alpha` variable, to return the newly generated α parameter;
- an allocated `enc_pow_a` array of 16 elements of `uint8_t` type, to return the `uint32_t` local exponentiation result of device A encrypted with AES-256 (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

17. Key Agreement Protocol - Step #2 (device B)

int cmd_kap_2b(uint8_t device_id, uint32_t key_size, uint32_t beta, uint32_t alpha, uint8_t *enc_pow_a, uint8_t *enc_pow_b, uint8_t *enc_chlg_b, uint8_t *response, uint16_t *response_len_p)

Launch step #2 of the Key Agreement Protocol, meant to be executed on device B. Device B receives β, α and the result of device A local exponentiation, so that it can generate the new shared key K. It returns to the host its local exponentiation result encrypted with P and a random challenge encrypted with K, to be able to check whether device A will reach the same agreed key K.

- a `device_id` specifying the device to which the command must be sent;
- a `key_size` integer specifying the desired size of the new key upon which devices A and B are agreeing;
- the `beta` parameter generated by device A in step #1;
- the `alpha` parameter generated by device A in step #1;
- a `enc_pow_a` array of 16 elements of `uint8_t` type, containing the device A local exponentiation encrypted with AES-256 using P;
- an allocated `enc_pow_b` array of 16 elements of `uint8_t` type, to return the `uint32_t` local exponentiation result of device B encrypted with AES-256 using P (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- an allocated `enc_chlg_b` array of 16 elements of `uint8_t` type, to return the `uint32_t` random challenge generated by device B and encrypted with AES-256 using K (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

18. Key Agreement Protocol - Step #3 (device A)

int cmd_kap_3a(uint8_t device_id, uint32_t key_size, uint8_t *enc_pow_b, uint8_t *enc_chlg_b, uint8_t *enc_reply_a, uint8_t *response, uint16_t *response_len_p)

Launch step #3 of the Key Agreement Protocol, meant to be executed on device A. Device A receives device B local exponentiation, generating the same shared key K, and the random challenge generated by device B. It then generates the reply to the challenge from device B and a new random challenge, encrypting everything with K before returning the resulting cipher text to the host.

- a `device_id` specifying the device to which the command must be sent;
- a `key_size` integer specifying the desired size of the new key upon which devices A and B are agreeing;
- a `enc_pow_b` array of 16 elements of `uint8_t` type, containing the `uint32_t` local exponentiation result of device B encrypted with AES-256 using P (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- a `enc_chlg_b` array of 16 elements of `uint8_t` type, containing the `uint32_t` random challenge generated by device B and encrypted with AES-256 using K (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- an allocated `enc_reply_a` array of 16 elements of `uint8_t` type, to return the two `uint32_t` elements {reply to challenge from device B, random challenge of device A} generated by device A and encrypted with AES-256 using K (8 bytes padded to 16 bytes, the minimum length of AES-256 output);
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

19. Key Agreement Protocol - Step #4 (device B)

int cmd_kap_4b(uint8_t device_id, uint8_t *enc_reply_a, uint8_t *enc_reply_b, uint8_t *response, uint16_t *response_len_p)

Launch step #4 of the Key Agreement Protocol, meant to be executed on device B. Device B receives the reply to the random challenge it previously generated and, if correct, goes ahead computing the response to the challenge received from device A and sending it back encrypted with K to the host.

- a `device_id` specifying the device to which the command must be sent;
- a `enc_reply_a` array of 16 elements of `uint8_t` type, containing the two `uint32_t` elements {reply to challenge from device B, random challenge of device A} generated by device A and encrypted with AES-256 using K (8 bytes padded to 16 bytes, the minimum length of AES-256 output);
- an allocated `enc_reply_b` array of 16 elements of `uint8_t` type, to return the `uint32_t` reply to the challenge from device A, computed by device B and encrypted with AES-256 using K (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

20. Key Agreement Protocol - Step #5 (device A)

int cmd_kap_4b(uint8_t device_id, uint32_t key_id, uint32_t key_cryptoperiod, uint8_t *enc_reply_b, uint8_t *response, uint16_t *response_len_p)

Launch step #5 of the Key Agreement Protocol, meant to be executed on device A. Device A receives the reply to the random challenge it previously generated and, if correct, stores the newly agreed key K in the KMS database with the specified ID and cryptoperiod; the key is set in the *pre-active* state, and needs an explicit activation before its actual use. Device A also sends a positive acknowledgment back to the host, to signal the success of the operation to device B.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID to assign to the newly agreed key in the KMS;
- a `key_cryptoperiod` integer specifying the lifetime (in seconds) of the new key;
- a `enc_reply_b` array of 16 elements of `uint8_t` type, containing the `uint32_t` reply to the challenge from device A, computed by device B and encrypted with AES-256 using K (4 bytes padded to 16 bytes, the minimum length of AES-256 output);
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

21. Key Agreement Protocol - Step #6 (device B)

int cmd_kap_4b(uint8_t device_id, uint32_t key_id, uint32_t key_cryptoperiod, uint8_t *response, uint16_t *response_len_p)

Launch step #6 of the Key Agreement Protocol, meant to be executed on device B. Requesting this command to device B from the host is equivalent to forwarding to device B the positive acknowledgment generated by device A during step #5: device B thus stores the newly agreed key K in the KMS database with the specified ID and cryptoperiod; the key is set in the *pre-active* state, and needs an explicit activation before its actual use.

- a `device_id` specifying the device to which the command must be sent;
- a `key_id` integer specifying the ID to assign to the newly agreed key in the KMS;
- a `key_cryptoperiod` integer specifying the lifetime (in seconds) of the new key;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

22. Reset Key Agreement Protocol

int cmd_kap_reset(uint8_t device_id, uint8_t *response, uint16_t *response_len_p)

Reset the state of the KAP by wiping all global variables used as buffers among its different steps. This command is useful to avoid information leak due to stale data still present in the device, or to start the protocol flow from scratch if it gets interrupted before its successful termination.

- a `device_id` specifying the device to which the command must be sent;
- a `response` array of `PAYLOAD_BUF_OUT_SIZE` elements of `uint8_t` type to return the device output;
- the address of a `uint16_t response_len` variable, to store the response length in bytes.

B Project files organization

In the following, the main directories of the project and their file organization is reported; note that, as far as the device-side project is concerned, only files which were modified to add our functionalities are reported.

`secube_proj`

- `device` ... Device-side C project, managed by Eclipse
  - `include`
    - `com_channel.h` = Communication system low-level driver and high-level interface
    - `device_cmds.h` = All implemented device commands functionalities
    - `nv_mem.h` = Non-volatile memory low-level driver and high-level interface
    - `rng_custom.h` = Driver for custom random number generator
    - `time_handler.h` = Driver for custom timer functionalities
    - `kms.h` = Key management system functionalities and interface
    - `numth_arith.h` = Library of number theory arithmetical operations on 32 bits
    - `eke_exp.h` = Library for Key Aagreement Protocol based on Encrypted Key Exchange
    - `aes256.h` = AES-256 cryptographic functionalities library
    - `sha256.h` = SHA-256 cryptographic hashing functionalities library
    - `hmac_sha256.h` = HMAC-SHA256 functionalities library
  - `src`
    - `Device`
      - `se3_core.c` = Management of communication flow and commands decoding
    - `com_channel.c` = Implementation of `com_channel.h`
    - `device_cmds.c` = Implementation of `device_cmds.h`
    - `nv_mem.c` = Implementation of `nv_mem.h`
    - `rng_custom.c` = Implementation of `rng_custom.h`
    - `time_handler.c` = Implementation of `time_handler.h`
    - `kms.c` = Implementation of `kms.h`
    - `numth_arith.c` = Implementation of `numth_arith.h`
    - `eke_exp.c` = Implementation of `eke_exp.h`
    - `aes256.c` = Implementation of `aes256.h`
    - `sha256.c` = Implementation of `sha256.h`
    - `hmac_sha256.c` = Implementation of `hmac_sha256.h`
- `host` ... Host-side C project, with related makefile
  - `include`
    - `com_channel.h` = Communication system driver and host interface
    - `host_cmds.h` = Host library for device commands API
  - `src`
    - `main.c` = Example wrapper to send commands from `host_cmds.h` to the device
    - `com_channel.c` = Implementation of `com_channel.h`
    - `host_cmds.c` = Implementation of `host_cmds.h`
- `install` ... Installation directory for the binaries of the devices and the host
- `Makefile` ... Makefile to automatically manage installation and execution of the project
- `README.md` ... Instructions about the project and how to install, run and clean it