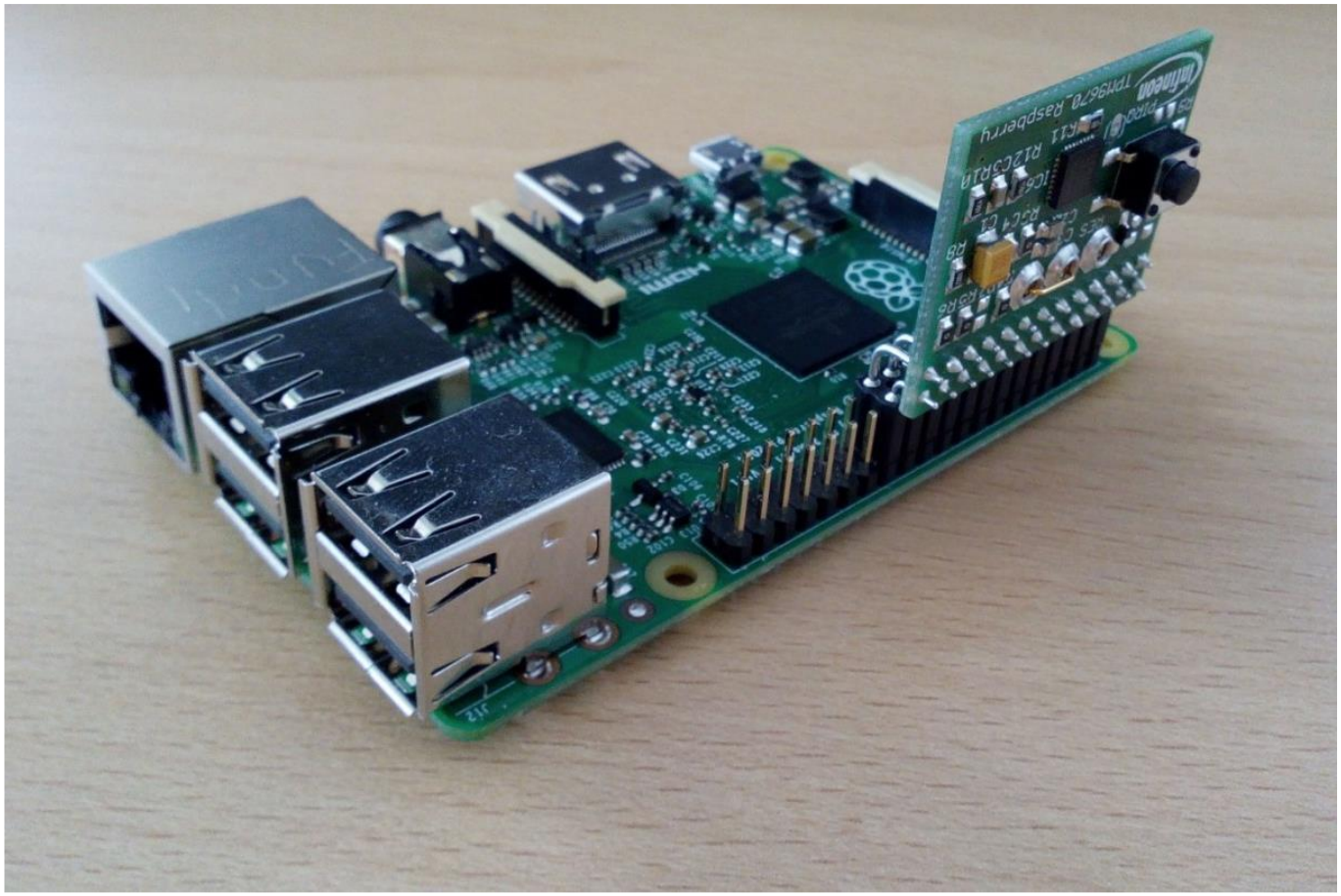
OPTIGA™ TPM Application Note

Integration of the Open Source TPM Software Stack 2.0 on a Raspberry Pi® 3 Linux environment with OPTIGA™ TPM SLx 9670 TPM 2.0

Devices

- OPTIGA™ TPM SLB 9670 TPM2.0
- OPTIGA™ TPM SLI 9670 TPM2.0
- OPTIGA™ TPM SLM 9670 TPM2.0



About this document

Scope and purpose

This document contains step-by-step instructions on how to setup the Open Source TPM Software Stack 2.0 (TSS 2.0) and related Software on a Raspberry Pi® 3 Linux environment to communicate with the Trusted Platform Module OPTIGA™ TPM SLx 9670 TPM2.0 from Infineon Technologies. In addition, software usage and programming examples are shown.

The OPTIGA™ TPM SLx 9670 TPM2.0 uses a SPI interface to communicate with the Raspberry Pi®. The OPTIGA™ TPM SLx 9670 TPM2.0 product family with SPI interface consists of 3 different products:

- OPTIGA™ TPM SLB 9670 TPM2.0 standard security applications
- OPTIGA™ TPM SLI 9670 TPM2.0 automotive security applications
- OPTIGA™ TPM SLM 9670 TPM2.0 industrial security applications

We refer with “OPTIGA™ TPM SLx 9670 TPM2.0” to all of the above 3 members of the OPTIGA™ TPM 2.0 product family with SPI interface.

OPTIGA™ TPM SLx 9670 TPM2.0 products are fully TCG compliant TPM products with Common Criteria (EAL4+) certification and FIPS 140 approval. The OPTIGA™ TPM SLx 9670 TPM2.0 products standard, automotive and industrial differ with regards to supported temperature range, lifetime, quality grades, test environment, qualification and reliability to fit the target applications requirements. For more details refer to Infineon’s website [5]. Since all OPTIGA™ TPM SLx 9670 TPM2.0 products are fully TCG compliant with regards to the SPI and software interface, the described steps to integrate a TPM 2.0 in a Raspberry Pi® 3 Linux environment are the same and valid for all variants. We are referring in this document also to the OPTIGA™ TPM SLx 9670 TPM2.0 by using simply “TPM” and to the Raspberry Pi® 3 by using “RPi”. An overview of all Infineon OPTIGA™ TPM products can be found on Infineon’s website [5]. More information about the TPM in general and how to integrate it into a platform can be found in the corresponding specifications of the Trusted Computing Group (TCG) in reference [6].

The described steps to integrate an OPTIGA™ TPM in a Raspberry Pi® 3 Linux environment can be performed with one of the Infineon Iridium SLx 9670 TPM2.0 SPI Boards, listed in the table below.

Iridium Boards:

Supported TPM	Order type	Order number:
OPTIGA™ TPM SLB 9670 TPM2.0	IRIDIUM 9670 TPM2.0	SP001596592
OPTIGA™ TPM SLI 9670 TPM2.0	IRIDIUM SLI 9670 TPM2.0	SP004232000
OPTIGA™ TPM SLM 9670 TPM2.0	IRIDIUM SLM 9670 TPM2.0	SP004232004

The 3 Infineon Iridium Boards are referred in the following as “Infineon Iridium SLx 9670 TPM2.0 SPI Board”

Intended audience

This document is intended for users who want to increase the security level of their embedded platforms using a OPTIGA™ TPM SLx 9670 TPM2.0 from Infineon Technologies in combination with the Open Source TPM Software Stack 2.0 (TSS 2.0) and like to evaluate how to start with the TPM software integration for their target applications. Furthermore several source code examples of typical use cases provide an introduction into the TPM API and the programming with the TSS 2.0.

Table of contents

About this document **2**
Table of contents **3**
List of figures **5**
List of tables **6**
Acronyms and Abbreviations **7**
1 The TPM Software Stack (TSS) **8**
2 Setup and Usage Environment **10**
3 Installed Software Overview **13**
4 TSS Setup **15**
4.1 General assumptions before installing 15
4.2 TPM Software Stack 2.0 (tpm2-tss) 16
4.2.1 Pre-Installation 16
4.2.2 Installation 16
4.2.3 Post Installation 17
4.2.4 FAPI Configuration 17
4.3 TPM2 Access Broker & Resource Manager (tpm2-tabrmd) 20
4.3.1 Pre-Installation 20
4.3.2 Installation 20
4.3.3 Post-Installation 21
4.3.4 Sanity Tests 21
4.4 TPM2 Tools (tpm2-tools) 21
4.4.1 Pre-Installation 21
4.4.2 Installation 21
4.4.3 Post-Installation 22
4.5 TPM2 TSS Engine (tpm2-tss-engine) 22
4.5.1 Pre-Installation 22
4.5.2 Installation 22
4.5.3 Post-Installation 23
4.5.4 Sanity-Tests 24
4.6 Cryptsetup (cryptsetup) 25
4.6.1 Pre-Installation 25
4.6.2 Installation 25
4.6.3 Sanity-Tests 26
4.7 TPM2 PKCS#11 (tpm2-pkcs11) 26
4.7.1 Pre-Installation 26
4.7.2 Installation 26
4.8 Setting the log level for the following usage and programming examples 27
5 TSS Usage **28**
5.1 TPM2 Tools 28
5.1.1 Create a random number 29
5.1.2 Sealing data to the TPM 29
5.1.3 Encryption and decryption 30
5.2 TPM2 TSS Engine 32
5.2.1 Create a random number 32
5.2.2 RSA decrypt 32
5.2.3 ECDSA sign and verify 34

Table of contents

5.3 Cryptsetup 35
5.3.1 NVRAM key slot 35
5.3.2 Additional key slot 37
5.4 TPM2 PKCS11 39
5.4.1 Creating random data using PKCS11 39
5.4.2 Token initialization and object creation 39
5.4.3 Sign and verify using PKCS11 41
5.4.4 Encrypt and decrypt using PKCS11 42
5.4.5 Change PIN and delete keys using PKCS11 43
6 TSS Programming 45
6.1 Basics 45
6.1.1 TSS documentation 45
6.1.2 Compiling code that uses the TSS FAPI library 46
6.1.3 Clearing the TPM NVRAM 47
6.1.4 General information about the programming examples 47
6.2 Programming Examples FAPI 48
6.2.1 Generation of various keys 48
6.2.1.1 Primary Key (createPrimaryKey_FAPI.c) 48
6.2.1.2 Signature Key (createSignatureKey_FAPI.c) 49
6.2.1.3 Encryption Key (createEncryptionKey_FAPI.c) 49
6.2.1.4 Restricted Key (createRestrictedKey_FAPI.c) 50
6.2.2 Signature generation and validation (createSignatureVerifyWithOpenSSL_FAPI.c) 51
6.2.3 Decryption of data using signature-based authorization (signatureAuthorization_FAPI.c) 53
6.2.4 Write to and read from NV-Index (readWriteNVIndexPasswordAuth_FAPI.c) 55
6.2.5 Secure Time (secureTime_FAPI.c) 56
6.2.6 Key Duplication (keyDuplication_FAPI.c) 57
6.2.7 Quote (executeQuote_FAPI.c) 58
6.3 ESAPI Usage and Programming 61
6.3.1 TPM2 Tools with ESAPI 61
6.3.1.1 Create a random number 61
6.3.1.2 Sealing data to the TPM 61
6.3.1.3 RSA encryption and decryption 64
6.3.2 Programming Examples ESAPI 66
6.3.2.1 Compiling code that uses the TSS ESAPI library 66
6.3.2.2 Generation of various keys 66
6.3.2.3 Signature generation and validation (createSignatureVerifyWithOpenSSL.c) 78
6.3.2.4 Decryption of data using signature-based authorization (signatureAuthorization.c) 81
6.3.2.5 Write to and read from NV-Index (readWriteNVIndexPasswordAuth.c) 86
6.3.2.6 Secure Time (secureTime.c) 88
6.3.2.7 Key Duplication (keyDuplication.c) 94
6.3.2.8 Quote (executeQuote.c) 99
7 References 102
8 Appendix 103
8.1 TPM Simulator 103
8.2 TCTI Loader 103
Revision history 105

List of figures

Figure 1 TSS architecture overview 9
Figure 2 For this guide Raspberry Pi OS (based on Debian 10) is used with kernel version 4.19 10
Figure 3 TPM2.0 SBL 9670 SPI on RPi 3 Model B 11
Figure 4 Command to check if TPM driver is enabled 11
Figure 5 Schematic overview of the setup. The numbers refer to the corresponding sections. 14
Figure 6 Command to check if current user was correctly added to the group tss 17
Figure 7 Sanity test for TSS Engine 24
Figure 8 Command tss2_getrandom() with tpm2-tools 29
Figure 9 Command tss2_unseal() with tpm2-tools (excerpt) 30
Figure 10 Command tss2_encrypt() and tss2_decrypt() with tpm2-tools (excerpt) 31
Figure 11 OpenSSL uses the TPM's hardware-based random number generator to generate an 8-Byte random number 32
Figure 12 Encrypt and decrypt data with the TPM OpenSSL Engine 33
Figure 13 Sign data and verify signature with the TPM OpenSSL Engine 34
Figure 14 Store volume key in the TPM's NVRAM 36
Figure 15 Add a new key slot for the encrypted partition 38
Figure 16 Using the TPM's hardware-based random number generator to generate an 8-Byte random number 39
Figure 17 Initialize a store, create a slot and token and create an object under the token 40
Figure 18 The tpm2-pkcs11 tool is used to sign some data that is then verified with OpenSSL 41
Figure 19 The tpm2-pkcs11 tool is used to decrypt some data that is encrypted with OpenSSL 42
Figure 20 The tpm2-pkcs11 tool is used to change the user PIN and delete an object 43
Figure 21 Main page of the documentation 46
Figure 22 Command tpm2_getrandom() with tpm2-tools 61
Figure 23 Command tpm2_unseal() with tpm2-tools (excerpt) 63
Figure 24 Command tpm2_rsaencrypt() and tpm2_rsadecrypt() with tpm2-tools (excerpt) 65
Figure 25 Using a encryption key with HMAC to implement use cases using only symmetric cryptography 73

List of tables

Table 1 Summary of used Hardware 10

Acronyms and Abbreviations

Acronym	Explanation
API	Application Programming Interface
ESAPI	Enhanced System API
EK	Endorsement Key
FAPI	Feature API
GPIO	General Purpose Input/Output
HTML	Hypertext Markup Language
IMA	Integrity Measurement Architecture
IPC	Inter-Process Communication
LUKS	Linux Unified Key Setup
NV-Index	Non-Volatile-Index
NVRAM	Non-Volatile Random Access Memory
OS	Operating System
PCR	Platform Configuration Register
RPi	Raspberry Pi
SAPI	System API
SPI	Serial Peripheral Interface
TCG	Trusted Computing Group
TCTI	TPM Command Transmission Interface
TPM	Trusted Platform Module
TSS	TPM Software Stack

1 The TPM Software Stack (TSS)

The TPM Software Stack (TSS) is a middleware software to support, improve and simplify TPM usage for developers and system integrators. In general, the TSS features a layered design to fit various systems and requirements from application developers. The layers provide different APIs with varying abstraction layers and different level of detail for the functionality. Depending on the chosen API level, its main features include carrying out concurrent access and session handling, resource and key management. Next to the user APIs, the TSS consists of other components the user APIs are relying on. These are the TPM Command Transmission Interface (TCTI), which is used for inter-process communication (IPC), and the TPM Access Broker and Resource Manager (TABRM), which handles concurrent access to the TPM. The different components and their relations are depicted in Figure 1.

The TSS implementation follows secure coding guidelines and provides full documentation as a programming reference. For development, it provides full debug and logging functionality on a selectable detail level and layer. The TSS is supported by various Linux distributions like Ubuntu, Fedora, RHEL and Arch Linux.

Overall, three user APIs are defined with increasing abstraction levels: System API (SAPI), Enhanced SAPI (ESAPI) and Feature API (FAPI). FAPI and ESAPI are high level APIs and are intended for application programming, while the low level SAPI is intended for resource-constrained devices (low-level programming).

FAPI is the highest-level API designed to cover 80% of the use cases, which can be implemented with low effort. To achieve easy usability and low programming overhead, a default configuration of parameters is provided, which are set to reasonable values for typical scenarios. For special scenarios, where functionality or parameters are not covered by FAPI, ESAPI and SAPI should be used, which provide one-to-one mappings of the entire TPM functionalities. As the second highest level API, ESAPI also provides some convenience functionality, e.g., built-in crypto support. The SAPI is the lowest level API mapping all of the TPM's functionality one-to-one to commands in the C world. SAPI does not provide any auxiliary functions like the higher-level APIs. On the one hand, SAPI gives the developer the most detailed control of parameters and is the most lightweight API, which is also usable in microcontroller environments; on the other hand, it is not as easy to use as the other APIs since the developer needs to implement basic functionalities by himself, like crypto support and session handling.

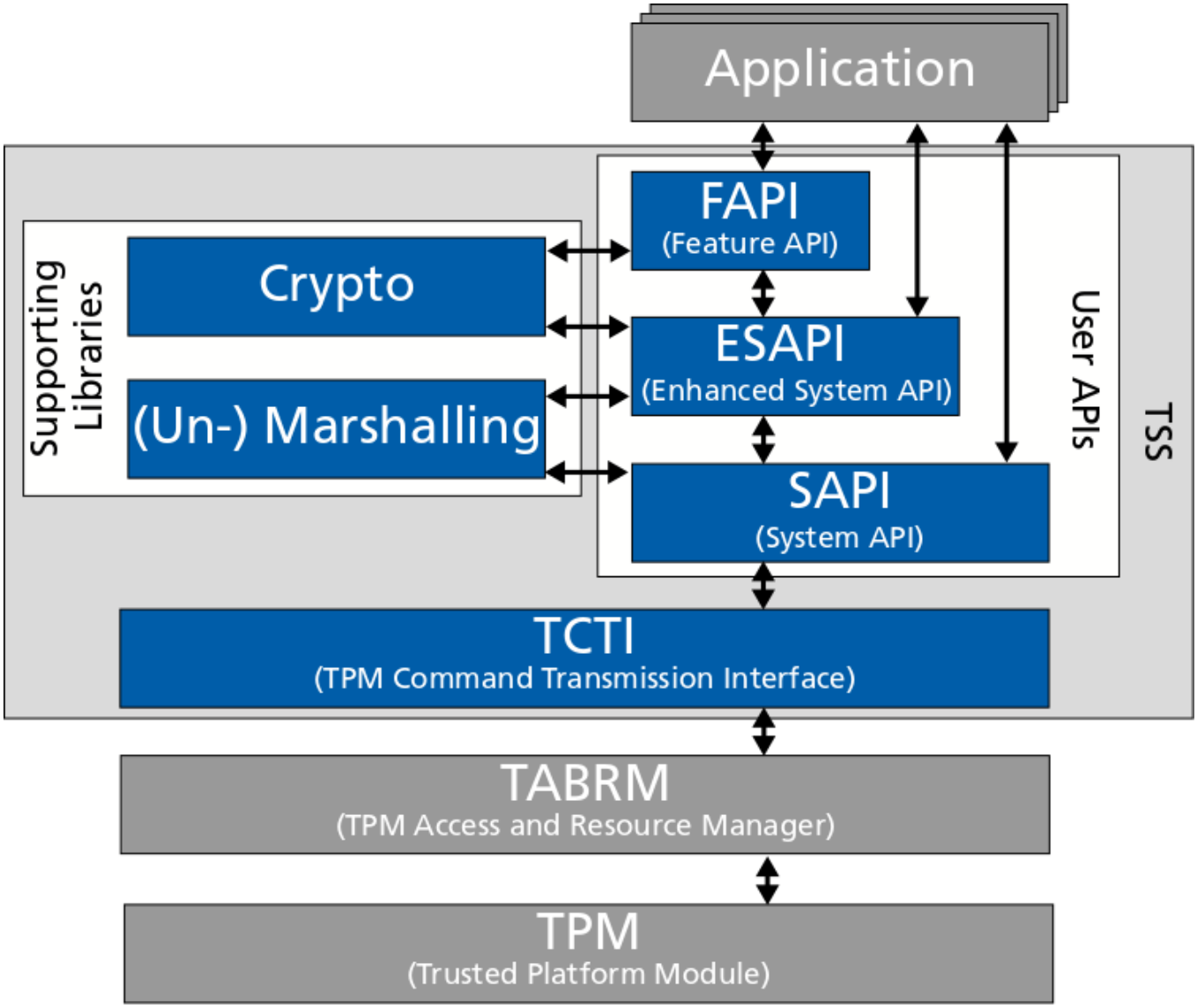


Figure 1 TSS architecture overview

2 Setup and Usage Environment

This guide was tested on a Raspberry Pi (RPi) 3 Model B with Raspberry Pi OS (32-bit) Lite, a minimal image based on Debian 10 (Buster), and kernel version 4.19 with an Infineon OPTIGA(TM) TPM 2.0 attached to the RPi's GPIOs (s. Figure 2 and Figure 3).

Table 1 shows a summary of the hardware used.

Table 1 Summary of used Hardware

Hardware	Version and Firmware/OS	Comment
Host PC	Arbitrary as long as SSH client is present	Only used to connect to the RPi.
TPM	TPM2.0 SLB 9670 SPI, Rev. 1.16	The TPM used is part of an Iridium 9670 add-on board [1], which next to the TPM features a 26-pin header for easy connection with RPi 1, 2 and 3.
RPi	Model 3 B, Raspberry Pi OS (based on Debian 10), Kernel 4.19	The Iridium board including the TPM is attached to the RPi's GPIO interface.

Note: Generally, it is possible to follow the instructions on x86-based platforms (instead of RPi's ARM) and the TPM simulator. Necessary adjustments are described throughout the Application Note in the specific section.

```
pi@raspberrypi:~ $ lsb_release -a
No LSB modules are available.
Distributor ID: Raspbian
Description:    Raspbian GNU/Linux 10 (buster)
Release:        10
Codename:       buster
pi@raspberrypi:~ $ uname -r
4.19.69-v7+
```

Figure 2 For this guide Raspberry Pi OS (based on Debian 10) is used with kernel version 4.19

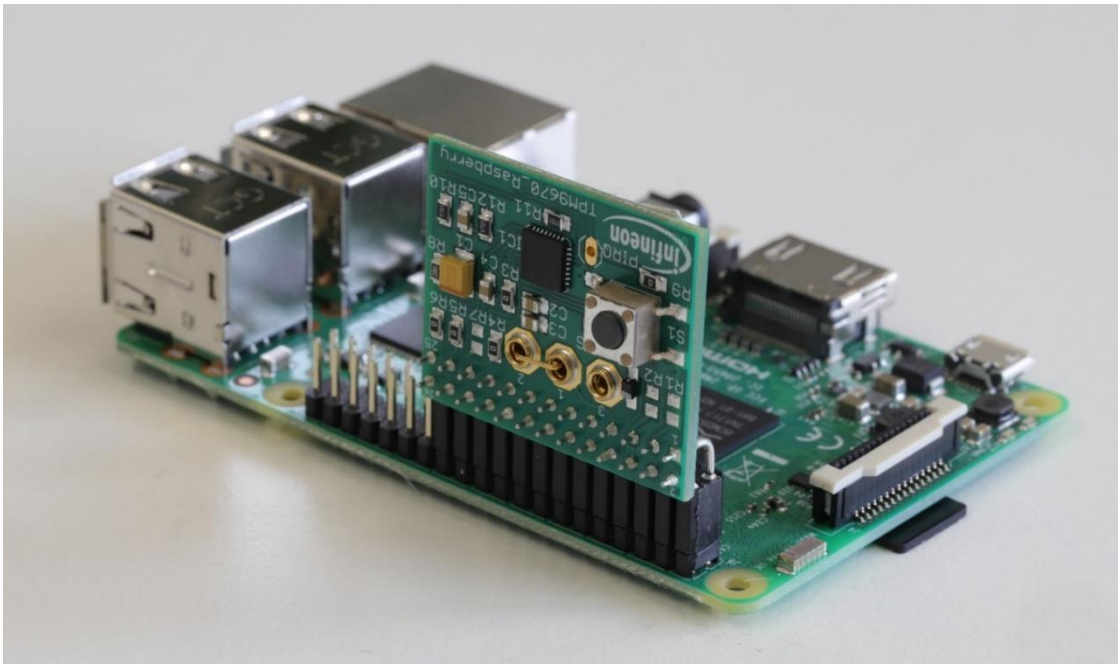


Figure 3 TPM2.0 SBL 9670 SPI on RPi 3 Model B

The corresponding Linux kernel on the RPi must have driver support for the TPM2.0. There are multiple ways to enable TPM2.0 driver support.

With current kernels versions starting from 4.14, the TPM2.0 driver can be enabled by executing the following commands on the RPi running Raspbian:

1. "`sudo rpi-update`" to update to the latest firmware version
2. Open the file "`/boot/config.txt`" with your favorite editor as super user
    a. Activate SPI by uncommenting the line "`dtparam=spi=on`"
    b. Load the TPM device tree overlay by adding the following line "`dtoverlay=tpm-slb9670`"
    c. Save and close the file
3. If not already done connect TPM and reboot
4. To verify if TPM2.0 driver is enabled, typing the command "`ls /dev/tpm0`" in the terminal window should return `/dev/tpm0` (cf., Figure 4).

```
pi@raspberrypi:~ $ ls /dev/tpm0
/dev/tpm0
```

Figure 4 Command to check if TPM driver is enabled

Note: Alternatively, the kernel can be compiled manually to enable the TPM2.0 driver. Detailed instructions on how to compile an own kernel with enabled TPM2.0 driver support is described in the Application Note "Getting Started with OPTIGA™ TPM 2.0 SLB 9670 and Raspberry Pi® 3" [2].

This guide was tested with the Infineon hardware-based TPM. For development and evaluation purposes, it is recommended to use the TSS also with a software-based TPM simulator that emulates the TPM API. The TPM simulator may be used only for evaluation, development and debugging purposes, because it does not provide any security for the keys and cryptographic operations. The TPM simulator allows a debugging into the internal

functions of the TPM, which is beneficial to find implementation improvements and bugs. The setup of the TPM simulator is explained in the appendix in Section 8.1.

3 Installed Software Overview

A complete overview of all the software installed within this guide is depicted in Figure 5. As seen in the figure, a hardware-based TPM and corresponding hardware drivers are used.

This guide focuses on installing five software components for use with the TPM. These are

1. *TPM Software Stack 2.0 (tpm2-tss)*
2. *TPM2 Access Broker & Resource Manager (tpm2-abrmd)*
3. *TPM2 Tools (tpm2-tools)*
4. *TPM2 TSS Engine (tpm2-tss-engine)*
5. *Cryptsetup (cryptsetup)*
6. *PKCS#11 (tpm2-pkcs11)*

The TPM Software Stack 2.0 (TSS 2.0) is an implementation of the Trusted Computing Group's (TCG) specification and consists of various Application Programming Interfaces (APIs) to communicate with the TPM on different abstraction layers.

Accordingly, the TPM2 Access Broker and Resource Manager (tpm2-tabrmd) implements the TCG's specification that manages the TPM context in a manner similar to a virtual memory manager. It swaps objects, sessions and sequences in and out of the limited TPM memory as needed. In general, it is an optional component. However, if not implemented, the upper layers will be responsible for TPM context management.

The TPM2 Tools are command-line tools that, if installed, will use the Feature API (FAPI) and Enhanced System API (ESAPI) of the TSS to communicate with the TPM or Resource Manager. The tools for both APIs are installed in parallel with different program names.

The TPM2 TSS Engine implements a cryptographic engine for OpenSSL for the TPM2.0 using the TPM2.0 TSS. As soon as the engine is loaded, the OpenSSL operations are executed using native TPM functionalities.

The cryptsetup extension described in this Application Note adds TPM2.0 support for disk encryption.

The PKCS#11 software uses the TPM2.0 device as cryptographic token according to the Public-Key Cryptography Standard PKCS#11. Thereby, PKCS#11 defines a standard method to access cryptographic services from tokens or devices, e.g., hardware security modules (HSM) or smart cards.

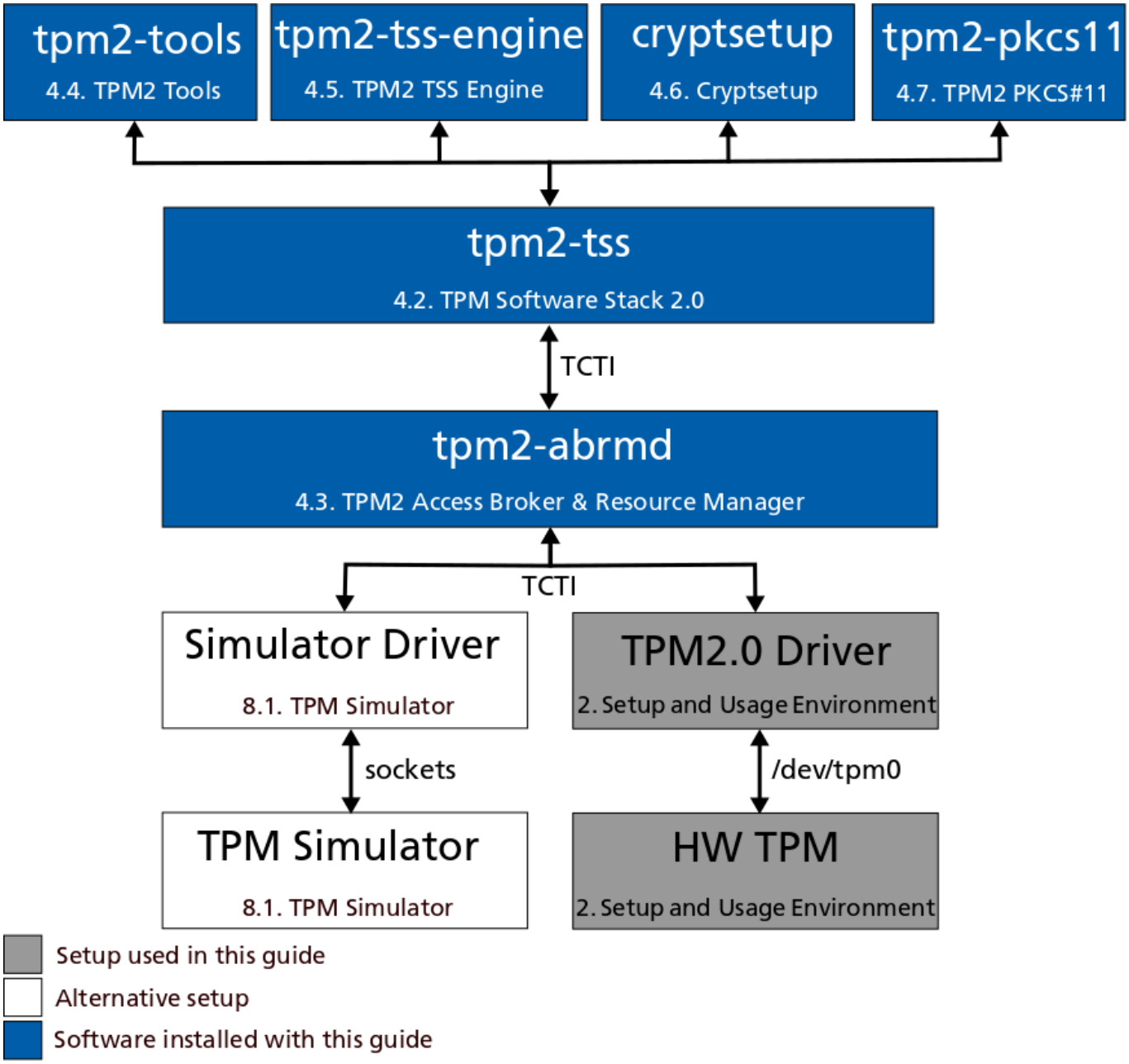


Figure 5 Schematic overview of the setup. The numbers refer to the corresponding sections.

4 TSS Setup

This chapter shows step-by-step how to setup the aforementioned projects. Up to three basic steps are needed to install each project. The first step sets up the build environment, in the second step the project is installed and the third step may involve some post-installation measures.

The pre-installation step is about setting up additional dependencies used in the second step for the build process. Mostly this will be about installing additional packages.

The installation step consists of five sub-steps. First, the repository containing the software will be downloaded. To guarantee reproducibility of this guide in the future either the "`-b`" parameter is used to fetch a stable bugfix-only branch of the software or the version is specified with the commit id. Second, a bootstrapping process is executed which will bring the downloaded source tree into a state for further processing. In a third step, various parameters for the build process can be set with the configure script. Afterwards, in the fourth step, the make command is used to compile the project with the previously defined parameters. Finally, the program is installed via the "`make install`" command.

In the post-installation step, the installed software is configured for use by other software. This might include announcing the installed binaries to the operating system.

For the following steps, we recommend to clone all repositories into the folder "`/home/pi/tssSetup`" and to put all sample codes into the folder "`/home/pi/codeExamples`". Usage examples may be executed in "`/home/pi/usageExamples`". When another directory structure is used, some of the presented commands have to be adapted in order to refer to the expected files. The recommended folder structure can be created with the command in l.001 shown in Code Listing 1. Additionally, it is necessary to create the user and group `tss` and add this user to the group (l.002).

Note: In all code listings within this Application Note, line breaks for commands that span multiple lines are completed with backslashes "`\`". This allows copying multiline commands directly into the terminal window. If the commands are typed in manually, backslashes at line ends must be omitted.

Code Listing 1 Create installation and code directories

```
001         mkdir /home/pi/tssSetup /home/pi/codeExamples \
   /home/pi/usageExamples
002         sudo useradd --system --user-group tss
```

4.1 General assumptions before installing

It is assumed that the distribution's package repository is up to date. This can be achieved as shown in Code Listing 2.

Code Listing 2 Update of the package repository

```
001         sudo apt update
```

For the steps after downloading the respective software via repository, it is assumed that the user is inside the root directory of the downloaded project.

4.2 TPM Software Stack 2.0 (tpm2-tss)

4.2.1 Pre-Installation

Before installing the software, some dependencies need to be fulfilled. These are installed as shown in Code Listing 3.

Code Listing 3 Installation of packages required to build the TSS

```
001         sudo apt -y install autoconf-archive libcmocka0 libcmocka-dev \
   procps iproute2 build-essential git pkg-config gcc libtool automake \
   libssl-dev uthash-dev autoconf doxygen libjson-c-dev libgcrypt20-dev \
   libcurl4-gnutls-dev
```

4.2.2 Installation

As seen in Code Listing 4, the software is downloaded from the repository. All upcoming examples are tested with TSS release version 3.0.0-rc1.

Code Listing 4 Change to the installation directory and download the TSS

```
001         cd /home/pi/tssSetup
002         git clone -b 3.0.x https://github.com/tpm2-software/tpm2-tss.git
```

The next step is bootstrapping (Code Listing 5).

Code Listing 5 Change to the source code directory and bootstrap the TSS

```
001         cd tpm2-tss
002         ./bootstrap
```

After bootstrapping, the build will be configured as shown in Code Listing 6. During installation, the software will generate a udev rule granting the system user `tss` permission to access the TPM device. For this rule, the installation path and the prefix will be adjusted with two parameters. For FAPI the flags `sysconfdir`, `localstatedir` and `runstatedir` will be set. The `sysconfdir` defines the location of the FAPI configuration and profile files. The `localstatedir` defines the location of the system keystore and `runstatedir` defines the location to store event logs.

Code Listing 6 Configuring the TSS

```
001         ./configure --with-udevrulesdir=/etc/udev/rules.d \
   --with-udevrulesprefix=70- --sysconfdir=/etc --localstatedir=/var \
   --runstatedir=/run
```

Once the configuration has been completed, the TSS can be compiled and installed as shown in Code Listing 7. During installation, the TSS will create a "`tpm2-tss`" file structure under the specified directories and create the default configuration and profile files.

Note: Since the RPi has very limited resources, executing the `make` call in Code Listing 7 with too many jobs in parallel (parameter `-j`) may result in virtual memory exhaustion errors.

Code Listing 7 Compiling and installing of the TSS

```
001         make
002         sudo make install
```

4.2.3 Post Installation

Once the installation is complete, udev needs to be instructed to reload its rules and apply the new rule. Finally, an update of the run-time bindings might be necessary. These steps are described in Code Listing 8

Code Listing 8 Post-installation steps for TSS

```
001  sudo udevadm control --reload-rules && sudo udevadm trigger
002  sudo ldconfig
003  cd /home/pi/tssSetup
```

Note: The documentation of the TSS is automatically generated since the doxygen package was installed as dependency before TSS installation. Details about the documentation are described in Section 6.1.1.

4.2.4 FAPI Configuration

When the TSS is installed, FAPI will read a configuration file to retrieve the information about necessary parameters. This is done by default without user interaction. The default lookup paths are defined during the installation with the parameters `sysconfdir`, `localstatedir` and `runstatedir`. Most directories are already created during installation. However, the user needs to create an own user keystore (Code Listing 9).

Code Listing 9 Creation of the user keystore

```
001  mkdir -p ~/.local/share/tpm2-tss/user/keystore
```

During installation, the default system keystore (`/var/lib/tpm2-tss/system/keystore`) should be created to be accessible by the group and user `tss`. During provision and for PCR-related commands, FAPI needs to store data either in the system keystore or the eventlog directory. To avoid permission denied errors, the user (pi) needs to be added to the group `tss` (Code Listing 10, l.001). A logout is necessary for the changes to take effect (Code Listing 10, l.002).

Code Listing 10 Adding current user to group tss

```
001  sudo adduser pi tss
002  logout
```

After logging in again, the command `groups` can be used to verify that the user is in the right group (Figure 6).

Figure 6 Command to check if current user was correctly added to the group tss

```
pi@raspberrypi:~ $ groups | grep tss
pi adm dialout cdrom sudo audio video plugdev games users input netdev tss gpio
i2c spi
```

Note: If the Endorsement Seed of a TPM was changed, FAPI cannot map the former Endorsement Certificates to the new Endorsement Keys (EK) anymore. FAPI will then produce an error message ("No EK certificate found"). This applies also when no Endorsement Certificates are present at all. To avoid this error, the line in brackets (`"ek_cert_less": "yes"`) has to be added to the config file at `/etc/tpm2-tss/fapi-config.json`. The adapted config file is shown in Code Listing 15. Please note the additional comma at the end of line 008.

Optionally, certain parameters (e.g., cryptographic parameters, used TCTI or keystore and log directories) can be customized by the user. This is described in the following.

Note: The following steps are optional and needed to customize FAPI initialization. If you are satisfied with the default parameters set during installation, you may skip the following part and continue with the next section.

To customize FAPI initialization, an own configuration file can be written and exported via environment variable. When using the configuration parameters in Code Listing 6, the default configuration file is located in `/etc/tpm2-tss/fapi-config.json` and shown in Code Listing 11. This configuration file can be copied to a customized location. This new configuration file can be edited and announced to the FAPI by exporting an environment variable (Code Listing 13) that points to the new configuration file. In this configuration file, a profile can be defined with `profile_name`. A profile defines cryptographic algorithms and primitives as well as key types and PCR selections. FAPI is installed with two exemplary profiles for RSA and ECC. By default the ECC profile is selected via the default configuration file. For example, to change the profile from ECC to RSA, the value of `P_ECCP256SHA256` needs to be replaced with `P_RSA2048SHA256` (cf., Code Listing 11 for ECC and Code Listing 12 for RSA). The contents of the ECC profile are shown in 4.3. With the configuration parameters used in Code Listing 6 it is located in `/etc/tpm2-tss/fapi-profiles/`. It can also be copied to a self-defined location and edited to meet custom needs. The path to the profile is defined with `profile_dir`. The user and system keystore path is defined by `user_dir` and `system_dir`. The TCTI interface that transmits commands from and to the TPM is defined with the `tcti` value. If it is empty, the TCTI is decided by TSS. Alternatively, the `tcti` value can be set to explicitely use the TPM with `"tcti"="device:/dev/tpm0"` or to use the TABRMD with `"tcti"="tabrmd"` (which may again use the device or simulator, c.f., Section 8.1). PCRs used for the Integrity Measurement Architecture (IMA) are defined with `system_pcrs` and the logging directory is defined with `log_dir`. In any case, if the paths defined in the new configuration file do not exist, they have to be created manually by the user.

Code Listing 11 Default FAPI Configuration File (with ECC selected)

```
{
    "profile_name": "P_ECCP256SHA256",
    "profile_dir": "/etc/tpm2-tss/fapi-profiles/",
    "user_dir": "~/.local/share/tpm2-tss/user/keystore",
    "system_dir": "/var/lib/tpm2-tss/system/keystore",
    "tcti": "",
    "system_pcrs" : [],
    "log_dir" : "/run/tpm2-tss/eventlog/”
}
```

Code Listing 12 Alternative FAPI Configuration File (with RSA selected)

```
{
    "profile_name": "P_RSA2048SHA256",
    "profile_dir": "/etc/tpm2-tss/fapi-profiles/",
    "user_dir": "~/.local/share/tpm2-tss/user/keystore",
    "system_dir": "/var/lib/tpm2-tss/system/keystore",
    "tcti": "",
    "system_pcrs" : [],
    "log_dir" : "/run/tpm2-tss/eventlog/”
}
```

Code Listing 13 Export of the environment variable

Code Listing 11 Default FAPI Configuration File (with ECC selected)

```
001         export TSS2_FAPICONF=/etc/tpm2-tss/fapi-config.json
```

Code Listing 14 Default ECC Profile (/etc/tpm2-tss/fapi-profiles/P_ECCP256SHA256.json)

```
001         {
002             "type": "TPM2_ALG_ECC",
003             "nameAlg":"TPM2_ALG_SHA256",
004             "srk_template": "system,restricted,decrypt,0x81000001",
005             "srk_description": "Storage root key SRK",
006             "srk_persistent": 0,
007             "ek_template":  "system,restricted,decrypt",
008             "ek_description": "Endorsement key EK",
009             "ecc_signing_scheme": {
010                 "scheme":"TPM2_ALG_ECDSA",
011                 "details":{
012                     "hashAlg":"TPM2_ALG_SHA256"
013                 },
014             },
015             "sym_mode":"TPM2_ALG_CFB",
016             "sym_parameters": {
017                 "algorithm":"TPM2_ALG_AES",
018                 "keyBits":"128",
019                 "mode":"TPM2_ALG_CFB"
020             },
021             "sym_block_size": 16,
022             "pcr_selection": [
023                { "hash": "TPM2_ALG_SHA1",
024                  "pcrSelect": [ ],
025                },
026                { "hash": "TPM2_ALG_SHA256",
027                  "pcrSelect": [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
028                  13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23 ]
029                }
030             ],
031             "curveID": "TPM2_ECC_NIST_P256",
032             "ek_policy": {
033                 "description": "Endorsement hierarchy used for policy
034                 secret.",
035                 "policy":[
036                     {
037                         "type":"POLICYSECRET",
038                         "objectName": "4000000b",
039                     }
040                 ]
041             }
042         }
```

Code Listing 15 Adapted FAPI Configuration File (only necessary if seed has changed or no certificates present)

```
001         {
002             "profile_name": "P_ECCP256SHA256",
003             "profile_dir": "/etc/tpm2-tss/fapi-profiles/",
004             "user_dir": "~/.local/share/tpm2-tss/user/keystore",
005             "system_dir": "/var/lib/tpm2-tss/system/keystore",
006             "tcti": "",
007             "system_pcrs" : [],
```

```
008             "log_dir" : "/run/tpm2-tss/eventlog/",
009             "ek_cert_less" : "yes"
010         }
```

4.3 TPM2 Access Broker & Resource Manager (tpm2-tabrmd)

4.3.1 Pre-Installation

Before installing the software, the dependencies shown in Code Listing 16 need to be installed.

Code Listing 16 Installation of packages required to build the TABRMD

```
001         sudo apt -y install libdbus-1-dev libglib2.0-dev
```

4.3.2 Installation

First, the software is downloaded from the repository. This is shown in Code Listing 17. All upcoming examples are tested with TABRMD release version 2.3.2.

Code Listing 17 Change to the installation directory and download the TABRMD source code

```
001         cd /home/pi/tssSetup
002         git clone -b 2.3.2 https://github.com/tpm2-software/tpm2-abrmd.git
```

The next step is bootstrapping as shown in Code Listing 18.

Code Listing 18 Change to the source code directory and bootstrap the TABRMD

```
001         cd tpm2-abrmd/
002         ./bootstrap
```

After bootstrapping, the build will be configured. For configuration, a few parameters are adjusted. The first parameter specifies the directory for the user `tss` to claim the D-Bus system bus. The second and third option specify the path to generate a system unit and preset file in order to start the daemon on system boot. Finally, the fourth parameter defines the path to install the D-Bus service files. The configuration process is shown in Code Listing 19.

Code Listing 19 Configuring the TABRMD

```
001         ./configure --with-dbuspolicydir=/etc/dbus-1/system.d \
    --with-systemdsystemunitdir=/lib/systemd/system \
    --with-systemdpresetdir=/lib/systemd/system-preset \
    --datarootdir=/usr/share
```

Once the configuration has been completed, the project can be compiled and installed as shown in Code Listing 20.

Note: Since the RPi has very limited resources, executing the `make` call in Code Listing 20 with too many jobs in parallel (parameter `-j`) may result in virtual memory exhaustion errors.

Code Listing 20 Building and installing the TABRMD

```
001         make
002         sudo make install
```

4.3.3 Post-Installation

Once the installation is complete, an update of the run-time bindings may be necessary. Additionally, the D-Bus daemon needs to be instructed to read the new configuration file and `systemd` needs to be triggered to reload, enable and start the new unit file. These steps are shown in Code Listing 21.

Note: If the TPM simulator is used, the commands to enable and start the `tpm2-abrmd.service` (l.4 and 5) should be omitted. Instead, the abrmd is started manually as described in Section 8.1 in the appendix.

Code Listing 21 Post-installation steps for TABRMD

```
001         sudo ldconfig
002         sudo pkill -HUP dbus-daemon
003         sudo systemctl daemon-reload
004         sudo systemctl enable tpm2-abrmd.service
005         sudo systemctl start tpm2-abrmd.service
006         cd /home/pi/tssSetup
```

4.3.4 Sanity Tests

Before continuing installing the other software, the following command can be executed to check if the tpm2-abrmd service has been installed successfully. This is shown in Code Listing 22.

Note: In case that the simulator is used, the service will not be enabled (cf., note in Section 4.3.3). Therefore, the following test is unnecessary and can be skipped.

Code Listing 22 Sanity check for TABRMD

```
001         dbus-send --system --dest=org.freedesktop.DBus --type=method_call \
   --print-reply /org/freedesktop/DBus org.freedesktop.DBus.ListNames
```

The returned array should contain the string `"com.intel.tss2.Tabrmd"`. If not, a reboot may be necessary.

4.4 TPM2 Tools (tpm2-tools)

4.4.1 Pre-Installation

Before installing the software, some additional dependencies need to be fulfilled as shown in Code Listing 23.

Code Listing 23 Installation of packages required to build the TPM2 Tools

```
001         sudo apt -y install autoconf automake libtool pkg-config \
   gcc libssl-dev libcurl4-gnutls-dev pandoc python-yaml expect
```

4.4.2 Installation

As a first step, the software is downloaded from the repository as shown in Code Listing 24. All upcoming examples are tested with TPM2 Tools release version 4.2.1.

Code Listing 24 Change to the installation directory and download the TPM2 Tools source code

```
001         cd /home/pi/tssSetup
002         git clone -b 4.2.X https://github.com/tpm2-software/tpm2-tools.git
```

The next step is bootstrapping. This is shown in Code Listing 25.

Code Listing 25 Change to the source code directory and bootstrap the TPM2 Tools

```
001         cd tpm2-tools/
002         ./bootstrap
```

After bootstrapping, the build will be configured. For configuration, no additional parameters need to be adjusted. This is shown in Code Listing 26.

Code Listing 26 Configuring the TPM2 Tools

```
001         ./configure
```

Once configuration has been done, the project can be compiled and installed. This is shown in Code Listing 27.

Note: Since the RPi has very limited resources, executing the `make` call in Code Listing 27 with too many jobs in parallel (parameter `-j`) may result in virtual memory exhaustion errors.

Code Listing 27 Building and Installing the TPM2 Tools

```
001         make
002         sudo make install
003         sudo ldconfig
004         cd /home/pi/tssSetup
```

4.4.3 Post-Installation

No post-installation steps are necessary.

4.5 TPM2 TSS Engine (tpm2-tss-engine)

4.5.1 Pre-Installation

Before installing the software, the dependencies shown in Code Listing 28 need to be fulfilled.

Code Listing 28 Installing the packages required to build the TSS Engine

```
001         sudo apt -y install build-essential autoconf automake m4 \
   libtool gcc pkg-config libssl-dev ruby-ronn pandoc expect
```

4.5.2 Installation

First, the software has to be downloaded from the repository (Code Listing 29). All upcoming examples are tested with TSS Engine release version v1.1.0-rc1.

Code Listing 29 Changing the installation directory and downloading the TSS Engine source code

```
001         cd /home/pi/tssSetup
002         git clone -b v1.1.x https://github.com/tpm2-software/\
```

Code Listing 29 Changing the installation directory and downloading the TSS Engine source code

```
      tpm2-tss-engine.git
```

The next step is bootstrapping (Code Listing 30).

Code Listing 30 Change to the source code directory and bootstrap the TSS Engine

```
001       cd tpm2-tss-engine/
002       ./bootstrap
```

After bootstrapping, the build will be configured (Code Listing 31). For configuration, no additional parameters need to be adjusted.

Code Listing 31 Configuring the TSS Engine

```
001       ./configure
```

Once the configuration has been completed, the project can be compiled and installed as shown in Code Listing 32.

Code Listing 32 Building and installing the TSS Engine

```
001       make -j
001       sudo make install
002       sudo ldconfig
003       cd /home/pi/tssSetup
```

4.5.3 Post-Installation

After the installation is complete, the new module needs to be registered with OpenSSL and an update of the run-time bindings may be necessary.

Therefore, first use your favorite editor, e.g., nano, to create a file named `engines.conf` with the content "`/usr/lib/arm-linux-gnueabihf/engines-1.1`" and save it in `/etc/ld.so.conf.d/`. Afterwards, the command in Code Listing 33 needs to be executed.

Code Listing 33 Post-installation steps for the TSS Engine

```
001       sudo ldconfig
```

These steps should complete the TSS engine setup needed for all examples within this Application Note. The remainder of this section shows how to set up an engine configuration file, which is only needed if no "`-engine`" parameter is required for example when using libmqtt with SSL. However, this is not part of this Application Note and only for information.

Note: The following steps are optional. They are not needed for the examples shown in this Application Note and are therefore only for information. You may skip the following part and continue with the next section.

To use the TSS Engine also with applications that do not use a special engine parameter, a config file needs to be created and its location exported via an environment variable.

First, create a config file with your favorite editor, e.g., nano, with the content of Code Listing 34 and store it at a distinct location. For this Application Note the storage path will be "`/home/pi/tssSetup/tpm2-tss-`

engine/" and the file name will be "tpm2-tss-engine.conf". Among other things, the config file defines the name of the engine and the path to the respective library.

Code Listing 34 Configuration file "tpm2-tss-engine.conf"

```
001         openssl_conf = openssl_init
002
003         [openssl_init]
004         engines = engine_section
005
006         [engine_section]
007         tpm2 = tpm2_section
008
009         [tpm2_section]
010         engine_id = tpm2tss
011         dynamic_path = /usr/lib/arm-linux-gnueabihf/engines-1.1/
   libtpm2tss.so
012         default_algorithms = ALL
013         init = 0
```

Afterwards, the file is registered with OpenSSL via `export OPENSSL_CONF`. The path in Code Listing 35 needs to be adjusted accordingly.

Every time the engine should be used the environment variable "`OPENSSL_CONF`" needs to be set as described in Code Listing 35. Keep in mind though that the path above has to point to the previously created tpm2-tss-engine.conf file. Otherwise, the command will fail.

Code Listing 35 Export command with exemplary path

```
001         export OPENSSL_CONF=/home/pi/tssSetup/tpm2-tss-engine/\
   tpm2-tss-engine.conf
```

Note: To check if the variable has already been set, the command "`echo $OPENSSL_CONF`" should return the path to the config file.

Alternatively, the environment variable can be set permanently within a file to avoid having to set the variable manually ever again. This can be done by writing "`OPENSSL_CONF=/home/pi/tssSetup/tpm2-tss-engine/tpm2-tss-engine.conf`" in the file at "`/etc/environment`". A reboot is necessary for the changes to come into effect.

4.5.4 Sanity-Tests

The following command can be executed to check if the tpm2-tss-engine has been installed successfully.

Code Listing 36 Sanity test for the TSS Engine

```
001         openssl engine -t -c tpm2tss
```

It should retrieve the engine information about the name and the available functions as shown in Figure 7.

```
pi@raspberrypi:~ $ openssl engine -t -c tpm2tss
(tpm2tss) TPM2-TSS engine for OpenSSL
 [RSA, RAND]
     [ available ]
```

Figure 7 Sanity test for TSS Engine

4.6 Cryptsetup (cryptsetup)

Note: In this AppNote, only a work proposal for a TPM-based cryptsetup extension is described. An official release is expected with the 2.4.0 release of cryptsetup. It will feature a more flexible approach where the TPM is added as module. Thus, handling may be different to the proposal described here.

4.6.1 Pre-Installation

Before installing the software, the dependencies shown in Code Listing 37 need to be fulfilled.

Code Listing 37 Installation of packages required to build cryptsetup

```
001        sudo apt -y install autopoint uuid-dev libdevmapper-dev \
   libpopt-dev cryptsetup libgcrypt20-dev libjson-c-dev gettext python-dev
```

4.6.2 Installation

First, the software is downloaded from the repository as shown in Code Listing 38.

Code Listing 38 Change to the installation directory and download the cryptsetup source code

```
001        cd /home/pi/tssSetup
002        git clone https://gitlab.com/AndreasFuchsSIT/\
   cryptsetup.git
```

Currently, there are no tags available, so a certain version is checked out using the commit id within the next step. Afterwards, bootstrapping is done with the autogen script as shown in Code Listing 39.

Code Listing 39 Change to the source code directory and bootstrap cryptsetup

```
001        cd cryptsetup
002        git checkout 15c28319
003        ./autogen.sh
```

After bootstrapping the build will be configured. For configuration purposes, some additional parameters are passed that define, e.g., installation paths, that the software should be built as a shared library and that TPM support should be enabled. These steps are shown in Code Listing 40.

Note: For platforms other than the RPi's armhf, the library path needs to be adjusted during configuration according to the platform, e.g., `--libdir=/lib/x86_64-linux-gnu/` for x86 64-bit platforms.

Code Listing 40 Configuring cryptsetup

```
001        ./configure --prefix=/usr/ --libdir=/lib/arm-linux-gnueabihf/ \
   --sbindir=/sbin/ --mandir=/usr/share/man --enable-shared \
   --enable-cryptsetup-reencrypt --enable-tpm2
```

After the configuration has been completed, the project can be compiled and installed as depicted in Code Listing 41.

Code Listing 41 Building and installing the cryptsetup

```
001         make -j
002         sudo make install
003         cd /home/pi/tssSetup
```

4.6.3 Sanity-Tests

Cryptsetup must use the installed libcryptsetup version instead of an already installed one. Therefore, use `ldd` to check which libcryptsetup path is used (e.g., /lib/arm-linux-gnueabihf/libcryptsetup.so.12). In the discovered path check with `ls -l` if it is pointing to the correct library version (/lib/arm-linux-gnueabihf/libcryptsetup.so.12 -> libcryptsetup.so.12.3.0). If this is correct, you can skip the command in l.003 and continue with the next chapter.

Otherwise, link libcryptsetup.so.12 to libcryptsetup.so.12.3.0 by executing l.003.

Code Listing 42 Check used library version

```
001         ldd /sbin/cryptsetup | grep libcryptsetup
002         ls -l /lib/arm-linux-gnueabihf/libcryptsetup.so.12
003         sudo ln -sf libcryptsetup.so.12.3.0 \
   /lib/arm-linux-gnueabihf/libcryptsetup.so.12
```

4.7 TPM2 PKCS#11 (tpm2-pkcs11)

4.7.1 Pre-Installation

Before installing the software, the dependencies shown in Code Listing 43 need to be fulfilled.

Code Listing 43 Installation of packages required to build tpm2-pkcs11

```
001         sudo apt -y install libsqlite3-dev libyaml-dev opensc
```

4.7.2 Installation

First, the software is downloaded from the repository as shown in Code Listing 44. All upcoming examples are tested with TSS Engine release version 1.3.1-RC0.

Code Listing 44 Change to the installation directory and download the tpm2-pkcs11 source code

```
001         cd /home/pi/tssSetup
002         git clone -b 1.X https://github.com/tpm2-software/\
   tpm2-pkcs11.git
```

Currently, there are no tags available, so a certain version is checked out using the commit id within the next step. Afterwards, bootstrapping is done with the autogen script as shown in Code Listing 45.

Code Listing 45 Change to the source code directory and bootstrap tpm2-pkcs11

```
001         cd tpm2-pkcs11
002         ./bootstrap
```

After bootstrapping the build will be configured. These steps are shown in Code Listing 46.

Code Listing 46 Configuring tpm2-pkcs11

```
./configure
```

After the configuration has been completed, the project can be compiled and installed as depicted in Code Listing 47.

Code Listing 47 Building and installing tpm2-pkcs11

```
make
sudo make install
sudo ldconfig
cd /home/pi/tssSetup
```

4.8 Setting the log level for the following usage and programming examples

The TSS and related software feature various fine-granular log levels that can be specified at runtime for all or only specific components by an environment variable. Possible levels are `NONE`, `ERROR`, `WARNING`, `INFO`, `DEBUG`, `TRACE`. The default log level is `WARNING`. In this Application Note a runtime log level of `NONE` is used for all components to avoid distracting message output. Code Listing 48 shows how to set the log level to `NONE`.

Code Listing 48 Specification of the runtime log level

```
export TSS2_LOG=all+NONE
```

5 TSS Usage

Within this chapter exemplary usage of various applications are shown that use the TSS to communicate with the TPM.

All usage examples are expected to be executed within "`home/pi/usageExamples/`". To avoid interference among different examples, it is recommended to clean the `usageExamples` directory after an example has been executed. This can be done for example with "`rm -rf /home/pi/usageExamples/*`".

Note: To avoid distracting message output, the log level should be set as described in Section 4.8.

5.1 TPM2 Tools

This chapter shows some examples on how to use the TPM2 Tools (tpm2-tools), which are command line tools to execute basic TPM functionalities. In the examples in this document, only the most relevant parameters of the tools will be explained. More detailed information about the tools and the parameters can be found in the respective man pages [3], e.g., "`man tpm2_getrandom`". For convenience, the used commands are listed in addition as text for easier copy & paste.

There are different TPM2 Tools for ESAPI and FAPI, which use the respective API to execute the TPM functionality. The TPM2 Tools commands starting with "`tpm2_`" are using ESAPI functions and commands starting with "`tss2_`" are using FAPI functions.

This section describes the usage of the TPM2 Tools for FAPI, which provide generic methods to create keys and perform cryptographic operations with the TPM using the keys stored in the FAPI metadata store. If a special usage of the TPM with specific parameters is required, the ESAPI can be used to specify the TPM command parameters in detail. The usage of the ESAPI is described in Appendix 0.

Note: To successfully execute the following examples, FAPI needs to be initialized as described in chapter 4.2.4. In short: When using the standard profiles and configuration file, at least the user storage has to be created. If not already done, the command in Code Listing 9 can be used. Otherwise, in case of a custom configuration file and/or profile, all custom paths need to be existing and set correctly in the custom configuration file. If they are not existing yet, they need to be created by the user.

Note: For the following examples to work, `tss2_provision` needs to be executed once to initialize the keystore and create a persistent EK (Code Listing 49). The successful execution of the commands can be confirmed with `tss2_list`, which prints the created provisioning keys and their paths in the FAPI metadata store.

Code Listing 49 Template for command tss2_provision with tpm2-tools

```
001         tss2_provision
```

Note: FAPI reports an error if there is an issue with the EK certificate, e.g., if no certificate is present at all (`fapi:No certificate`). To still continue with provisioning, the config file need to be adjusted with the `ek_cert_less` option as described in section 4.2.4 and shown in Code Listing 15.

5.1.1 Create a random number

The `tss2_getrandom` command is used to get a random value from the TPM's hardware random number generator. The length of the random number can be defined using parameters. The retrieved value is printed either to stdout or to a file.

While terminal inputs and outputs are depicted in Figure 8, the template shown in Code Listing 50 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ tss2_getrandom --numBytes 4 --data - | hexdump
-C
00000000  27 22 24 34 0a                                    |'"$4.|
00000005
pi@raspberrypi:~/usageExamples $ tss2_getrandom --numBytes 8 --data - | hexdump
-C
00000000  1a 20 59 21 8e 58 c8 86  0a                       |. Y!.X...|
00000009
pi@raspberrypi:~/usageExamples $ tss2_getrandom --numBytes 4 --data random.out
pi@raspberrypi:~/usageExamples $ hexdump -C random.out
00000000  a5 25 94 2e                                       |.%..|
00000004
```

Figure 8 Command tss2_getrandom() with tpm2-tools

Code Listing 50 Template for command tss2_getrandom() with tpm2-tools

```
001  cd /home/pi/usageExamples
002  tss2_getrandom --numBytes 4 --data - | hexdump -C
003  tss2_getrandom --numBytes 8 --data - | hexdump -C
004  tss2_getrandom --numBytes 4 --data random.out
005  hexdump -C random.out
```

5.1.2 Sealing data to the TPM

With the TPM, data can be encrypted and then sealed to the platform state, specifically to the TPM's PCR values. This means that the sealed data can be unsealed (decrypted) successfully only if the PCR values are identical to the state when the data was sealed.

To demonstrate this, first the data file "data.File" containing data to be sealed ("secret data") is created. Next, sealing key is defined with a password ("def456") and the data to-be-sealed from the file "data.File".
The now successfully sealed data should be unsealed again. Therefore, the `unseal()` command is executed. Parameters are the key path and a file to write the output to. A successful unsealing can be examined using `hexdump`. Finally, the key is deleted again.

The sealing method of the TPM2-tools supports data sizes up to 128 Byte. In order to perform sealing of larger bulk data, the method of hybrid cryptography can be used. This method involves a new AES key, which is used for the bulk data encryption external of the TPM. The AES key is then sealed to the TPM as the low size of the AES key fits to the data size limit of the TPM.

While terminal inputs and outputs are depicted in Figure 9, the template shown in Code Listing 51 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo "secret data" > data.File
pi@raspberrypi:~/usageExamples $ tss2_createseal --path HS/SRK/mySealKey --authV
alue def456 \
> --type "noDa" --data data.File
pi@raspberrypi:~/usageExamples $ tss2_unseal --path HS/SRK/mySealKey --data unse
aled.data
Authorize object :
pi@raspberrypi:~/usageExamples $ hexdump -C unsealed.data
00000000  73 65 63 72 65 74 20 64  61 74 61 0a              |secret data.|
0000000c
pi@raspberrypi:~/usageExamples $ tss2_delete --path HS/SRK/mySealKey
```

Figure 9 Command tss2_unseal() with tpm2-tools (excerpt)

Code Listing 51 Template for command tss2_unseal() with tpm2 tools

```
001         cd /home/pi/usageExamples
002         echo "secret data" > data.File
003         tss2_createseal --path HS/SRK/mySealKey --authValue def456 \
   --type "noDa" --data data.File
004         tss2_unseal --path HS/SRK/mySealKey --data unsealed.data
005         hexdump -C unsealed.data
006         tss2_delete --path HS/SRK/mySealKey
```

5.1.3 Encryption and decryption

Note: At the time of creation of this document, FAPI only supports the use of RSA keys for encryption. Therefore, the profile needs to set to use RSA as described in section 4.2.4 and shown in Code Listing 12. The support of encryption with ECC is under discussion in the TSS software community, so that it might be added in a future release (e.g. version 3.1). It is then recommended to switch to ECC for the encryption of data.

With the command `tss2_decrypt()` the TPM decrypts data. As in the example before, the data file “data.File” is created containing the sensitive data string “secret data”. The method of encryption described in this document supports data sizes for encryption in RSA OAEP mode up to 190 Byte. In order to perform encryption of larger bulk data, the method of hybrid cryptography can be used. This method involves a new AES key, which is used for the bulk data encryption external of the TPM. The AES key is then sealed to the TPM as the low size of the AES key fits to the data size limit of the TPM.

First, a primary key is created with `tss2_provision` under which an encryption key is created. The type of the encryption key is specified in the FAPI profile. However, as described above, currently only RSA keys are supported in FAPI for encryption. The encryption key has the passphrase “abc” as authorization value, which will be required by the TPM to allow the usage of the private key later on. The authorization value is a secret, which protects the access to the key from unauthorized party. The created encryption key is used with the command `tss2_encrypt()` to encrypt the file “data.File”. This operation uses the public part of the encryption key. A successful encryption can be evaluated “hexdumping” the encrypted file. Next to the resulting ciphertext, this file also contains (encrypted) data that is needed by the TPM to decrypt the ciphertext again. Therefore, this file gets quite large. Thus, only the first 40 bytes of the resulting encrypted file are printed in the screenshot. The ciphertext starts right after the key name (“b53a997c”).

After the data has been encrypted successfully, it is decrypted again using the private part of the encryption key, which is stored inside the TPM. Therefore, the command `tss2_decrypt()` is used with the encrypted file

as input and the passphrase to authorize the usage of the private key. The successful decryption of the resulting file “decrypted.File” can be verified via `hexdump`.

While terminal inputs and outputs are depicted in Figure 10, the template shown in Code Listing 52 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo "secret data" > data.File
pi@raspberrypi:~/usageExamples $ tss2_createkey --path HS/SRK/myCryptKey --type
"noDa, decrypt" --authValue abc
pi@raspberrypi:~/usageExamples $ tss2_encrypt --keyPath HS/SRK/myCryptKey --plai
nText data.File --cipherText encrypted.File
pi@raspberrypi:~/usageExamples $ hexdump -C encrypted.File
00000000  72 f7 f3 95 cb a6 e1 2c  3f 15 bd 3a ce 82 a3 dc  |r......,?..:....|
00000010  9b 6c 99 18 5e b7 ce 72  a2 8d ff 2c 1e 97 dc 40  |.l..^..r...,...@|
00000020  10 5e ff f9 f1 e4 40 6a  94 43 26 76 41 7a be 62  |.^....@j.C&vAz.b|
00000030  6c 83 d6 ea 93 c5 83 3d  cf 6d 3f a9 8b 91 4c 2b  |l......=.m?...L+|
00000040  fd d5 14 59 cd 0a 37 2b  cf f1 f0 fb 29 24 2d cb  |...Y..7+....)$-.|
00000050  fb 37 f4 0d 5c bb fe 29  17 93 22 fb 6c 78 39 1c  |.7..\..)..".lx9.|
00000060  f6 c4 b6 c6 b7 db 3d 3a  f3 8f 9d ed 59 a3 b3 28  |......=:....Y..(|
00000070  ff 86 fd b6 9d 14 bf 8c  75 2d ab d7 69 6f a0 a0  |........u-..io..|
00000080  67 e7 0a 6e dd a6 38 84  cf 6d d4 d6 f2 de ba dd  |g..n..8..m......|
00000090  53 8c aa 39 04 c4 99 b4  23 56 68 8f f4 0e a0 72  |S..9....#Vh....r|
000000a0  8a d7 a7 cc 72 48 8b 86  dc 93 d0 07 15 df 89 ea  |....rH..........|
000000b0  14 74 6c 9f e3 29 64 2f  8e e4 90 22 6b ba b5 e5  |.tl..)d/..."k...|
000000c0  ae a4 42 d8 fb 53 f2 ce  9c 4d 9d 1f b9 d7 61 9c  |..B..S...M....a.|
000000d0  1a e0 23 4d ff 36 9d ed  dd 7f 77 49 7a 17 ca 10  |..#M.6....wIz...|
000000e0  19 f0 fc ed 4e ce 75 1c  f1 25 67 f8 6e 7d e8 13  |....N.u..%g.n}..|
000000f0  f7 c6 3a ba c0 69 cc 0d  28 dc 9f c0 7c 28 af f5  |..:..i..(...|(..|
00000100
pi@raspberrypi:~/usageExamples $ tss2_decrypt --keyPath HS/SRK/myCryptKey --ciph
erText encrypted.File --plainText decrypted.File
Authorize object :
pi@raspberrypi:~/usageExamples $ hexdump -C decrypted.File
00000000  73 65 63 72 65 74 20 64  61 74 61 0a              |secret data.|
0000000c
pi@raspberrypi:~/usageExamples $ tss2_delete --path HS/SRK/myCryptKey
```

Figure 10 Command tss2_encrypt() and tss2_decrypt() with tpm2-tools (excerpt)

Code Listing 52 Template for command tss2_rsaencrypt() and tss2_rsadecrypt() with tpm2-tools

```
001         cd /home/pi/usageExamples
002         echo "secret data" > data.File
003         tss2_createkey --path HS/SRK/myCryptKey \
   --type "noDa, decrypt" --authValue abc
004         tss2_encrypt --keyPath HS/SRK/myCryptKey --plainText data.File \
   --cipherText encrypted.File
005         hexdump -C encrypted.File
006         tss2_decrypt --keyPath HS/SRK/myCryptKey \
   --cipherText encrypted.File --plainText decrypted.File
007         hexdump -C decrypted.File
008         tss2_delete --path HS/SRK/myCryptKey
```

5.2 TPM2 TSS Engine

Generally, OpenSSL can be instructed to use a certain engine to execute commands. This is done with the parameter `engine` and the value `tpm2tss`.

5.2.1 Create a random number

With specifying the TPM as an engine for OpenSSL, OpenSSL can use the TPM's hardware random number generator to retrieve random values.

While terminal in- and outputs are depicted in Figure 11, the template shown in Code Listing 53 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ openssl rand -engine tpm2tss -hex 8
engine "tpm2tss" set.
bbf0aa4d3afd951e
```

Figure 11 OpenSSL uses the TPM's hardware-based random number generator to generate an 8-Byte random number

Code Listing 53 Template for command rand with OpenSSL engine

```
001         cd /home/pi/usageExamples
002         openssl rand -engine tpm2tss -hex 8
```

5.2.2 RSA decrypt

Note: At the time of creation of this document, TPM2 TSS Engine only supports the use of RSA keys for encryption. The support of encryption with ECC might be added in a future release. It is then recommended to switch to ECC for the encryption of data.

In this example, the TPM is used as an engine for OpenSSL to decrypt a cipher text.

First, the file is created that will be encrypted and later decrypted ("plain.data"). Next, the TPM is used to create a 2048-bit RSA key pair ("key.pair"). The corresponding secret key is protected by a passphrase ("abc"). The key pair is loaded into the TPM and the public key is exported ("key.pub"). It is used to encrypt the plain data file.

After the file has been encrypted successfully, OpenSSL's TPM engine is used for decryption. Therefore, the passphrase has to be entered. A final `hexdump` reveals the decrypted message.

While terminal in- and outputs are depicted in Figure 12, the template shown in Code Listing 54 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo "secret text" > plain.data
pi@raspberrypi:~/usageExamples $ tpm2tss-genkey -a rsa -s 2048 -p abc key.pair
pi@raspberrypi:~/usageExamples $ openssl rsa -engine tpm2tss -inform engine -in
key.pair -pubout -outform pem -out key.pub
engine "tpm2tss" set.
Enter password for user key:
writing RSA key
pi@raspberrypi:~/usageExamples $ openssl pkeyutl -pubin -inkey key.pub -in plain
.data -encrypt -out encrypted.data
pi@raspberrypi:~/usageExamples $ hexdump -C encrypted.data
00000000  34 ec f1 a0 69 f8 5b 49  80 dc e2 b9 49 8f 0a 3b  |4...i.[I....I..;|
00000010  c2 98 4a cd 99 17 45 e4  8c 7f 38 eb c1 09 90 6c  |..J...E...8....l|
00000020  ca 23 56 38 53 38 e5 b2  0c 79 c9 c9 45 01 cf a1  |.#V8S8...y..E...|
00000030  32 07 a4 02 8f 3c 13 52  f5 c2 1f e8 ed 26 60 ce  |2....<.R.....&`.|
00000040  66 f8 f8 db 23 0d 39 b7  99 d9 25 1c 75 b6 6c 6c  |f...#.9...%.u.ll|
00000050  be 97 e0 a8 df e9 af 64  bd 72 17 e1 67 e4 60 06  |.......d.r..g.`.|
00000060  1e 9a 4b e5 a6 0d 0d 61  5e 6a ab 44 e0 3b 4b c5  |..K....a^j.D.;K.|
00000070  c9 b0 64 a0 20 45 ee 91  25 1f c0 f5 c2 14 b2 cd  |..d. E..%.......|
00000080  e2 10 85 0a d4 84 b3 04  3d 02 f5 08 45 9f 97 b8  |........=...E...|
00000090  60 ad ed 52 88 0c b7 6a  55 de 62 2c 78 34 82 d3  |`..R...jU.b,x4..|
000000a0  73 2d fe 01 c1 54 c9 4c  1e 78 3e ba 0a 0d a8 0c  |s-...T.L.x>.....|
000000b0  e6 f0 ca 5e 95 01 e6 7d  08 3f fb f3 cb 30 c5 ed  |...^...}.?...0..|
000000c0  ac cb 11 f2 12 d9 f3 dd  e1 06 58 3b 09 32 38 56  |..........X;.28V|
000000d0  61 a9 a6 23 55 8e 91 35  2a b2 5b e7 18 b4 49 8e  |a..#U..5*.[...I.|
000000e0  ff 1b a4 24 7c 75 e3 18  0b 0a 42 2d 19 2b 59 e4  |...$|u....B-.+Y.|
000000f0  f8 d8 35 71 f8 3f 32 d0  85 85 07 86 a6 55 c8 04  |..5q.?2......U..|
00000100
pi@raspberrypi:~/usageExamples $ openssl pkeyutl -engine tpm2tss -keyform engine
 -inkey key.pair -decrypt -in encrypted.data  -out decrypted.data
engine "tpm2tss" set.
Enter password for user key:
pi@raspberrypi:~/usageExamples $ hexdump -C decrypted.data
00000000  73 65 63 72 65 74 20 74  65 78 74 0a              |secret text.|
0000000c
```

Figure 12 Encrypt and decrypt data with the TPM OpenSSL Engine

Code Listing 54 Template for command encrypt/decrypt with OpenSSL engine

```
001         cd /home/pi/usageExamples
002         echo "secret text" > plain.data
003         tpm2tss-genkey -a rsa -s 2048 -p abc key.pair
004         openssl rsa -engine tpm2tss -inform engine -in key.pair -pubout \
   -outform pem -out key.pub
005         openssl pkeyutl -pubin -inkey key.pub -in plain.data -encrypt \
   -out encrypted.data
006         hexdump -C encrypted.data
007         openssl pkeyutl -engine tpm2tss -keyform engine -inkey key.pair \
   -decrypt -in encrypted.data  -out decrypted.data
008         hexdump -C decrypted.data
```

5.2.3 ECDSA sign and verify

In this example, the OpenSSL TPM-Engine is used to sign and verify data.

Therefore, in a first step, the to-be-signed data file is created (“plain.data”). Afterwards, a 256-bit ECC key pair is created and the public key is exported. The secret key is secured with a passphrase (“abc”) and is used to sign the data file. The resulting signature is verified again with the OpenSSL TPM engine and a successful verification message is displayed.

While terminal in- and outputs are depicted in Figure 13, the template shown in Code Listing 55 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo -n "my_authentic_message" > plain.data
pi@raspberrypi:~/usageExamples $ tpm2tss-genkey -a ecdsa -c nist_p256 -p abc key
.pair
pi@raspberrypi:~/usageExamples $ openssl pkeyutl -engine tpm2tss -keyform engine
 -inkey key.pair -sign -in plain.data -out sign.data
engine "tpm2tss" set.
Enter password for user key:
pi@raspberrypi:~/usageExamples $ openssl pkeyutl -engine tpm2tss -keyform engine
 -inkey key.pair -verify -in plain.data -sigfile sign.data
engine "tpm2tss" set.
Enter password for user key:
Signature Verified Successfully
```

Figure 13 Sign data and verify signature with the TPM OpenSSL Engine

Code Listing 55 Template for command sign/verify with OpenSSL engine

```
001         cd /home/pi/usageExamples
002         echo -n "my_authentic_message" > plain.data
003         tpm2tss-genkey -a ecdsa -c nist_p256 -p abc key.pair
004         openssl pkeyutl -engine tpm2tss -keyform engine -inkey key.pair \
   -sign -in plain.data -out sign.data
005         openssl pkeyutl -engine tpm2tss -keyform engine -inkey key.pair \
   -verify -in plain.data -sigfile sign.data
```

5.3 Cryptsetup

Cryptsetup is a utility used to conveniently set up disk encryption based on the DMCrypt kernel module. The installed version adds a new “tpm2” keyslot type in DMCrypt, which is used for storing disc encryption keys inside the TPM. The keys are stored in the NV storage of the TPM and the read access of the keys are protected by a password and optionally by a PCR restriction. If PCR restrictions are used, the key is only accessible if certain PCR values are present inside the TPM.

In the following examples the log level is not propagated when sudo is used, so that detailed debug messages might be displayed. In order to avoid these debug messages, the environment variable for the log level is set in conjunction with the sudo command. There may appear also a warning message regarding a missing directory /run/cryptsetup. This is a non-critical warning and should disappear after a reboot.

Note: *In the following examples, dd is used to create an image file that will be used as volume with DMCrypt to perform the encryption (`luks2tpm.img`). Instead of using `dd`, also a system partition may be used . To create such a system partition various tools can be used[1]. In this case, the image file `luks2tpm.img` needs to be replaced with the path and name of the resulting partition, e.g., `/dev/mmcblk0p3`, in the respective examples.*

Note: *The NVRAM is a non-volatile memory area inside the TPM to store arbitrary data. Should the NVRAM erroneously remain occupied (e.g., `Error Code 0X14c`), the TPM’s NVRAM can be cleared with the program referred to in Section 6.1.3. Alternatively, the TPM2 Tools can be used to check for occupied indices (`tpm2_getcap handles-nv-index`) and to remove them (`tpm2_nvundefine`)*

Attention: ***Wrong handling of dd, cryptsetup or the partition tools can render the whole system unusable.***

5.3.1 NVRAM key slot

In this example, the TPM is used to store the volume key directly within the TPM's NVRAM memory (s. Figure 14).The access to the NVRAM area of the volume key is protected with a password.

First, the image file is created with `dd` and then `luksFormat` is used to initialize a LUKS partition, in this case the image “`luks2tpm.img`”. The TPM creates internally the initial encryption key, which is secured by the passphrase from the user input. Afterwards, the LUKS partition is opened with `luksOpen` by typing the previously defined passphrase to authorize key usage. The command `mkfs.ext4` is used to create an EXT4 file system when the partition has been opened (i.e., decrypted) successfully. Next, the partition is mounted to a previously created folder (`tpmFS`) and a file (“secretFile”) is created within the partition. Afterwards, it is unmounted (`umount`) and `luksClose` is used to close the image file so that it remains encrypted again. Finally, the key is deleted with `luksRemoveKey`.

While terminal in- and outputs are depicted in Figure 14, the template shown in Code Listing 56 enables easy copying of the respective commands.

[1] https://wiki.archlinux.org/index.php/partitioning#Partitioning_tools

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ dd if=/dev/zero of=luks2tpm.img bs=1k count=100
00
10000+0 records in
10000+0 records out
10240000 bytes (10 MB, 9.8 MiB) copied, 0.18618 s, 55.0 MB/s
pi@raspberrypi:~/usageExamples $ cryptsetup luksFormat --type=luks2 --tpm --tpm-
bank=sha1,sha256 luks2tpm.img

WARNING!
========
This will overwrite data on luks2tpm.img irrevocably.

Are you sure? (Type uppercase yes): YES
Enter passphrase for luks2tpm.img:
Verify passphrase:
pi@raspberrypi:~/usageExamples $ sudo TSS2_LOG=all+NONE cryptsetup luksOpen luks
2tpm.img luks2tpmFS
Enter passphrase for luks2tpm.img:
pi@raspberrypi:~/usageExamples $ sudo cryptsetup status /dev/mapper/luks2tpmFS
/dev/mapper/luks2tpmFS is active.
  type:    LUKS2
  cipher:  aes-xts-plain64
  keysize: 256 bits
  key location: keyring
  device:  /dev/loop0
  loop:    /home/pi/usageExamples/luks2tpm.img
  sector size:  512
  offset:  8192 sectors
  size:    11808 sectors
  mode:    read/write
pi@raspberrypi:~/usageExamples $ sudo mkfs.ext4 /dev/mapper/luks2tpmFS
mke2fs 1.44.5 (15-Dec-2018)
Creating filesystem with 5904 1k blocks and 1480 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

pi@raspberrypi:~/usageExamples $ mkdir tpmFS
pi@raspberrypi:~/usageExamples $ sudo mount /dev/mapper/luks2tpmFS tpmFS
pi@raspberrypi:~/usageExamples $ sudo touch tpmFS/secretFile
pi@raspberrypi:~/usageExamples $ sudo umount /home/pi/usageExamples/tpmFS
pi@raspberrypi:~/usageExamples $ sudo cryptsetup luksClose luks2tpmFS
pi@raspberrypi:~/usageExamples $ cryptsetup luksRemoveKey luks2tpm.img
Enter passphrase to be deleted:

WARNING!
========
This is the last keyslot. Device will become unusable after purging this key.

Are you sure? (Type uppercase yes): YES
```

Figure 14 Store volume key in the TPM's NVRAM

Code Listing 56 Template for storing passphrase in TPM's NVRAM with Cryptsetup

```
001         cd /home/pi/usageExamples
002         dd if=/dev/zero of=luks2tpm.img bs=1k count=10000
003         cryptsetup luksFormat --type=luks2 --tpm --tpm-bank=sha1,sha256 \
   luks2tpm.img
004         sudo TSS2_LOG=all+NONE cryptsetup luksOpen luks2tpm.img luks2tpmFS
005         sudo cryptsetup status /dev/mapper/luks2tpmFS
006         sudo mkfs.ext4 /dev/mapper/luks2tpmFS
007         mkdir tpmFS
008         sudo mount /dev/mapper/luks2tpmFS tpmFS
009         sudo touch tpmFS/secretFile
010         sudo umount /home/pi/usageExamples/tpmFS
011         sudo cryptsetup luksClose luks2tpmFS
012         cryptsetup luksRemoveKey luks2tpm.img
```

5.3.2 Additional key slot

In this example, a second key slot is added in the NVRAM of the TPM to the encrypted partition so that the partition can additionally be decrypted with a second key and passphrase. After the image has been created with `dd` and formatted with `luksFormat`, `luksAddKey` is used to add a new volume key secured with another passphrase in the specified NV index of the TPM.

Therefore, first the old passphrase needs to be entered for authentication, which allows the access to the volume encryption key. The volume encryption key is then re-encrypted using a new TPM-secured "Key Encryption Key".

Afterwards the new passphrase for the second key needs to be provided to create the additional key slot in the TPM. The new passphrase can then additionally be used with `luksOpen` to decrypt the partition. The proceedings explained in the example before can be used to store data on the partition. Finally, the second key is deleted with `luksRemoveKey` and the first key with `tpm2_nvundefine`. The command `tpm2_getcap handles-nv-index` is used to observe the reserved the NV indices of the TPM.

While terminal in- and outputs are depicted in Figure 15, the template shown in Code Listing 57 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ dd if=/dev/zero of=luks2tpm.img bs=1k count=100
00
10000+0 records in
10000+0 records out
10240000 bytes (10 MB, 9.8 MiB) copied, 0.160655 s, 63.7 MB/s
pi@raspberrypi:~/usageExamples $ cryptsetup luksFormat --type=luks2 --tpm --tpm-
bank=sha1,sha256 luks2tpm.img

WARNING!
========
This will overwrite data on luks2tpm.img irrevocably.

Are you sure? (Type uppercase yes): YES
Enter passphrase for luks2tpm.img:
Verify passphrase:
pi@raspberrypi:~/usageExamples $ cryptsetup luksAddKey --tpm --tpm-nv=0x01BFFFFD
 --tpm-bank=sha1,sha256 luks2tpm.img
Enter any existing passphrase:
Enter new passphrase for key slot:
Verify passphrase:
pi@raspberrypi:~/usageExamples $ sudo TSS2_LOG=all+NONE cryptsetup luksOpen luks
2tpm.img luks2tpmFS
Enter passphrase for luks2tpm.img:
pi@raspberrypi:~/usageExamples $ sudo cryptsetup luksClose luks2tpmFS
pi@raspberrypi:~/usageExamples $ tpm2_getcap handles-nv-index
- 0x1BF0000
- 0x1BFFFFD
- 0x1C00002
- 0x1C0000A
pi@raspberrypi:~/usageExamples $ cryptsetup luksRemoveKey luks2tpm.img
Enter passphrase to be deleted:
pi@raspberrypi:~/usageExamples $ tpm2_getcap handles-nv-index
- 0x1BF0000
- 0x1C00002
- 0x1C0000A
pi@raspberrypi:~/usageExamples $ tpm2_nvundefine 0x1BF0000
pi@raspberrypi:~/usageExamples $ tpm2_getcap handles-nv-index
- 0x1C00002
- 0x1C0000A
```

Figure 15 Add a new key slot for the encrypted partition

Code Listing 57 Template for adding new key slot for encrypted partition with Cryptsetup

```
001         cd /home/pi/usageExamples
002         dd if=/dev/zero of=luks2tpm.img bs=1k count=10000
003         cryptsetup luksFormat --type=luks2 --tpm --tpm-bank=sha1,sha256 \
   luks2tpm.img
004         cryptsetup luksAddKey --tpm --tpm-nv=0x01BFFFFD \
   --tpm-bank=sha1,sha256 luks2tpm.img
005         sudo TSS2_LOG=all+NONE cryptsetup luksOpen luks2tpm.img luks2tpmFS
006         sudo cryptsetup luksClose luks2tpmFS
007         tpm2_getcap handles-nv-index
008         cryptsetup luksRemoveKey luks2tpm.img
009         tpm2_getcap handles-nv-index
010         tpm2_nvundefine 0x1BF0000
011         tpm2_getcap handles-nv-index
```

5.4 TPM2 PKCS11

In the following examples, the tpm2-pkcs11 library is used as a module for the pkcs11-tool, which is part of OpenSC software. This is done by specifying the module parameter as shown in Code Listing 58. The tpm2-pkcs11 library uses the FAPI to execute the respective operations with the TPM.

Code Listing 58 Creating an alias for pkcs11-tool with tpm2-pkcs11 as module

```
001         export mod_path=/home/pi/tssSetup/\
   tpm2-pkcs11/src/.libs/libtpm2_pkcs11.so
002         export TPM2_PKCS11_STORE="$HOME/usageExamples/store"
003         alias pkcs11_tool='pkcs11-tool --module $mod_path'
004         mkdir $TPM2_PKCS11_STORE
```

Note: To avoid distracting message output, the log level should be set as described in Section 4.8.

5.4.1 Creating random data using PKCS11

Using the tpm2_pkcs11 library as a module for the pkcs11-tool, the TPM's hardware number generator can be used to retrieve random values.

While terminal in- and outputs are depicted in Figure 16, the template shown in Code Listing 59 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ pkcs11_tool --slot=1 --generate-random 8 | xxd
00000000: c426 3294 c449 bde6                      .&2..I..
```

Figure 16 Using the TPM's hardware-based random number generator to generate an 8-Byte random number

Code Listing 59 Template for the command generate-random of the pkcs11-tool

```
001         cd /home/pi/usageExamples
002         pkcs11_tool --slot=1 --generate-random 8 | xxd
```

5.4.2 Token initialization and object creation

In this example, the `pkcs11_tool` is used to initialize a key store and create two slots. For each slot a token is created and an object stored under it (Figure 17)[1]. This configuration is used also for the other examples.

In a first step, the token is initialized in slot 1 with a label and an admin PIN. Then a user PIN is created for this token and a key is created with the user PIN as authorization. This procedure is repeated for the second slot. Afterwards, the created objects are listed.

While terminal in- and outputs are depicted in Figure 17, the template shown in Code Listing 60 enables easy copying of the respective commands.

[1] The error (ERROR: Mechanism not supported) in Figure 17 during creation of the ECC key pair can savely be ignored and will be fixed in a future version (see https://github.com/tpm2-software/tpm2-pkcs11/issues/555)

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ mkdir -p $TPM2_PKCS11_STORE
pi@raspberrypi:~/usageExamples $ pkcs11_tool --init-token --slot=1 --label=myfir
sttoken \
> --so-pin=myfirstsopin
Token successfully initialized
pi@raspberrypi:~/usageExamples $ pkcs11_tool --init-pin --slot=1 --login --login
-type="so" \
> --so-pin=myfirstsopin --pin=myfirstuserpin
User PIN successfully initialized
pi@raspberrypi:~/usageExamples $ pkcs11_tool --keypairgen --slot=1 --label="mysi
gnkey" --login \
> --pin=myfirstuserpin --usage-sign --key-type=EC:prime256v1
ERROR: Mechanism not supported, got: 0x66656666
Key pair generated:
Private Key Object; EC
  label:      mysignkey
  Usage:      decrypt, sign
Public Key Object; EC  EC_POINT 256 bits
  EC_POINT:   044104626a79437c34c180a61180b62b14bf098ba66a13509a63c2c068d58a8ad2
f00abd9b6e611a3537d1eb9f16197463fdae24886393fb87a194c792241342f248c3
  EC_PARAMS:  06082a8648ce3d030107
  label:      mysignkey
  Usage:      encrypt, verify
pi@raspberrypi:~/usageExamples $ pkcs11_tool --init-token --slot=2 --label=mysec
ondtoken \
> --so-pin=mysecondsopin
Token successfully initialized
pi@raspberrypi:~/usageExamples $ pkcs11_tool --init-pin --slot=2 --login --login
-type="so" \
> --so-pin=mysecondsopin --pin=myseconduserpin
User PIN successfully initialized
pi@raspberrypi:~/usageExamples $ pkcs11_tool --keypairgen --slot=2 --label="myde
cryptkey" --login \
> --pin=myseconduserpin
Key pair generated:
Private Key Object; RSA
  label:      mydecryptkey
  Usage:      decrypt, sign
Public Key Object; RSA 2048 bits
  label:      mydecryptkey
  Usage:      encrypt, verify
pi@raspberrypi:~/usageExamples $ pkcs11_tool --list-objects --slot=1
Public Key Object; EC  EC_POINT 256 bits
  EC_POINT:   044104626a79437c34c180a61180b62b14bf098ba66a13509a63c2c068d58a8ad2
f00abd9b6e611a3537d1eb9f16197463fdae24886393fb87a194c792241342f248c3
  EC_PARAMS:  06082a8648ce3d030107
  label:      mysignkey
  Usage:      encrypt, verify
pi@raspberrypi:~/usageExamples $ pkcs11_tool --list-objects --slot=2
Public Key Object; RSA 2048 bits
  label:      mydecryptkey
  Usage:      encrypt, verify
pi@raspberrypi:~/usageExamples $ pkcs11_tool --list-slots
Available slots:
Slot 0 (0x1): myfirsttoken                    Infineon
  token label        : myfirsttoken
  token manufacturer : Infineon
  token model        : SLB9670
  token flags        : login required, rng, token initialized, PIN initialized
  hardware version   : 1.38
  firmware version   : 7.85
  serial num         : 0000000000000000
  pin min/max        : 0/128
Slot 1 (0x2): mysecondtoken                   Infineon
  token label        : mysecondtoken
  token manufacturer : Infineon
  token model        : SLB9670
  token flags        : login required, rng, token initialized, PIN initialized
  hardware version   : 1.38
  firmware version   : 7.85
  serial num         : 0000000000000000
  pin min/max        : 0/128
Slot 2 (0x3):                                 Infineon
  token state:   uninitialized
```

Figure 17 Initialize a store, create a slot and token and create an object under the token

Code Listing 60 Template for token initialization and object creation

```
001         cd /home/pi/usageExamples
002         mkdir -p $TPM2_PKCS11_STORE
003         pkcs11_tool --init-token --slot=1 --label=myfirsttoken \
   --so-pin=myfirstsopin
004         pkcs11_tool --init-pin --slot=1 --login --login-type="so" \
   --so-pin=myfirstsopin --pin=myfirstuserpin
005         pkcs11_tool --keypairgen --slot=1 --label="mysignkey" --login \
   --pin=myfirstuserpin --usage-sign --key-type=EC:prime256v1
006         pkcs11_tool --init-token --slot=2 --label=mysecondtoken \
   --so-pin=mysecondsopin
007         pkcs11_tool --init-pin --slot=2 --login --login-type="so" \
   --so-pin=mysecondsopin --pin=myseconduserpin
008         pkcs11_tool --keypairgen --slot=2 --label="mydecryptkey" --login \
   --pin=myseconduserpin
009         pkcs11_tool --list-objects --slot=1
010         pkcs11_tool --list-objects --slot=2
011         pkcs11_tool --list-slots
```

5.4.3 Sign and verify using PKCS11

In this example, the `pkcs11_tool` is used with the TPM to sign some data that is then verified using OpenSSL. For this example the key from example 5.4.2 is used.

In a first step, some data file is created and is used as input for the sign operation inside the TPM. Then, the public key is read out. Finally, OpenSSL is used to encode the public key into the PEM format and verify the signature.

While terminal in- and outputs are depicted in Figure 18, the template shown in Code Listing 61 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo -n "01234567890123456789" > sign_data.file
pi@raspberrypi:~/usageExamples $ pkcs11_tool --sign --slot=1 --label="mysignkey"
 \
> --pin=myfirstuserpin --mechanism=ECDSA-SHA1 --signature-format=sequence \
> --input-file=sign_data.file > sign_data.sig
Using signature algorithm ECDSA-SHA1
pi@raspberrypi:~/usageExamples $ pkcs11_tool --read-object --slot=1 --label="mys
ignkey" \
> --pin=myfirstuserpin --type pubkey > pubkey.der
pi@raspberrypi:~/usageExamples $ openssl ec -inform DER -outform PEM -in pubkey.
der -pubin > \
> pubkey.pem
read EC key
writing EC key
pi@raspberrypi:~/usageExamples $ openssl dgst -sha1 -verify pubkey.pem -signatur
e sign_data.sig \
> sign_data.file
Verified OK
```

Figure 18 The tpm2-pkcs11 tool is used to sign some data that is then verified with OpenSSL

Code Listing 61 Template for sign and verify with the pkcs11-tool

```
001         cd /home/pi/usageExamples
002         echo -n "01234567890123456789" > sign_data.file
003         pkcs11_tool --sign --slot=1 --label="mysignkey" \
   --pin=myfirstuserpin --mechanism=ECDSA-SHA1 --signature-format=sequence \
   --input-file=sign_data.file > sign_data.sig
004         pkcs11_tool --read-object --slot=1 --label="mysignkey" \
   --pin=myfirstuserpin --type pubkey > pubkey.der
005         openssl ec -inform DER -outform PEM -in pubkey.der -pubin > \
   pubkey.pem
006         openssl dgst -sha1 -verify pubkey.pem -signature sign_data.sig \
   sign_data.file
```

5.4.4 Encrypt and decrypt using PKCS11

Note: At the time of creation of this document, TPM2 PKCS11 only supports the use of RSA keys for encryption. The support of encryption with ECC might be added in a future release. It is then recommended to switch to ECC for the encryption of data.

In this example, OpenSSL is used to encrypt some data with the public key, that is then decrypted using the `pkcs11_tool` with the private key inside the TPM. For this example the key from example 5.4.2 is used.

In a first step, the data to encrypt is created and the `pkcs11_tool` is used to read out the public key. Again, OpenSSL is used to encode the public key into the PEM format and encrypt the data. The encrypted data is then decrypted by the TPM using FAPI and the `pkcs11_tool`.

While terminal in- and outputs are depicted in Figure 19, the template shown in Code Listing 62 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo "secret data" > encrypt_data.file
pi@raspberrypi:~/usageExamples $ pkcs11_tool --read-object --slot=2 --label="myd
ecryptkey" \
> --pin=myseconduserpin --type pubkey > pubkey.der
pi@raspberrypi:~/usageExamples $ openssl rsa -inform DER -outform PEM -in pubkey
.der \
> -pubin > pubkey.pem
writing RSA key
pi@raspberrypi:~/usageExamples $ openssl rsautl -encrypt -inkey pubkey.pem -in e
ncrypt_data.file \
> -pubin -out encrypted.File
pi@raspberrypi:~/usageExamples $ cat encrypted.File | pkcs11_tool --decrypt --sl
ot=2 \
> --label=mysecondtoken --pin=myseconduserpin --mechanism=RSA-PKCS \
> > decrypted_data.file
Using decrypt algorithm RSA-PKCS
pi@raspberrypi:~/usageExamples $ xxd -s -12 decrypted_data.file
000000f4: 7365 6372 6574 2064 6174 610a            secret data.
```

Figure 19 The tpm2-pkcs11 tool is used to decrypt some data that is encrypted with OpenSSL

Code Listing 62 Template for encrypt and decrypt with the pkcs11-tool

```
001         cd /home/pi/usageExamples
002         echo "secret data" > encrypt_data.file
003         pkcs11_tool --read-object --slot=2 --label="mydecryptkey" \
   --pin=myseconduserpin --type pubkey > pubkey.der
004         openssl rsa -inform DER -outform PEM -in pubkey.der \
   -pubin > pubkey.pem
005         openssl rsautl -encrypt -inkey pubkey.pem -in encrypt_data.file \
   -pubin -out encrypted.File
006         cat encrypted.File | pkcs11_tool --decrypt --slot=2 \
   --label=mysecondtoken --pin=myseconduserpin --mechanism=RSA-PKCS \
   > decrypted_data.file
007         xxd -s -12 decrypted_data.file
```

5.4.5 Change PIN and delete keys using PKCS11

In this example, the user PIN for a key slot is changed and then the key is deleted. For this example the key from example 5.4.2 is used.

In a first step, the objects stored under slot 2 are retrieved. Then, the user PIN for key slot 2 is changed. A successful PIN change is verified by decrypting the encrypted data file from the previous example again. Key usage is authorized with the new user PIN. Finally, the key in slot 2 is deleted. A successful deletion is verified by retrieving the now empty list of objects stored under key slot 2.

While terminal in- and outputs are depicted in Figure 20, the template shown in Code Listing 63 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ pkcs11_tool --list-objects --slot=2
Public Key Object; RSA 2048 bits
  label:      mydecryptkey
  Usage:      encrypt, verify
pi@raspberrypi:~/usageExamples $ pkcs11_tool --change-pin --slot=2 --pin=mysecon
duserpin \
> --new-pin=mynewuserpin
PIN successfully changed
pi@raspberrypi:~/usageExamples $  cat encrypted.File | pkcs11_tool --decrypt --s
lot=2 \
> --label=mysecondtoken --pin=mynewuserpin --mechanism=RSA-PKCS \
> > decrypted_data.file_new
Using decrypt algorithm RSA-PKCS
pi@raspberrypi:~/usageExamples $ xxd -s -12 decrypted_data.file_new
000000f4: 7365 6372 6574 2064 6174 610a            secret data.
pi@raspberrypi:~/usageExamples $ pkcs11_tool --delete-object --slot=2 --label="m
ydecryptkey" \
> --pin=mynewuserpin --login --type=pubkey
pi@raspberrypi:~/usageExamples $ pkcs11_tool --list-objects --slot=2
```

Figure 20 The tpm2-pkcs11 tool is used to change the user PIN and delete an object

Code Listing 63 Template for changing the PIN and deleting keys using the pkcs11-tool

```
001         cd /home/pi/usageExamples
002         pkcs11_tool --list-objects --slot=2
003         pkcs11_tool --change-pin --slot=2 --pin=myseconduserpin \
   --new-pin=mynewuserpin
004         cat encrypted.File | pkcs11_tool --decrypt --slot=2 \
   --label=mysecondtoken --pin=mynewuserpin --mechanism=RSA-PKCS \
   > decrypted_data.file_new
005         xxd -s -12 decrypted_data.file_new
006         pkcs11_tool --delete-object --slot=2 --label="mydecryptkey" \
   --pin=mynewuserpin --login --type=pubkey
007         pkcs11_tool --list-objects --slot=2
```

6 TSS Programming

This section shows how to create and compile programs using the installed TSS FAPI library. The FAPI provides a generic method to implement use cases with a low programming overhead and is recommended for typical scenarios.

If a specific use case with special TPM functionalities needs to be implemented that is not available in FAPI, it is recommended to use the TSS ESAPI for this use case. The usage of the ESAPI requires more development effort, however it has the advantage that the entire TPM commandset can be used. The usage and programming with the ESAPI is described in Appendix 0.

Note: To avoid distracting message output, the log level should be set as described in Section 4.8.

6.1 Basics

This section shows how to generate and use the TSS documentation, how to compile code that uses the TSS and how to clear the TPM's NVRAM in case it remains erroneously occupied.

6.1.1 TSS documentation

The TSS library includes an HTML-based documentation that is generated with Doxygen. The documentation supports developers using the TSS library, e.g., by explaining the commands and parameters of the Feature API (FAPI) or Enhanced System API (ESAPI) and TSS data structures. By following this Application Note, the documentation is automatically generated since the doxygen package was installed as dependency before installing the TSS.

However, the documentation can also be installed afterwards as shown in Code Listing 64.

Code Listing 64 Manually installing the TSS doxygen documentation

```
001 sudo apt install doxygen
002 cd /home/pi/tssSetup/tpm2-tss/
003 ./configure --enable-doxygen-doc
004 make doxygen-doc
```

Within the root directory of the TSS software (`/home/pi/tssSetup/tpm2-tss/`), the documentation will be installed into `/doxygen-doc/html/` and can be viewed with various web browsers by opening the `index.html`. This will open the main page of the documentation (s. Figure 21). The navigation bar on the left and top can be used to navigate through the documentation.

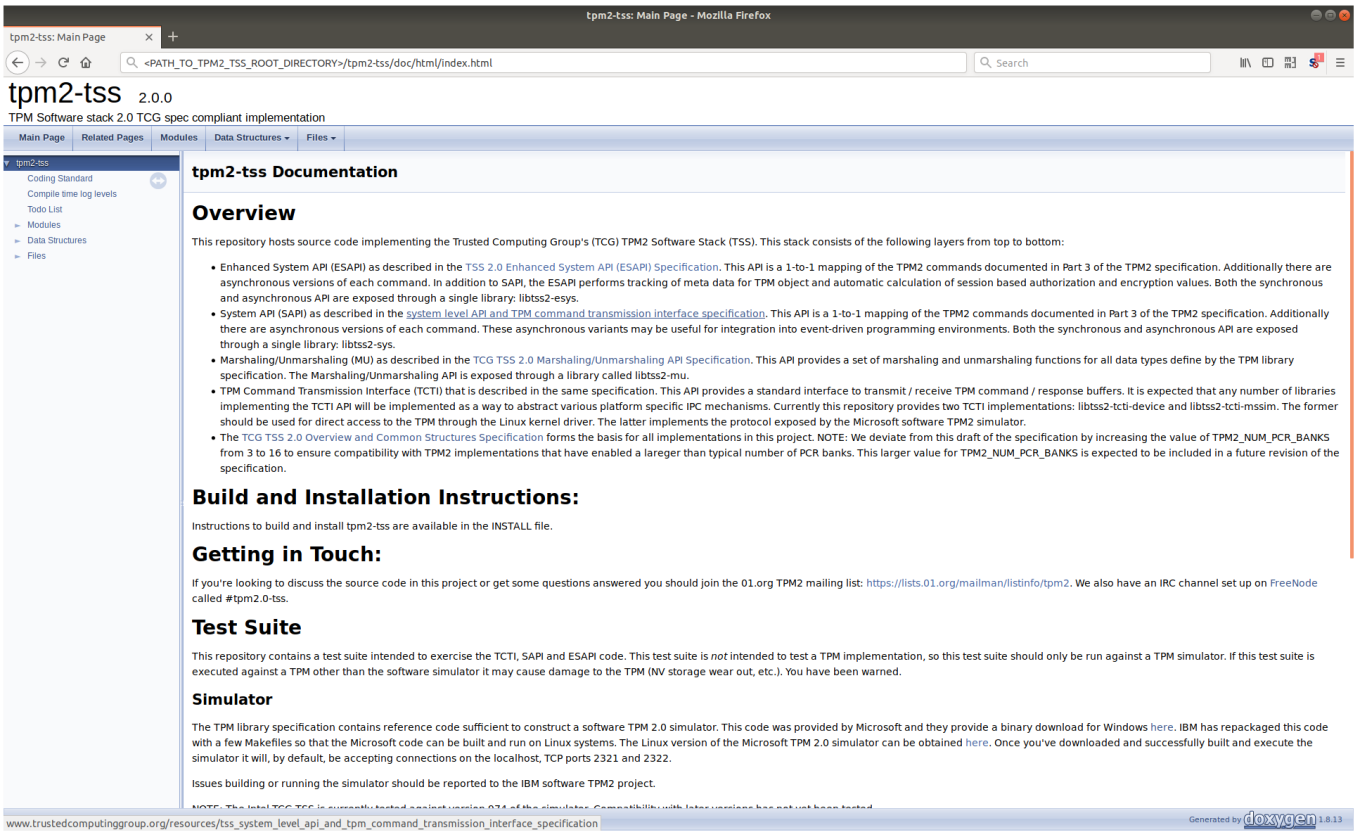


Figure 21 Main page of the documentation

6.1.2 Compiling code that uses the TSS FAPI library

This document has several source code files attaced, which provide basic examples of the programming with the FAPI to implement generic use cases. These programming examples are compiled and explained in the following sections.

The code examples of Sections 6.2.1 (Generation of various keys), 6.2.4 (Write to and read from NV-Index) and 6.2.6 (Key Duplication) can be compiled with the command in Code Listing 65. The variable `OWN_PROGRAM` needs to be replaced with the name of the `.c file` that should be compiled.

Code Listing 65 Command to compile programs that use the TSS FAPI library (6.2.1, 6.2.4, 6.2.6)

```
001    gcc OWN_PROGRAM.c -ltss2-fapi -o OWN_PROGRAM
```

The programming examples of Sections 6.2.2 (Signature generation and validation) and 6.2.3 (Decryption of data using signature-based authorization) use the OpenSSL library. Therefore, this library needs to be linked additionally with `-lcrypto`. The command is shown in Code Listing 66.

Code Listing 66 Command to compile programs that use the OpenSSL library (6.2.2, 6.2.3)

```
001    gcc OWN_PROGRAM.c -ltss2-fapi -lcrypto -o OWN_PROGRAM
```

The programming example in Section 6.2.7 (Quote) has a dependency on the json library. Therefore, this library needs to be linked additionally with `-ljson-c`. The command is shown in Code Listing 67.

Code Listing 67 Command to compile programs that use the JSON library (6.2.7)

```
001    gcc OWN_PROGRAM.c -ltss2-fapi -ljson-c -o OWN_PROGRAM
```

6.1.3 Clearing the TPM NVRAM

All code and usage examples in this Application Note will do a clean up of the TPM's NVRAM, e.g., when using the NV index. However, when modifying or writing new code or scripts, the NVRAM might remain occupied due to an incomplete clean up, which might result in errors. In this case, the NVRAM can be cleaned up manually using the TPM2 Tools command `tpm2_clear`. However, since `tpm2_clear` uses ESAPI, it is unware of the FAPI meta data stores and they become out-of-sync with the TPM state. Therefore, it is necessary to clear the FAPI meta stores manually and redo the provisioning. Code Listing 68 shows an example how to do this. First the system (l.001) and user (l.002) stores are deleted and then the TPM2 Tools are used to redo provisioning (l.003).

Code Listing 68 Example to manually clear the Fapi meta stores in case they are out-of-sync with the TPM

```
001 rm -rf /var/lib/tpm2-tss/system/keystore/
002 rm -rf ~/.local/share/tpm2-tss/user/keystore/
003 tss2_provision
```

6.1.4 General information about the programming examples

For clarity, the following sections for the FAPI and the section 0 for the ESAPI show only the important code snippets for the corresponding programming examples. The full standalone working examples are delivered with this Application Note in separate `.c files`. The corresponding `.c file` is referenced in the section header of corresponding section.

General functions excluded within this Application Note but shown in the `.c files` are error handling, code cleanup, ESAPI/FAPI context initialization and hierarchy authentication. Moreover, all include headers are being left out. Especially, in the key generation examples, all keys are created in the hierarchy of a primary key. The primary key's creation will be shown in the first example only.

6.2 Programming Examples FAPI

The following shows programming examples using the FAPI.

To compile the following programming examples, FAPI needs to be initialized as described in chapter 4.2.4. In short: When using the standard FAPI profiles and configuration file, at least the FAPI user storage has to be created. If not already done, the command in Code Listing 9 can be used. Otherwise, in case of a custom FAPI configuration file and/or profile, all custom paths need to be existing and set correctly in the custom FAPI configuration file. If they do not exist yet, they need to be created manually by the user.

6.2.1 Generation of various keys

This example shows how to generate a Primary Key, a Signature Key, an Encryption Key and a Restricted Key with the TPM and using the FAPI. The Signature, Encryption and Restricted Keys are being created with the Primary Key as the parent. The explanation of the properties of these key types is explained in the TPM library specification [4].

6.2.1.1 Primary Key (createPrimaryKey_FAPI.c)

In this example, an ECC P256 primary key is created according to the profile set with the configuration file. This primary key will be used as Storage Root Key (SRK) in the following examples. Further details on the SRK and the provisioning is provided in the TCG TPM 2.0 Provisioning Guidance [7].

A primary key is a specific key, which is located at the top layer of the storage key hierarchy. It is used to encrypt other (ordinary) keys, which can be created later on without limitation and are then located below the primary key in the key hierarchy. These ordinary keys are automatically encrypted involving their parent primary key, so that the resulting key blob does not need to be protected inside the TPM anymore. Therefore, the encrypted key blob is safe to be stored outside of the TPM within untrusted memory.

1. Initialize FAPI (Code Listing 69)
2. Create the primary key according to the profile set with the configuration file (Code Listing 70)
3. Release context

As seen in Code Listing 69 the FAPI is initialized by passing only the FAPI context. The uri parameter is currently unused in the specification and thus set to NULL. The uri parameter is intended to allow the caller to specify how to connect to the TPM if non-default options are needed.

Code Listing 69 Initialize FAPI

```
001         /* Initialize the FAPI context. */
002         FAPI_CONTEXT *fapi_context;
003         r = Fapi_Initialize(&fapi_context, NULL);
004         if (r != TSS2_RC_SUCCESS) {
005             printf("Error: Fapi_Initialize\n");
006             goto error;
007         }
```

During FAPI initialization, the profile set with the configuration file is read. The parameters defined in the profile are used to create the primary key (Code Listing 70).

Code Listing 70 Create the primary key according to the profile set with the configuration file

```
001         r = Fapi_Provision(fapi_context, NULL, NULL, NULL);
002         if (r != TSS2_FAPI_RC_ALREADY_PROVISIONED && r != TSS2_RC_SUCCESS) {
003             printf("Error: Fapi_Provision\n");
```

Code Listing 70 Create the primary key according to the profile set with the configuration file

```
004             goto error;
005         }
```

6.2.1.2 Signature Key (createSignatureKey_FAPI.c)

In this example, an ECC P256 signature key is created in the storage hierarchy (HS) of the TPM below the SRK. Its cryptographic algorithms and parameters are specified in the FAPI profile (more information with the console command “man fapi-profile”). Generally, signature keys are used to sign various data or used within a TPM quote command to sign data used for attestation. The latter use case requires a "restricted" signing key and is shown within the "Quote" example.

To create a signature key the following steps have to be carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.
2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.
3. Create the signature key with the parameters "sign" (signing key) and "noDa" (no dictionary attack protection) (Code Listing 71)

As seen in Code Listing 71, the signature key is created with the FAPI command `Fapi_CreateKey`. With the first parameter the current FAPI context is passed. With the second parameter the key path and name is specified. The third parameter defines the type of key. In this case the parameters “sign” to create a signing key and “noDa” to disable dictionary attack protection are used. The next parameters would add a policy and a passphrase for key usage. In this example both parameters are left empty.

Code Listing 71 Create the signature key with the parameters "sign" (signing key) and "noDa" (no dictionary attack protection)

```
001         r = Fapi_CreateKey(fapi_context, "HS/SRK/mySignKey",
002             "noDa, sign", "", NULL);
003         if (r != TSS2_RC_SUCCESS) {
004             printf("Error: Fapi_CreateKey\n");
005             goto error;
006         }
```

6.2.1.3 Encryption Key (createEncryptionKey_FAPI.c)

Note: FAPI currently only supports to use RSA keys for encryption. Therefore, the profile needs to set to use RSA as described in section 4.2.4 and shown in Code Listing 12. The support of encryption with ECC is under discussion in the TSS software community, so that it might be added in a future release (e.g., version 3.1). It is then recommended to switch to ECC for the encryption of data. The creation and usage of an encryption key using symmetric cryptography is described in the ESAPI programming example in section 6.3.2.2.3.

In this example, an encryption key is created in the storage hierarchy (HS) below the SRK. Its cryptographic algorithms and parameters are specified in the FAPI profile (more information with the console command “man fapi-profile”). An encryption key of the TPM is used to encrypt and decrypt arbitrary data.

To create an encryption key the following steps are carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.

2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.
3. Create the encryption key with the parameters "decrypt" (encryption key) and noDa (no dictionary attack protection)

As seen in Code Listing 72, the signature key is created with the FAPI command `Fapi_CreateKey`. With the first parameter the current FAPI context is passed. With the second parameter the key path and name is specified. The third parameter defines the type of key. In this case the parameters “decrypt” to create an encryption key and “noDa” to disable dictionary attack protection are used. The next parameters would add a policy and a passphrase for key usage. In this example both parameters are left empty.

Code Listing 72 Create the encryption key with the parameters "decrypt" (encryption key) and noDa (no dictionary attack protection)

```
001         r = Fapi_CreateKey(fapi_context, "HS/SRK/myDecryptKey",
002             "decrypt, noDa", "", NULL);
003         if (r != TSS2_RC_SUCCESS) {
004             printf("Error: Fapi_CreateKey\n");
005             goto error;
006         }
```

6.2.1.4 Restricted Key (createRestrictedKey_FAPI.c)

In this example, a restricted (ECC P256 signature) key is created in the storage hierarchy (HS) below the SRK. Its cryptographic algorithms and parameters are specified in the FAPI profile (more information with the console command “man fapi-profile”). Since it is a restricted signature key, only data created by the TPM can be signed. An example for the usage of a restricted signature key is signing of the TPM internal attestation structures which, among other data, contain a selection of PCR values.

To create a restricted (signature) key the following steps are carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.
2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.3.1.1.
3. Create the restricted (signature) key with the parameters "restricted" (restricted key), "sign" (signature key) and "noDa" (no dictionary attack protection)

As seen in Code Listing 73, the signature key is created with the FAPI command `Fapi_CreateKey`. With the first parameter the current FAPI context is passed. With the second parameter the key path and name is specified. The third parameter defines the type of key. In this case the parameters “restricted” to create a restricted key, “sign” to create a signature key and “noDa” to disable dictionary attack protection are used. The next parameters would add a policy and a passphrase for key usage. In this example both parameters are left empty.

Code Listing 73 Create the restricted (signature) key with the parameters "restricted" (restricted key), "sign" (signature key) and noDa" (no dictionary attack protection)

```
001         r = Fapi_CreateKey(fapi_context, "HS/SRK/RestrictedSignKey",
002             "restricted, sign, noDa", "", NULL);
003         if (r != TSS2_RC_SUCCESS) {
004             printf("Error: Fapi_CreateKey\n");
005             goto error;
006         }
```

6.2.2 Signature generation and validation (createSignatureVerifyWithOpenSSL_FAPI.c)

In this example, an ECC P256 signature key is created in the storage hierarchy (HS) below the SRK. Its cryptographic algorithms and parameters are specified in the FAPI profile (more information with the console command “man fapi-profile”). It is used to sign some data digest. The signature is then verified with both TPM and OpenSSL. The signature key is created inside the TPM chip based on the validated TRNG and the private key never leaves the TPM. Also the security critical signing operation is done inside the TPM which protects it from software and hardware attacks.

A use case would be to authenticate the TPM to the host. In a simple challenge-response protocol the TPM signs a challenge from the host. The authentication is successful, if the host can verify the signature using the respective public key.

To create a signature key the following steps are carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.
2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.
3. Create the signature key with the parameters "sign" (signature key) and "noDa" (no dictionary attack protection) (not shown in Listings). This can be achieved as described in 6.2.1.2.
4. Use TPM to sign a defined digest and verify the signature (Code Listing 74)
5. Use OpenSSL to verify the signature created with the TPM (Code Listing 75)

After the signature key is created, the TPM is used to create and verify a signature (Code Listing 74). First a digest is defined (l.30ff) and a callback to authorize key usage is set (l.36). Then the key is used to sign the digest (l.43ff). In order to verify the signature the TPM key needs to be imported into the public key directory (l.52ff). Afterwards, the signature can be verified using the TPM (l.59ff).

Code Listing 74 Use TPM to sign a defined digest and verify the signature

```
001         /* Initialize variables used to signature generation and
023          * verification */
024         size_t signatureSize = 0;
025         uint8_t *signature;
026         char *publicKey;
027
028         /* Define digest which will be signed */
029         size_t digestSize = 20;
030         uint8_t digestStruct[] = { 0x67, 0x68, 0x03, 0x3e, 0x21, 0x64,
031             0x68, 0x24, 0x7b, 0xd0, 0x31, 0xa0, 0xa2, 0xd9, 0x87, 0x6d,
032             0x79, 0x81, 0x8f, 0x8f };
033         uint8_t *digest = &digestStruct[0];
034
035         /* Simulate callback to get key passphrase */
036         r = Fapi_SetAuthCB(fapi_context, auth_callback, "");
037         if (r != TSS2_RC_SUCCESS) {
038             printf("Error: Fapi_SetAuthCB\n");
039             goto error;
040         }
041
042         /* Sign the digest with TPM */
043         r = Fapi_Sign(fapi_context, "HS/SRK/RSASignKey", NULL,
044                       digest, digestSize, &signature,
045                       &signatureSize, &publicKey, NULL);
046         if (r != TSS2_RC_SUCCESS) {
```

Code Listing 74 Use TPM to sign a defined digest and verify the signature

```
047             printf("Error: Fapi_Sign\n");
048             goto error;
049         }
050
051         /* Import the public key to the public key directory */
052         r = Fapi_Import(fapi_context, "myExtPubKey", publicKey);
053         if (r != TSS2_RC_SUCCESS) {
054             printf("Error Fapi_Import\n");
055             goto error;
056         }
057
058         /* Verify the signature with TPM */
059         r = Fapi_VerifySignature(fapi_context, "ext/myExtPubKey",
060             digest, digestSize, signature, signatureSize);
061         if (r != TSS2_RC_SUCCESS) {
062             printf("Error: Fapi_VerifySignature\n");
063             goto error;
064         }
```

Alternatively, the created signature can be verified with OpenSSL as shown in Code Listing 75. Therefore, the OpenSSL bio data structure is initialized with the PEM key (l.005) and used to create an OpenSSL EVP Key (l.021). After the verification is initialized (l.027), the correct hash algorithm (l.033) is set and the EVP key is used to verify the signature (l.039f).

Code Listing 75 Use OpenSSL to verify the signature created with the TPM

```
001         /* Initialize OpenSSL bio data structure with PEM key array.
002          * When -1 is selected as length, OpenSSL will use strlen to
003          * determine length of input */
004         BIO *bio = NULL;
005         bio = BIO_new_mem_buf(publicKey, -1);
006         if(!bio){
007             printf("Error: BIO_new_mem_buf\n");
008             goto error;
009         }
010
011         /* Create pkey from bio structure */
012         EVP_PKEY* pkey = NULL;
013         pkey = PEM_read_bio_PUBKEY(bio, NULL, NULL, NULL);
014         if(!pkey){
015             printf("Error: PEM_read_bio_PUBKEY\n");
016             goto error;
017         }
018
019         /* Initialize OpenSSL evp pkey context with verification key */
020         EVP_PKEY_CTX *pctx;
021         pctx = EVP_PKEY_CTX_new(pkey, NULL /* no engine */);
022         if(!pctx){
023           printf("EVP_PKEY_CTX_new failed: %d\n", r);
024         }
025
026         /* Initialize verification */
027         r = EVP_PKEY_verify_init(pctx);
028         if(r<=0){
029           printf("EVP_PKEY_verify_init failed: %d\n", r);
030         }
031
032         /* Set correct hash algorithm */
```

Code Listing 75 Use OpenSSL to verify the signature created with the TPM

```
033         r = EVP_PKEY_CTX_set_signature_md(pctx, EVP_sha256());
034         if(r<=0){
035           printf("EVP_PKEY_CTX_set_signature_md failed: %d\n", r);
036         }
037
038         /* Execute signature verification */
039         r = EVP_PKEY_verify(pctx, signature, signatureSize, digest,
040             digestSize);
041         if(r<=0){
042           printf("EVP_PKEY_verify failed: %d\n", r);
043         }
044         else{
045           printf("Signature successfully verified with OpenSSL!\n");
046         }
```

6.2.3 Decryption of data using signature-based authorization (signatureAuthorization_FAPI.c)

Note: FAPI currently only supports the use of RSA keys for encryption. Therefore, the profile needs to set to use RSA as described in section 4.2.4 and shown in Code Listing 12. The support of encryption with ECC is under discussion in the TSS software community, so that it might be added in a future release (e.g. version 3.1). It is then recommended to switch to ECC for the encryption of data.

This example shows how to use the TPM to decrypt data using an asymmetric TPM-generated key. This key is protected by a policy instead of a password that has to be fulfilled for every decryption operation. The policy provides the protection that only an allowed party can authorize the usage of the key for the decryption operation.

The TPM uses the internal TRNG to create a strong asymmetric key for the en-/decryption operation. This private key never leaves the TPM. The sender creates the ciphertext by encrypting the plaintext with the public key. Therefore only the device with the specific TPM can decrypt this ciphertext, which is in the possession of the respective private key. Additionally, the policy of the key validates, that the required user or machine is present, who can authorize and initiate the decryption operation.

Typically, the example can be also applied and extended for more comprehensive use cases like Feature Activation or Firmware Update. In these use cases, the backend would encrypt the data with the public key from the TPM and sign the data using a dedicated asymmetric key pair of the backend. After the transmission of the ciphertext and signature to the device, the TPM can perform the validation and decryption of the ciphertext data. The TPM uses the signature verification operation to validate, that the ciphertext was created by the authentic backend. For this operation the dedicated public key from the backend has been stored in the TPM during the provisioning beforehand. This public key is then used to validate the signature from the backend of the data. If this signature validation was successful, the TPM would authorize the decryption operation, so that the plaintext can be installed as software update package or used as a new feature in the device.

In this example, the key usage of a decryption key is authorized with a signature-based policy. The policy requires a signature to authorize the usage of the decryption key. Together with the required signature key, the decryption key is embedded into the hierarchy of a primary key.

The following steps are carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.

2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.
3. Import policy to the current FAPI context
4. Create the decryption key with the policy signed as authorization value
5. Authorize key usage for encryption/decryption by signing a challenge with the private key
6. Encrypt some data
7. Decrypt the previously encrypted data

As seen in Code Listing 76, the JSON-formatted policy is imported into the policy store.

Code Listing 76 Read policy from template and import policy to the FAPI context

```
001         /* Import policy to policy store */
001         r = Fapi_Import(fapi_context, policy_name, json_policy);
002         if (r != TSS2_RC_SUCCESS) {
003             printf("Error: Fapi_Import\n");
004             goto error;
005         }
```

As seen in Code Listing 77, an encryption key is created with the imported policy as authorization value.

Code Listing 77 Create the decryption key with the policy signed as authorization value

```
001         r = Fapi_CreateKey(fapi_context, "HS/SRK/decryptionKey",
002             "noDa, decrypt", policy_name, NULL);
003         if (r != TSS2_RC_SUCCESS) {
004             printf("Error: Fapi_CreateKey\n");
005             goto error;
006         }
```

To authorize key usage, a signature callback is set, where a challenge from the TPM is signed (Code Listing 78).

Code Listing 78 Authorize key usage for encryption/decryption by signing a challenge with the private key

```
001         r = Fapi_SetSignCB(fapi_context,
002             signatureCallback, "");
003         if (r != TSS2_RC_SUCCESS) {
004             printf("Error: Fapi_SetSignCB\n");
005             goto error;
006         }
```

After key usage is authorized, it is used in Code Listing 79 to encrypt the data string `ABC`.

Code Listing 79 Encrypt some data

```
001         uint8_t *plaintext = "ABC";
002         size_t plainTextSize = 4;
003         char *cipherText;
004         size_t cipherTextSize;
005
006         r = Fapi_Encrypt(fapi_context, "HS/SRK/decryptionKey", plaintext
007             plaintextSize, &cipherText &cipherTextSize);
008         if (r != TSS2_RC_SUCCESS) {
009             printf("Error: Fapi_Encrypt\n");
010             goto error;
```

Code Listing 79 Encrypt some data

```
011         }
```

As seen in Code Listing 80, the previously encrypted data is again decrypted.

Code Listing 80 Decrypt the previously encrypted data

```
001         uint8_t *plainText2 = { 0 };
002         size_t plainText2_size;
003
004         r = Fapi_Decrypt(fapi_context, "HS/SRK/decryptionKey",
005             cipherText, cipherTextSize, &plainText2, &plainText2_size);
006         if (r != TSS2_RC_SUCCESS) {
007             printf("Error: Fapi_Decrypt\n");
008             goto error;
009         }
```

6.2.4 Write to and read from NV-Index (readWriteNVIndexPasswordAuth_FAPI.c)

The example shows how security critical data like certificates can be stored in a protected memory area (NV-index) inside the TPM. The NV-index memory area can be configured in such a way that the modification or deletion of this data (e.g., root certificates) is bound to certain conditions or not allowed at all. This programming example shows the generic usage of NV-indices by explaining how to define a password-protected NV-index on the TPM, write data to it, read data from it and undefine it after it was used.

Security-critical data can be stored in the NV-index memory area of the TPM to protect the data from malicious read, write or delete operations. The NV-index can store, e.g., certificates, which are used by the host to verify incoming data. These certificates need to be protected from replacement, so that the data verification process in the host can be performed correctly. In general, the TPM supports more authorization restrictions on NV-indices than passwords. The complex Enhanced Authorization policies functionality can be also applied to NV-indices.

In this example, a region in the TPM's NV memory is defined where read and write access is secured with a required password authorization.

The following steps are carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.
2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.
3. Define a region in the TPM's NV memory with the desired properties, e.g., password protection to read and write (Code Listing 81)
4. Authorize NV write and read usage (Code Listing 82)
5. Write data to previously defined memory region and read it again (Code Listing 83)

In Code Listing 81, a region in the TPM's NV memory is defined with the desired properties, e.g., password protection to read and write.

Code Listing 81 **Define a region in the TPM's NV memory with the desired properties, e.g., password protection to read and write**

```
001         char *nvPathWrite = "/nv/Owner/myNVwrite";
002
003         r = Fapi_CreateNv(fapi_context, nvPathWrite,
004             "noDa", DATA_SIZE, "", password);
005         if (r != TSS2_RC_SUCCESS) {
006             printf("Error: Fapi_CreateNv\n");
007             goto error;
008         }
```

NV usage is authorized by setting a callback (Code Listing 82).

Code Listing 82 **Authorize NV write and read usage**

```
001         r = Fapi_SetAuthCB(fapi_context, auth_callback, "");
002         if (r != TSS2_RC_SUCCESS) {
003             printf("Error: Fapi_SetAuthCB\n");
004             goto error;
005         }
```

After NV usage is authorized, some defined data (l.002) is written to the memory region (l.004f) and read again from that region (l.015f).

Code Listing 83 **Write data to previously defined memory region and read it again**

```
001         /* Write data to the previously defined NV memory region */
002         uint8_t data_write[DATA_SIZE] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
003
004         r = Fapi_NvWrite(fapi_context, nvPathWrite, &data_write[0],
005             DATA_SIZE);
006         if (r != TSS2_RC_SUCCESS) {
007             printf("Error: Fapi_NvWrite\n");
008             goto error;
009         }
010
011         /* Read data from the previously defined NV memory region */
012         uint8_t *data_read;
013         size_t data_read_size = DATA_SIZE;
014         char* log_data;
015         r = Fapi_NvRead(fapi_context, nvPathWrite, &data_read,
016             &data_read_size , &log_data);
017         if (r != TSS2_RC_SUCCESS) {
018             printf("Error: Fapi_NvRead\n");
019             goto error;
020         }
```

6.2.5 Secure Time (secureTime_FAPI.c)

This use case is not covered by FAPI since there is no command in the FAPI to set or get the TPM time. Therefore, Secure Time is one example of the special use cases, which need to be implemented in ESAPI. The example for the Secure Time implementation is shown in the ESAPI programming examples in section 6.3.2.6.

If FAPI and ESAPI commands are used in a single system in parallel for different use cases, it needs to be validated that the executed TPM commands in ESAPI and their effects on the keys stored in the FAPI store are synchronized. For example any changes to the provision data of the FAPI using ESAPI commands can be critical.

A typical method to improve this issues is the usage of independent keys or memory areas for either FAPI or ESAPI use cases, so that there is no cross-corelation between the executed commands.

6.2.6 Key Duplication (keyDuplication_FAPI.c)

In this example, an ECC P256 key stored in the TPM is duplicated. This functionality allows to transfer a key to another device or another TPM. An example would be to create a backup of a key or to enable redundancy of keys for virtualized systems. The TPM specification provides the functionality to select that a specific key is allowed to be duplicated and it also includes a protocol to transfer the key using a protection by an external key. In this example, the duplicate key is re-encrypted with an externally created OpenSSL key.

The duplication of a key is only allowed if the respective property was selected during the key creation. Therefore it needs to be decided at the key creation if a key is duplicable or if the key is a unique key, which is available only to the specific TPM. In this example a key, which allows duplication, is called duplicate key.

The following steps are carried out in this example.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.
2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.
3. Create and export a new decryption key with the parameters "noDa" (no dictionary attack protection), "restricted" (restricted key) and "decrypt" (decryption key) that will be used as new parent (Code Listing 84)
4. Import the exported key as "external" key. The key will be used as new parent (Code Listing 85)
5. Import a policy that allows duplication. The policy template already specifies the key path for the new parent (Code Listing 86)
6. Create a duplicate key with the imported duplication policy (Code Listing 87)
7. Export the duplicateKey with the new parent key (Code Listing 88)

In Code Listing 84, an encryption key is created (l.001f) that will be used as new parent. Thus, its public key is exported (l.009f).

Code Listing 84 Create and export a new decryption key that will be used as new parent

```
001         r = Fapi_CreateKey(fapi_context, "HS/SRK/myCryptKey",
002             "noDa, restricted, decrypt", "", NULL);
003         if (r != TSS2_RC_SUCCESS) {
004             printf("Error: Fapi_CreateKey myCryptKey\n");
005             goto error;
006         }
007
008         char *json_string_pub_key;
009         r = Fapi_ExportKey(fapi_context, "HS/SRK/myCryptKey", NULL,
010             &json_string_pub_key);
011         if (r != TSS2_RC_SUCCESS) {
012             printf("Error: Fapi_ExportKey\n");
013             goto error;
014         }
```

The exported “external” key is imported again in the public key directory (Code Listing 85).

Code Listing 85 Import the exported key as "external" key. The key will be used as new parent

```
001         r = Fapi_Import(fapi_context, "ext/myNewParent",
002             json_string_pub_key);
003         if (r != TSS2_RC_SUCCESS) {
```

Code Listing 85 Import the exported key as "external" key. The key will be used as new parent

```
004             printf("Error: Fapi_Import\n");
005             goto error;
006         }
```

To create a key that allows duplication, a duplication policy is imported to the policy store that already specifies the key path for the new parent (Code Listing 86).

Code Listing 86 Import a policy that allows duplication. The policy template already specifies the key path for the new parent (ext/myNewParent)

```
001         char* policy_duplicate = "/policy/pol_duplicate";
002
003         /* Import policy to policy store */
004         r = Fapi_Import(fapi_context, policy_duplicate, json_policy);
005         if (r != TSS2_RC_SUCCESS) {
006             printf("Error: Fapi_Import\n");
007             goto error;
008         }
```

In Code Listing 87, the duplicate key is created with the imported duplication policy.

Code Listing 87 Create a duplicate key with the imported duplication policy

```
001         r = Fapi_CreateKey(fapi_context, "HS/SRK/duplicateKey",
002             "noDa, restricted, decrypt, exportable", policy_duplicate,
003             NULL);
004         if (r != TSS2_RC_SUCCESS) {
005             printf("Error: Fapi_CreateKey duplicateKey\n");
006             goto error;
007         }
```

Finally, in Code Listing 88, the key is duplicated by encrypting it under the new parent key.

Code Listing 88 Export the duplicateKey with the new parent key

```
001         char *json_duplicate;
002         r = Fapi_ExportKey(fapi_context, "HS/SRK/duplicateKey",
003             "ext/myNewParent", &json_duplicate);
004         if (r != TSS2_RC_SUCCESS) {
005             printf("Error: Fapi_ExportKey\n");
006             goto error;
007         }
```

6.2.7 Quote (executeQuote_FAPI.c)

In this example, the FAPI Quote command is executed by the TPM. This is typically done when a remote party wants to verify the platform's integrity. Among other things, the quote consists of an ECC P256 signature of PCR values defined by the remote party. After the signature is verified by the remote party, the received PCR values need to be compared to a trustworthy reference value.

The following steps are carried out.

1. Initialize FAPI (not shown in Listings). This can be achieved as described in 6.2.1.1.
2. Create the primary key according to the profile set with the configuration file (not shown in Listings). This can be achieved as described in 6.2.1.1.

3. Create the restricted (signature) key with the parameters "restricted" (restricted key), "sign" (signing key) and "noDa" (no dictionary attack protection) (not shown in Listings). This can be achieved as described in 6.2.1.4.
4. Execute quote (Code Listing 89)
5. Get the public key in PEM format from TPM and use FAPI to verify the quote (Code Listing 90)

As seen in Code Listing 89, before the quote is executed (l.017ff), some additional data is specified that should be used as input for quote, e.g., a PCR selection or some user defined qualifying data (l.004ff).

Code Listing 89 Execute quote

```
001         /* Specify additional data that should be used as input for
002          * the quote, e.g., a PCR selection or some user defined
003          * qualifying data. */
001         uint32_t pcrList[1] = { 16 };
002         size_t pcrListSize = 1;
003         uint8_t qualifyingData[20] = {
004             0x67, 0x68, 0x03, 0x3e, 0x21, 0x64, 0x68, 0x24, 0x7b, 0xd0,
005             0x31, 0xa0, 0xa2, 0xd9, 0x87, 0x6d, 0x79, 0x81, 0x8f, 0x8f
006         };
007
008         /* Initialize return variables and execute quote */
009         uint8_t *signature = NULL;
010         size_t signatureSize = 0;
011         char *quoteInfo = NULL;
012         char *pcrEventLog = NULL;
013         char *certificate = NULL;
014         r = Fapi_Quote(fapi_context, pcrList, pcrListSize,
015             "HS/SRK/quoteKey", "TPM-Quote", qualifyingData, 20,
016             &quoteInfo, &signature, &signatureSize, &pcrEventLog,
017             &certificate);
018         if (r != TSS2_RC_SUCCESS) {
019             printf("Error: Fapi_Quote quoteKey\n");
020             goto error;
021         }
```

The quote is again verified in Code Listing 90. Therefore, the public key is exported in a JSON object from the TPM (l.003f) and imported again in the public key directory (l.021). Afterwards, the quote is verified (l.028).

Code Listing 90 Get the public key in PEM format from TPM and use FAPI to verify the quote

```
001         /* Export the public key within a JSON object from the TPM */
002         char *pubkey_pem = NULL;
003         r = Fapi_ExportKey(fapi_context, "HS/SRK/quoteKey", NULL,
004             &pubkey_pem);
005         if (r != TSS2_RC_SUCCESS) {
006             printf("Error: Fapi_Quote quoteKey\n");
007             goto error;
008         }
009
010         /* Get the public PEM key from the returned JSON object */
011         json_object *jso, *jso1;
012         jso = json_tokener_parse(pubkey_pem);
013         r = json_object_object_get_ex(jso, "pem_ext_public", &jso1);
014         if(!r){
015             printf("Error json_object_object_get_ex");
016             return 1;
017         }
```

Code Listing 90 Get the public key in PEM format from TPM and use FAPI to verify the quote

```
018         const char *pem_key = json_object_get_string(jso1);
019
020         /* Import the public key to the public key directory */
021         r = Fapi_Import(fapi_context, "myExtPubKey", pem_key);
022         if (r != TSS2_RC_SUCCESS) {
023             printf("Error Fapi_Import\n");
024             goto error;
025         }
026
027         /* Verify the signed quote */
028         r = Fapi_VerifyQuote(fapi_context,
029             "ext/myExtPubKey", qualifyingData, 20,
030             quoteInfo, signature, signatureSize, pcrEventLog);
031         if (r != TSS2_RC_SUCCESS) {
032             printf("Error Fapi_VerifyQuote\n");
033             goto error;
034         }
035         else{
036             printf("Quote successfully verified!\n");
037         }
```

6.3 ESAPI Usage and Programming

This section describes the usage of the ESAPI instead of the FAPI. The usage of the ESAPI might be required if specific parameters and configurations are not available in FAPI and a more detailed control with the entire commandset of the TPM via ESAPI is needed.

6.3.1 TPM2 Tools with ESAPI

This chapter describes the TPM2.0 Tools which use ESAPI functions. Their prefix is "`tpm2_`".

6.3.1.1 Create a random number

The `tpm2_getrandom` command is used to get a random value from the TPM's hardware random number generator. The length of the random number can be defined using parameters. The retrieved value is printed either to stdout or to a file.

While terminal inputs and outputs are depicted in Figure 22, the template shown in Code Listing 91 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ tpm2_getrandom 4 | hexdump -C
00000000  02 47 bc ed                                       |.G..|
00000004
pi@raspberrypi:~/usageExamples $ tpm2_getrandom 8 | hexdump -C
00000000  b3 61 bb 77 47 bd 8b 96                           |.a.wG...|
00000008
pi@raspberrypi:~/usageExamples $ tpm2_getrandom 8 -o random.out
pi@raspberrypi:~/usageExamples $ hexdump -C random.out
00000000  2d 39 df 48 3b c7 1e 61                           |-9.H;..a|
00000008
```

Figure 22 Command tpm2_getrandom() with tpm2-tools

Code Listing 91 Template for command tpm2_getrandom() with tpm2-tools

```
001         cd /home/pi/usageExamples
002         tpm2_getrandom 4 | hexdump -C
003         tpm2_getrandom 8 | hexdump -C
004         tpm2_getrandom 8 -o random.out
005         hexdump -C random.out
```

6.3.1.2 Sealing data to the TPM

With the TPM, data can be encrypted and then sealed to the platform state, specifically to the TPM's PCR values. This means that the sealed data can be unsealed (decrypted) successfully only if PCR values are identical to the state when the data was sealed.

The sealing method of the TPM2-tools supports data sizes up to 128 Byte. In order to perform sealing of larger bulk data, the method of hybrid cryptography can be used. This method involves a new AES key, which is used

for the bulk data encryption external of the TPM. The AES key is then sealed to the TPM as the low size of the AES key fits to the data size limit of the TPM. In contrast to the respective FAPI seal command described in section 5.1.2 that creates the PCR policy automatically, this policy needs to be created manually with the ESAPI tools to seal the data to a PCR value (not shown in this example).

To demonstrate this, first the data file “data.File” containing data to be sealed (“secret data”) is created. Next, a primary key is generated which initializes the TPM’s key hierarchy. The primary key serves as the parent for the next created key. This key is secured with a password (“def456”) and is defined to seal the data from the file “data.File”. The `hexdump` command verifies that the created key’s private data has been encrypted. Among other things, this private data contains the sealed data and the PCR values with which the key has been created.

The now successfully sealed data should be unsealed again. Therefore, the key holding the sealed data is loaded into the TPM. Afterwards, the `unseal()` command is executed. Parameters are the loaded key handle and the password used for securing the data. Again, a successful unsealing can be examined using `hexdump`.

While terminal inputs and outputs are depicted in Figure 23, the template shown in Code Listing 92 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo "secret data" > data.File
pi@raspberrypi:~/usageExamples $ tpm2_createprimary -C o -g sha256 -G ecc -c pri
mary.context
name-alg:
  value: sha256
  raw: 0xb
...
sym-keybits: 128
x: 3a20cfd35b77434c15df36806e719493d6ae21b778fdfe82abda5cb6e31f6fc5
y: 85c6a0cf9628b8a65e34c125a59e41457adc520642482f14b5a92e56892188c9
pi@raspberrypi:~/usageExamples $ tpm2_create -C primary.context -p def456 -g sha
256 -i data.File -u opu.File -r opr.File
name-alg:
  value: sha256
  raw: 0xb
attributes:
  value: fixedtpm|fixedparent|userwithauth
  raw: 0x52
type:
  value: keyedhash
  raw: 0x8
algorithm:
  value: null
  raw: 0x10
keyedhash: b62bdde5bf4050eaf85e092dddfed0ab833c86e84290124cf2655d0075896a5b
pi@raspberrypi:~/usageExamples $ hexdump -C opr.File
00000000  00 8a 00 20 7d 83 98 a4  4d 16 30 d7 33 e6 04 5a  |... }...M.0.3..Z|
00000010  7d ac d2 71 e4 b5 53 a9  be 6d b2 c7 ac 48 c8 2f  |}..q..S..m...H./|
00000020  a1 d8 1e 60 00 10 d9 47  a9 c0 15 7a f7 4e a2 be  |...`...G...z.N..|
00000030  93 9f da d3 de b3 73 74  a2 9f 37 2a 67 fe 3e 88  |......st..7*g.>.|
00000040  a8 a9 ed 3e 7d a1 e5 e7  e6 d2 fc 22 8c 16 1b f3  |...>}......"....|
00000050  33 53 43 b6 ad 03 a7 c0  d5 7a ae 8f a4 c3 90 a5  |3SC......z......|
00000060  16 f8 97 a0 0a bf f7 4d  a6 4c 53 ce dc b2 92 c2  |.......M.LS.....|
00000070  93 9c 28 4b 7a 24 9a fa  29 18 4f 23 d3 6e 7b 87  |..(Kz$..).O#.n{.|
00000080  c8 27 9e 04 97 81 11 e5  b8 10 80 90              |.'..........|
0000008c
pi@raspberrypi:~/usageExamples $ tpm2_load -C primary.context -u opu.File -r opr
.File -c loadedKey.handle
name: 000b611ec726edd8aebf5f800c11cf2980dc8b8da4d1bdcea09695b94f5f256cc143
pi@raspberrypi:~/usageExamples $ tpm2_unseal -c loadedKey.handle -p def456 -o de
crypted.dat
pi@raspberrypi:~/usageExamples $ hexdump -C decrypted.dat
00000000  73 65 63 72 65 74 20 64  61 74 61 0a              |secret data.|
0000000c
```

Figure 23 Command tpm2_unseal() with tpm2-tools (excerpt)

Code Listing 92 Template for command tpm2_unseal() with tpm2 tools

```
001         cd /home/pi/usageExamples
002         echo "secret data" > data.File
003         tpm2_createprimary -C o -g sha256 -G ecc -c primary.context
004         tpm2_create -C primary.context -p def456 -g sha256 -i data.File \
   -u opu.File -r opr.File
005         hexdump -C opr.File
006         tpm2_load -C primary.context -u opu.File -r opr.File \
   -c loadedKey.handle
007         tpm2_unseal -c loadedKey.handle -p def456 -o decrypted.dat
008         hexdump -C decrypted.dat
```

6.3.1.3 RSA encryption and decryption

With the command `tpm2_rsadecrypt()` the TPM decrypts data with RSA. The method of encryption described in this document supports data sizes for encryption in RSA OAEP mode up to 190 Byte. In order to perform encryption of larger bulk data, the method of hybrid cryptography can be used. This method involves a new AES key, which is used for the bulk data encryption external of the TPM. The AES key is then sealed to the TPM as the low size of the AES key fits to the data size limit of the TPM.

Like in the example before, the data file “data.File” is created containing the sensitive data string “secret data”. Again, a primary key is created under which a RSA key is created. Unlike before, the key does not have sealing data but has the passphrase “def456” to unlock the key.

The created RSA key is loaded into the TPM and is used together with the command `tpm2_rsaencrypt()` to encrypt the file “data.File”. For the purpose of evaluation, the contents of the resulting encrypted file are examined via `hexdump` to verify the successful encryption.

After the data has been encrypted successfully, it will be decrypted again. Therefore, the command `tpm2_rsadecrypt()` is used with the encrypted file and the passphrase as input. The successful decryption of the resulting file “decrypted.File” can be verified via `hexdump`.

While terminal inputs and outputs are depicted in Figure 24, the template shown in Code Listing 93 enables easy copying of the respective commands.

```
pi@raspberrypi:~ $ cd /home/pi/usageExamples
pi@raspberrypi:~/usageExamples $ echo "secret data" > data.File
pi@raspberrypi:~/usageExamples $ tpm2_createprimary -C o -g sha256 -G ecc -c primary.context
name-alg:
  value: sha256
  raw: 0xb
...
sym-keybits: 128
x: 3a20cfd35b77434c15df36806e719493d6ae21b778fdfe82abda5cb6e31f6fc5
y: 85c6a0cf9628b8a65e34c125a59e41457adc520642482f14b5a92e56892188c9
pi@raspberrypi:~/usageExamples $ tpm2_create -C primary.context -p def456 -g sha256 -G rsa -u o
pu.File -r opr.File
name-alg:
  value: sha256
  raw: 0xb
...
c7de53d4c8f72e2b4a8a8d287121690e807ba4ac73a29ffbb83d7d030f96a2b36ad7777a92b1fef0022438cece6373a
3502f4db1d23505e691dcce0e4129da053220e4f73
pi@raspberrypi:~/usageExamples $ tpm2_load -C primary.context -u opu.File -r opr.File -c loaded
Key.handle
name: 000bfc628fc16ecd80108595a73318335fbc6cdf1462f9752a81541351343e97a3b9
pi@raspberrypi:~/usageExamples $ tpm2_rsaencrypt -c loadedKey.handle -o encrypted.File data.Fil
e
pi@raspberrypi:~/usageExamples $ hexdump -C encrypted.File
00000000  73 6a 9f 44 f0 e1 1d 40  33 fa 52 33 b0 c2 96 e8  |sj.D...@3.R3....|
00000010  a4 f6 fd 35 42 42 4a e3  b1 ff 29 b8 fe a4 e9 ea  |...5BBJ...).....|
00000020  89 b3 b5 5e 6e 6f 4a 30  de 4c 3d 99 15 82 6b 3b  |...^noJ0.L=...k;|
00000030  a4 1f 51 9e 95 89 f2 14  56 f4 8b 25 67 b9 68 e1  |..Q.....V..%g.h.|
00000040  b6 d2 3b 5e 4c c9 04 79  e5 3d e5 66 03 9e 2b b5  |..;^L..y.=.f..+.|
00000050  3f 94 1b 4d 09 b1 35 79  ba 7c 78 bf 23 3c cb 7b  |?..M..5y.|x.#<.{|
00000060  13 d4 49 bb 7a 8b 30 d9  7b b9 a6 40 e8 b7 2d 30  |..I.z.0.{..@..-0|
00000070  fe f7 1e bb 83 31 1a 8b  e3 18 11 11 c0 e2 0b 3e  |.....1.........>|
00000080  db 9e 40 8e f6 26 69 c3  0b f9 6b 34 d4 2a 57 35  |..@..&i...k4.*W5|
00000090  89 87 e3 cf ea a5 3f ab  04 cc 78 b5 e0 cb 90 a2  |......?...x.....|
000000a0  e2 72 f0 2e a7 3a 6f 6a  d5 fa 9c 08 0d ed 09 52  |.r...:oj.......R|
000000b0  7b c3 b8 f9 fe 6f e6 2a  7f 5c fa 88 c2 af 75 ce  |{....o.*.\....u.|
000000c0  44 19 87 31 60 d3 67 ba  9c 94 76 47 41 9d ee 41  |D..1`.g...vGA..A|
000000d0  ed 91 55 22 79 40 b8 56  48 23 85 f7 4e 95 13 ed  |..U"y@.VH#..N...|
000000e0  63 92 04 23 8a 6a 19 ac  d0 68 19 b7 39 38 64 24  |c..#.j...h..98d$|
000000f0  24 c0 0e a8 8b 84 c3 ae  a3 07 2f 5c 6e a6 db eb  |$........./\n...|
00000100
pi@raspberrypi:~/usageExamples $ tpm2_rsadecrypt -c loadedKey.handle -p def456 -o decrypted.Fil
e encrypted.File
pi@raspberrypi:~/usageExamples $ hexdump -C decrypted.File
00000000  73 65 63 72 65 74 20 64  61 74 61 0a              |secret data.|
0000000c
```

Figure 24 Command tpm2_rsaencrypt() and tpm2_rsadecrypt() with tpm2-tools (excerpt)

Code Listing 93 Template for command tpm2_rsaencrypt() and tpm2_rsadecrypt() with tpm2-tools

```
001         cd /home/pi/usageExamples
002         echo "secret data" > data.File
003         tpm2_createprimary -C o -g sha256 -G ecc -c primary.context
004         tpm2_create -C primary.context -p def456 -g sha256 -G rsa \
   -u opu.File -r opr.File
005         tpm2_load -C primary.context -u opu.File -r opr.File \
   -c loadedKey.handle
006         tpm2_rsaencrypt -c loadedKey.handle -o encrypted.File data.File
007         hexdump -C encrypted.File
008         tpm2_rsadecrypt -c loadedKey.handle -p def456 -o decrypted.File \
   encrypted.File
009         hexdump -C decrypted.File
```

6.3.2 Programming Examples ESAPI

In the following programming examples using the ESAPI are shown.

6.3.2.1 Compiling code that uses the TSS ESAPI library

This document has several source code files attaced, which provide basic examples of the programming with the ESAPI to implement generic use cases. These programming examples are compiled and explained in the following sections.

Most programming examples using the TSS ESAPI library can be compiled with the command in Code Listing 94. These are 6.3.2.2 (Generation of various keys), 6.3.2.5 (Write to and read from NV-Index (readWriteNVIndexPasswordAuth.c)), 6.3.2.6 (Secure Time (secureTime.c)) and 6.3.2.7 (Key Duplication (keyDuplication.c)). The variable `OWN_PROGRAM` needs to be replaced with the name of the `.c file` that should be compiled. Additional flags only need to be adjusted if the `tpm2-tss` libraries were installed to a different path.

Code Listing 94 Command to compile programs that use the TSS library (6.3.2.2, 6.3.2.5, 6.3.2.6, 6.3.2.7)

```
001        gcc OWN_PROGRAM.c -ltss2-esys -o OWN_PROGRAM
```

The marshalling library (`-ltss2-mu`) is only mandatory for the programming examples in Sections 6.3.2.4 (signatureAuthorization.c) and 6.3.2.8 (executeQuote.c) since they use marshalling functions directly (s. Code Listing 95).

Code Listing 95 Command to compile programs that require the marshalling library (6.3.2.4, 6.3.2.8)

```
002        gcc OWN_PROGRAM.c -ltss2-esys -ltss2-mu -o OWN_PROGRAM
```

Functions of both OpenSSL and tpm2-tss-engine libraries are used in the code example of Section 6.3.2.3 (createSignatureVerifyWithOpenSSL.c). Therefore, `-lcrypto` and `-ltpm2tss` need to be set additionally to compile the code example. This is shown in Code Listing 96.

Code Listing 96 Command to compile programs that use the TSS and OpenSSL library (6.3.2.3)

```
001        gcc OWN_PROGRAM.c -L=/usr/local/lib -ltss2-esys -lcrypto \
   -L/usr/lib/arm-linux-gnueabihf/engines-1.1 -ltpm2tss -o OWN_PROGRAM
```

Note: The programming examples using the ESAPI library can optionally be compiled using the TCTI Loader. An example is shown in the Appendix in Section 8.2.

6.3.2.2 Generation of various keys

This example shows how to generate a Primary Key, a Signature Key, an Encryption Key and a Restricted Key with the TPM. The Signature, Encryption and Restricted Keys are being created with the Primary Key as the parent.

6.3.2.2.1 Primary Key (createPrimaryKey.c)

Note: ESAPI supports to the generation of RSA and ECC keys. However, this programming example uses an RSA key for legacy reasons. It is recommended to switch to ECC for key creation in future developments. An example of an ECC key creation is provided in the example in section 6.3.2.3.

Note:

In this example, a 2048-Bit RSA Primary key is created and secured with an authentication value. A primary key is at the top of the user/storage key hierarchy and used to encrypt later created non-primary keys. Keys protected by a primary key are secured and can be stored outside of the TPM within untrusted memory.

To create a primary key the following steps are carried out.

1. Authenticate with user/storage hierarchy (By default and in this example, the user/storage hierarchy is not protected. Thus, a "NULL" value is used for authentication and may be skipped. This is not shown in the Listings).
2. Specify the sensitive and public parameters for the to-be-created primary key (Code Listing 97).
3. Create the primary key with the respective ESAPI command with the priorly defined sensitive and public parameters as input (Code Listing 98).

As seen in Code Listing 97, the authentication value for the primary key will be “12345” (l.006) which is assigned to the sensitive parameters of the key to be created (l.026). Key details will be set in the public parameter structure (l.034ff), e.g., RSA as a key type (l.037) with a key length of 2048 (l.059). In addition, some key object details are specified in l.039ff, e.g., that the key is secured by the previously defined authentication (l.040) or that the key should be used for decryption (l.042).

Code Listing 97 Specify sensitive and public parameters of the to-be-created primary key

```
/* Initialize data structure for primary key's authentication
 * value.
 */
TPM2B_AUTH authValuePrimary = {
    .size = 5,
    .buffer = {1, 2, 3, 4, 5}
};
/* Initialize data structure for primary key's insensitive
 * parameters.
 */
TPM2B_SENSITIVE_CREATE inSensitivePrimary = {
    .size = 4,
    .sensitive = {
        .userAuth = {
            .size = 0,
            .buffer = {0},
        },
        .data = {
            .size = 0,
            .buffer = {0},
        },
    },
};

/* Set previously defined authentication value.*/
inSensitivePrimary.sensitive.userAuth = authValuePrimary;

/* Initialize data structure for primary key's public
 * parameters. Among other things, the key type (RSA) is defined
 * and some object attributes like a necessary authentication
 * for key usage, that the key may be used as decryption key and
 * that the key is bound to the specific TPM and hierarchy.
 */
TPM2B_PUBLIC inPublic = {
    .size = 0,
    .publicArea = {
        .type = TPM2_ALG_RSA,
        .nameAlg = TPM2_ALG_SHA256,
        .objectAttributes = (
            TPMA_OBJECT_USERWITHAUTH |
            TPMA_OBJECT_RESTRICTED |
            TPMA_OBJECT_DECRYPT |
            TPMA_OBJECT_FIXEDTPM |
            TPMA_OBJECT_FIXEDPARENT |
            TPMA_OBJECT_SENSITIVEDATAORIGIN
         ),
        .authPolicy = {
             .size = 0,
         },
        .parameters.rsaDetail = {
             .symmetric = {
                 .algorithm = TPM2_ALG_AES,
                 .keyBits.aes = 128,
                 .mode.aes = TPM2_ALG_CFB,
             },
             .scheme = {
                  .scheme = TPM2_ALG_NULL
              },
             .keyBits = 2048,
```

Code Listing 97 Specify sensitive and public parameters of the to-be-created primary key

```
060                         .exponent = 65537,
061                     },
062                     .unique.rsa = {
063                         .size = 0,
064                         .buffer = {},
065                     },
066                 },
067
```

Once the parameters for the to-be-created key have been defined in Code Listing 97, the primary key can be created. This is shown in Code Listing 98. First, some additional data may be defined, which will be stored for verification in the key's creation data, e.g., arbitrary text or PCR values (l.004ff), then the return variables will be defined (l.017ff) and finally the ESAPI command `Esys_CreatePrimary` (l.022ff) is executed to create the primary key.

Code Listing 98 Execute ESAPI command to create primary key

```
001             /* Additional information to be included into the key's creation
002              * data.
003              */
004             TPM2B_DATA outsideInfo = {
005                 .size = 0,
006                 .buffer = {},
007             };
008
009             /* Selection of PCRs to be included into the key's creation
010              * data.
011              */
012             TPML_PCR_SELECTION creationPCR = {
013                 .count = 0,
014             };
015
016             /* Define variables for the command's responses*/
017             TPM2B_PUBLIC *outPublic;
018             TPM2B_CREATION_DATA *creationData;
019             TPM2B_DIGEST *creationHash;
020             TPMT_TK_CREATION *creationTicket;
021             /* Execute ESAPI command to create primary key */
022             r = Esys_CreatePrimary(ctx, ESYS_TR_RH_OWNER, ESYS_TR_PASSWORD,
023                                    ESYS_TR_NONE, ESYS_TR_NONE,
024                                    &inSensitivePrimary, &inPublic,
025                                    &outsideInfo, &creationPCR,
026                                    &primaryHandle_handle, &outPublic,
027                                    &creationData, &creationHash,
028                                    &creationTicket);
```

6.3.2.2.2 Signature Key (createSignatureKey.c)

Note: ESAPI supports to the generation of RSA and ECC keys. However, this programming example uses an RSA key for legacy reasons. It is recommended to switch to ECC for key creation in future developments. An example of an ECC key creation is provided in the example in section 6.3.2.3.

In this example, a 2048-Bit RSA signature key is created below a primary key and secured with an authentication value. In general, signature keys are used to sign various data or used within a TPM quote command to sign data used for attestation. The latter case requires a "restricted" signing key and is shown within the “Quote” example (Section 6.3.2.8).

To create a signature key the following steps are carried out.

1-3. If not done yet, create a primary key to be used as a parent for the to-be-created signature key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.
4. Specify the sensitive and public parameters for the to-be-created signature key (Code Listing 99).
5. Create the signature key with the respective ESAPI command with the previously defined sensitive and public parameters as input (Code Listing 100).

As seen in Code Listing 99, and similar to the creation of the primary key, sensitive and public parameters need to be defined also for keys in its hierarchy. The authentication value defined in l.006 and assigned to the sensitive data structure in l.027 has the value “67891011”. Again, the public parameters specify, among other values, the key type RSA (l.038) with a key length of 2048 (l.059). This time the object attributes differ from the primary key since a signing key is created (l.041) now instead of a decryption key.

Code Listing 99 Specify sensitive and public parameters of the to-be-created signature key

```
001             /* Initialize data structure for signature key's authentication
002              * value
003              */
004             TPM2B_AUTH authValue2 = {
005                 .size = 6,
006                 .buffer = {6, 7, 8, 9, 10, 11}
007             };
008
009             /* Initialize data structure for signature key's insensitive
010              * parameters
011              */
012             TPM2B_SENSITIVE_CREATE inSensitive2 = {
013                 .size = 4,
014                 .sensitive = {
015                     .userAuth = {
016                          .size = 0,
017                          .buffer = {0}
018                      },
019                     .data = {
020                         .size = 0,
021                         .buffer = {0}
022                      }
023                 }
024             };
025
026             /* Set previously defined authentication value. */
027             inSensitive2.sensitive.userAuth = authValue2;
028
029             /* Initialize data structure for signature key's public
030              * parameters. Among other things, the key type (RSA) is defined
031              * and some object attributes like a necessary authentication
032              * for key usage, that the key may be used as signature key and
033              * that the key is bound to the specific TPM and hierarchy.
034              */
035             TPM2B_PUBLIC inPublic2 = {
036                 .size = 0,
037                 .publicArea = {
```

Code Listing 99 Specify sensitive and public parameters of the to-be-created signature key

```
038                     .type = TPM2_ALG_RSA,
039                     .nameAlg = TPM2_ALG_SHA1,
040                     .objectAttributes = (TPMA_OBJECT_USERWITHAUTH |
041                                          TPMA_OBJECT_SIGN_ENCRYPT  |
042                                          TPMA_OBJECT_FIXEDTPM |
043                                          TPMA_OBJECT_FIXEDPARENT |
044                                          TPMA_OBJECT_SENSITIVEDATAORIGIN),
045                     .authPolicy = {
046                          .size = 0,
047                      },
048                     .parameters.rsaDetail = {
049                          .symmetric = {
050                              .algorithm = TPM2_ALG_NULL,
051                              .keyBits.aes = 128,
052                              .mode.aes = TPM2_ALG_CFB},
053                          .scheme = {
054                               .scheme = TPM2_ALG_RSAPSS,
055                               .details = {
056                                   .rsapss = { .hashAlg = TPM2_ALG_SHA1 }
057                               }
058                           },
059                          .keyBits = 2048,
060                          .exponent = 65537,
061                      },
062                     .unique.rsa = {
063                          .size = 0,
064                          .buffer = {},
065                      },
066                 },
067             };
```

Again, after all parameters have been set the signature key can be created (Code Listing 100). First, some additional data may be defined, which will later be stored for verification in the key's creation data, e.g. arbitrary text or PCR values (l.004ff), then the return variables will be defined (l.017ff) and finally the ESAPI command `Esys_Create` is executed to create the signature key (l.026ff). Afterwards, the key is loaded into the TPM with `Esys_Load` (l.036ff).

Code Listing 100 Execute ESAPI command to create the previously defined (signature) key and load the key into the TPM

```
001             /* Additional information to be included into the key's creation
002              * data.
003              */
004             TPM2B_DATA outsideInfo2 = {
005                 .size = 0,
006                 .buffer = {},
007             };
008
009             /* Selection of PCRs to be included into the key's creation
010              * data.
011              */
012             TPML_PCR_SELECTION creationPCR2 = {
013                 .count = 0,
014             };
015
016             /* Define variables for the command's responses.*/
017             TPM2B_PUBLIC *outPublic2;
```

Code Listing 100 Execute ESAPI command to create the previously defined (signature) key and load the key into the TPM

```
018             TPM2B_PRIVATE *outPrivate2;
019             TPM2B_CREATION_DATA *creationData2;
020             TPM2B_DIGEST *creationHash2;
021             TPMT_TK_CREATION *creationTicket2;
022
023             /* Execute ESAPI command to create the previously defined
024              * (signature) key.
025              */
026             r = Esys_Create(ctx,
027                             primaryHandle_handle,
028                             ESYS_TR_PASSWORD, ESYS_TR_NONE, ESYS_TR_NONE,
029                             &inSensitive2, &inPublic2,
030                             &outsideInfo2, &creationPCR2,
031                             &outPrivate2, &outPublic2,
032                             &creationData2, &creationHash2,
033                             &creationTicket2);
034
035             /* Load the signature key into the TPM */
036             r = Esys_Load(ctx,
037                           primaryHandle_handle,
038                           ESYS_TR_PASSWORD,
039                           ESYS_TR_NONE,
040                           ESYS_TR_NONE, outPrivate2, outPublic2,
041                           &loadedKeyHandle);
```

6.3.2.2.3 Encryption key with HMAC (createHMACKey.c)

In this example, a symmetric key is created in the TPM below a primary key and secured with an authentication value. The key is considered as a master key, which is used with the HMAC operation to perform symmetric en-/decryption of data.

The generated symmetric key can be used in typical use cases like remote firmware updates and data exchange using only symmetric cryptography. For these use cases it is recommended to store the symmetric master key inside the TPM so that the key remains inside the protected environment and never leaves the TPM. For the en-/decryption of data, a specific AES key is generated each time for specific firmware update or data transfer using a key derivation with the HMAC operation. The derived key is then used as a specific AES key for each data transfer so that it is only a temporary key valid for the current operation. Therefore, this key does not require a high protection level because the impact of a threat is very limited. Therefore the derived AES key can be transmitted to the host, which is able to perform the symmetric decryption operation of the bulk data with performance optimized hardware accelerators.

Figure 25 provides an overview of this architecture. The backend generates a Nonce, which is used with the master key for the HMAC operation to derive the AES key for the current data transfer. The backend then encrypts the data with the derived AES key. After that the encrypted data and the Nonce are transferred of the to the target device. On the target device the Nonce is sent to the TPM, which performs the same HMAC operation with the master key as in the backend system. The resulting derived AES key is identical to the AES key in the backed, so that this key then used inside the host to perform the decryption of the data. This validates, that only the device with the specific TPM and the correct master key can perform the decryption of the data.

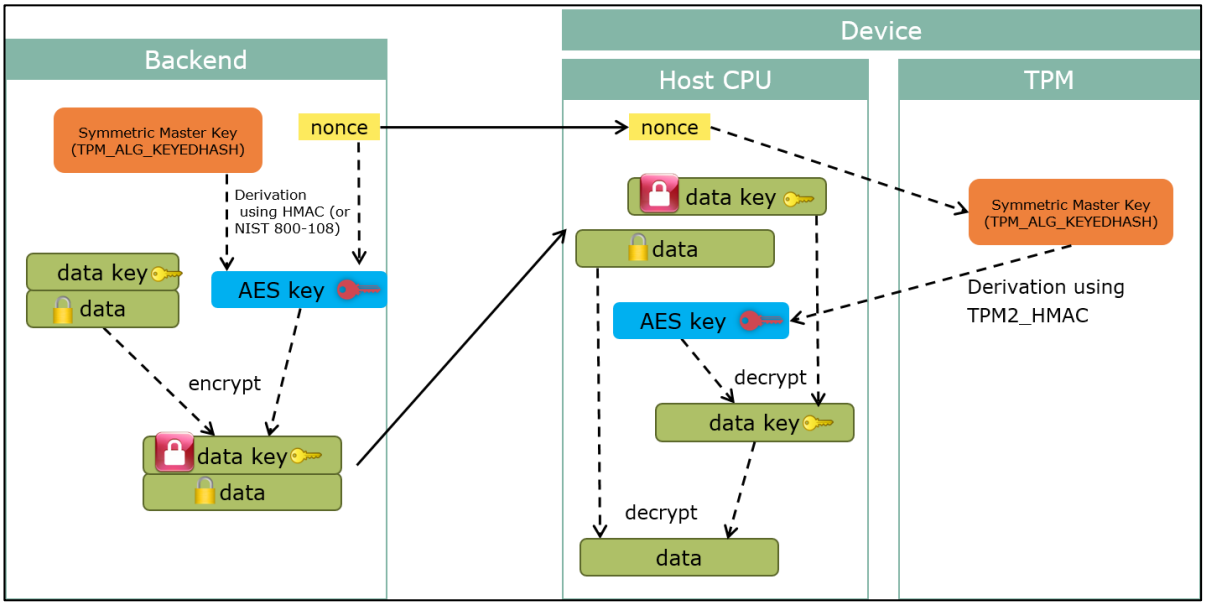


Figure 25 Using a encryption key with HMAC to implement use cases using only symmetric cryptography

To create an HMAC key the following steps are done.

1-3. If not done yet, a primary key is created, which will be used as a parent to the to-be-created HMAC key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.

4. Specify the sensitive and public parameters for the to-be-created HMAC key (Code Listing 101).
5. Create the HMAC key with the respective ESAPI command with the previously defined sensitive and public parameters as input (Code Listing 102).

As seen in Code Listing 101, and similar to the creation of the primary key, sensitive and public parameters need to be defined, also for keys in its hierarchy. The authentication value defined in l.006 and assigned to the sensitive data structure in l.027 has the value "67891011".

Again, the public parameters specify, among other values, the key type keyed-hash (l.039) with a hash function SHA256 (l.048).

Code Listing 101 Specify sensitive and public parameters of the to-be-created HMAC key

```
001             /* Initialize data structure for encryption key's authentication
002              * value.
003              */
004             TPM2B_AUTH authValue2 = {
005                 .size = 6,
006                 .buffer = {6, 7, 8, 9, 10, 11}
007             };
008
009             /* Initialize data structure for HMAC key's insensitive
010              * parameters.
011              */
012             TPM2B_SENSITIVE_CREATE inSensitive2 = {
013                 .size = 0,
014                 .sensitive = {
015                     .userAuth = {
016                         .size = 0,
017                         .buffer = {0}
018                     },
```

Code Listing 101 Specify sensitive and public parameters of the to-be-created HMAC key

```
019                     .data = {
020                          .size = 16,
021                          .buffer = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5}
022                      }
023                 }
024             };
025
026             /* Set previously defined authentication value. */
027             inSensitive2.sensitive.userAuth = authValue2;
028
029             /* Initialize data structure for HMAC key's public
030              * parameters. Among other things, the key type (keyed hash) is
031              * defined and some object attributes like a necessary
032              * authentication for key usage, that the key may be used to
033              * authenticate data and that the key is bound to the specific
034              * TPM and hierarchy.
035              */
036             TPM2B_PUBLIC inPublic2 = {
037               .size = 0,
038               .publicArea = {
039                   .type = TPM2_ALG_KEYEDHASH,
040                   .nameAlg = TPM2_ALG_SHA256,
041                   .objectAttributes = (TPMA_OBJECT_USERWITHAUTH |
042                                        TPMA_OBJECT_SIGN_ENCRYPT),
043                   .authPolicy = {
044                        .size = 0,
045                    },
046                   .parameters.keyedHashDetail.scheme = {
047                        .scheme = TPM2_ALG_HMAC,
048                        .details.hmac.hashAlg = TPM2_ALG_SHA256
049                    },
050                   .unique.sym = {
051                        .size = 0,
052                        .buffer = {}
053                    }
054               }
055           };
```

Again, after all parameters have been set the HMAC key can be created (Code Listing 102). First, some additional data may be defined, which will later be stored for verification in the key's creation data, e.g. arbitrary text or PCR values (l.004ff), then the return variables will be defined (l.017ff) and finally the ESAPI command `Esys_Create` is executed to create the HMAC key (l.026ff). Afterwards, the key is loaded into the TPM with `Esys_Load` (l.039ff).

Code Listing 102 Execute ESAPI command to create the previously defined (encryption) key and load the key into the TPM

```
001             /* Additional information to be included into the key's creation
002              * data.
003              */
004             TPM2B_DATA outsideInfo2 = {
005                 .size = 0,
006                 .buffer = {},
007             };
008
009             /* Selection of PCRs to be included into the key's creation
010              * data.
```

Code Listing 102 Execute ESAPI command to create the previously defined (encryption) key and load the key into the TPM

```
011              */
012             TPML_PCR_SELECTION creationPCR2 = {
013                 .count = 0,
014             };
015
016             /* Define variables for the command's responses.*/
017             TPM2B_PUBLIC *outPublic2;
018             TPM2B_PRIVATE *outPrivate2;
019             TPM2B_CREATION_DATA *creationData2;
020             TPM2B_DIGEST *creationHash2;
021             TPMT_TK_CREATION *creationTicket2;
022
023             /* Execute ESAPI command to create the previously defined
024              * (encryption) key.
025              */
026             r = Esys_Create(ctx,
027                             primaryHandle_handle,
028                             ESYS_TR_PASSWORD, ESYS_TR_NONE, ESYS_TR_NONE,
029                             &inSensitive2,
030                             &inPublic2,
031                             &outsideInfo2,
032                             &creationPCR2,
033                             &outPrivate2,
034                             &outPublic2,
035                             &creationData2, &creationHash2,
036                             &creationTicket2);
037
038             /* Load the encryption key into the TPM.*/
039             r = Esys_Load(ctx,
040                           primaryHandle_handle,
041                           ESYS_TR_PASSWORD,
042                           ESYS_TR_NONE,
043                           ESYS_TR_NONE, outPrivate2, outPublic2,
044                           &loadedKeyHandle);
```

6.3.2.2.4 Restricted Key (createRestrictedKey.c)

Note: ESAPI supports to the generation of RSA and ECC keys. However, this programming example uses an RSA key for legacy reasons. It is recommended to switch to ECC for key creation in future developments. An example of an ECC key creation is provided in the example in section 6.3.2.3.

In this example, a 2048-Bit RSA restricted (signature) key is created below a primary key and secured with an authentication value. Since it is a restricted signature key, only data created by the TPM can be signed. A use case for a restricted (signature) key is signing TPM attestation structures which, among other data, contain a selection of PCR values.

To create a restricted (signature) key the following steps are carried out.

2-3. If not done yet, a primary key is created, which will be used as a parent to the to-be-created restricted (signature) key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.
4. Specify the sensitive and public parameters for the to-be-created primary key (Code Listing 103).
5. Create the restricted signature key with the respective ESAPI command with the previously defined sensitive and public parameters as input (Code Listing 104).

As seen in Code Listing 103, and similar to the creation of the primary key, sensitive and public parameters need to be defined, also for keys in its hierarchy. The authentication value defined in l.006 and assigned to the sensitive data structure in l.027 has the value "`67891011`". Again, the public parameters specify, among other values, the key type RSA (l.041) with a key length of 2048 (l.063). To create a restricted key, the object attribute "`TPMA_OBJECT_RESTRICTED`" needs to be set (l.044).

Code Listing 103 Specify sensitive and public parameters of the to-be-created restricted (signature) key

```
001             /* Initialize data structure for signature key's authentication
002              * value
003              */
004             TPM2B_AUTH authValue2 = {
005                 .size = 6,
006                 .buffer = {6, 7, 8, 9, 10, 11}
007             };
008
009             /* Initialize data structure for signature key's insensitive
010              * parameters
011              */
012             TPM2B_SENSITIVE_CREATE inSensitive2 = {
013                 .size = 4,
014                 .sensitive = {
015                     .userAuth = {
016                          .size = 0,
017                          .buffer = {0}
018                      },
019                     .data = {
020                         .size = 0,
021                         .buffer = {0}
022                      }
023                 }
024             };
025
026             /* Set previously defined authentication value. */
027             inSensitive2.sensitive.userAuth = authValue2;
028
```

Code Listing 103 Specify sensitive and public parameters of the to-be-created restricted (signature) key

```
029             /* Initialize data structure for restricted (signature) key's
030              * public parameters. Among other things, the key type (RSA) is
031              * defined and some object attributes like a necessary
032              * authentication for key usage, that the key may be used as
033              * signature key and that the key is bound to the specific TPM
034              * and hierarchy. Important for this use case is that the
035              * parameter TPMA_OBJECT_RESTRICTED is set to create a
036              * restricted key.
037              */
038             TPM2B_PUBLIC inPublic2 = {
039                 .size = 0,
040                 .publicArea = {
041                     .type = TPM2_ALG_RSA,
042                     .nameAlg = TPM2_ALG_SHA1,
043                     .objectAttributes = (TPMA_OBJECT_USERWITHAUTH |
044                                          TPMA_OBJECT_RESTRICTED |
045                                          TPMA_OBJECT_SIGN_ENCRYPT |
046                                          TPMA_OBJECT_FIXEDTPM |
047                                          TPMA_OBJECT_FIXEDPARENT |
048                                          TPMA_OBJECT_SENSITIVEDATAORIGIN),
049                     .authPolicy = {
050                          .size = 0,
051                      },
052                     .parameters.rsaDetail = {
053                          .symmetric = {
054                              .algorithm = TPM2_ALG_NULL,
055                              .keyBits.aes = 128,
056                              .mode.aes = TPM2_ALG_CFB},
057                          .scheme = {
058                               .scheme = TPM2_ALG_RSAPSS,
059                               .details = {
060                                   .rsapss = { .hashAlg = TPM2_ALG_SHA1 }
061                               }
062                           },
063                          .keyBits = 2048,
064                          .exponent = 65537,
065                      },
066                     .unique.rsa = {
067                          .size = 0,
068                          .buffer = {},
069                      },
070                 },
071             };
```

The restricted (signature) key can be created (Code Listing 104), once all parameters are set. Therefore, some additional data may be defined, which will later be stored for verification in the key's creation data, e.g. arbitrary text or PCR values (l.004ff), then the return variables will be defined (l.017ff) and finally the ESAPI command `Esys_Create` is executed to create the restricted (signature) key (l.026ff). Afterwards, the key is loaded into the TPM with `Esys_Load` (l.039ff).

Code Listing 104 Execute ESAPI command to create the previously defined restricted (signature) key and load the key into the TPM

```
001             /* Additional information to be included into the key's creation
002              * data.
003              */
```

Code Listing 104 Execute ESAPI command to create the previously defined restricted (signature) key and load the key into the TPM

```
004             TPM2B_DATA outsideInfo2 = {
005                 .size = 0,
006                 .buffer = {},
007             };
008
009             /* Selection of PCRs to be included into the key's creation
010              * data.
011              */
012             TPML_PCR_SELECTION creationPCR2 = {
013                 .count = 0,
014             };
015
016             /* Define variables for the command's responses.*/
017             TPM2B_PUBLIC *outPublic2;
018             TPM2B_PRIVATE *outPrivate2;
019             TPM2B_CREATION_DATA *creationData2;
020             TPM2B_DIGEST *creationHash2;
021             TPMT_TK_CREATION *creationTicket2;
022
023             /* Execute ESAPI command to create the previously defined
024              * restricted (signature) key.
025              */
026             r = Esys_Create(ctx,
027                             primaryHandle_handle,
028                             ESYS_TR_PASSWORD, ESYS_TR_NONE, ESYS_TR_NONE,
029                             &inSensitive2,
030                             &inPublic2,
031                             &outsideInfo2,
032                             &creationPCR2,
033                             &outPrivate2,
034                             &outPublic2,
035                             &creationData2, &creationHash2,
036                             &creationTicket2);
037
038             /* Load the restricted (signature) key into the TPM.*/
039             r = Esys_Load(ctx,
040                           primaryHandle_handle,
041                           ESYS_TR_PASSWORD,
042                           ESYS_TR_NONE, ESYS_TR_NONE, outPrivate2,
043                           outPublic2, &loadedKeyHandle);
```

6.3.2.3 Signature generation and validation (createSignatureVerifyWithOpenSSL.c)

In this example, a NIST P256 ECC signature key is created below a primary key and secured with an authentication value. It is used to sign some data digest. The signature is then verified both with the TPM ESAPI command and with OpenSSL.

To create a signature key the following steps are carried out.

1-3. If not done yet, a primary key is created, which will be used as a parent to the to-be-created signature key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.

4-5 If not done yet, the signature key is created (not shown in Listings). This can be achieved as described in Section 6.3.2.2.2.

6. Authenticate with the signature key to authorize key usage and use the TPM ESAPI commands Esys_Sign and Esys_VerifySignature to sign and verify a previously defined digest (Code Listing 105).
7. Verify the TPM generated signature with OpenSSL (Code Listing 106).

After the signature key has been created in the key hierarchy of the previously created primary key, the key usage needs to be authorized as seen in Code Listing 105 (l.004). Afterwards, a to-be-signed digest is defined (l.013ff), the signing scheme is defined (l.021ff) and hash validation token is set to proof that the digest was generated by a TPM (l.028ff). This is only necessary if the signing key is restricted. Thus, the hash validation token is empty in this case. In addition, the signature data structure is initialized (l.035). Having all input parameters set and all output parameters defined, the ESAPI command `Esys_Sign` can be executed (l.040ff). To verify the signature with the TPM, the response variable is defined (l.053ff) and the ESAPI command `Esys_VerifySignature` is executed (l.058f).

Code Listing 105 Authorize key usage and use the TPM ESAPI commands Esys_Sign and Esys_VerifySignature to sign and verify a previously defined digest

```
001             /* Authorize key usage by authenticating the loaded key via
002              * handle and the previously defined authentication value.
003              */
004             r = Esys_TR_SetAuth(ctx, loadedKeyHandle, &authValue2);
005             if (r != TSS2_RC_SUCCESS){
006                 printf("Error: Esys_TR_SetAuth\n");
007                 goto error;
008             }
009
010             /* Define static digest to be signed. Normally digest has to be
011              * created from data to be signed.
012              */
013             TPM2B_DIGEST digest = {
014                 .size = 20,
015                 .buffer = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
016                     0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11,
017                     0x12, 0x13, 0x14}
018             };
019
020             /* Define the signing scheme to use.*/
021             TPMT_SIG_SCHEME inScheme = { .scheme = TPM2_ALG_ECDSA
022             };
023             inScheme.details.ecdsa.hashAlg = TPM2_ALG_SHA1;
024
025             /* Proof that digest was created by the TPM if key is a
026              * restricted key.
027              */
028             TPMT_TK_HASHCHECK hash_validation = {
029                 .tag = TPM2_ST_HASHCHECK,
030                 .hierarchy = TPM2_RH_OWNER,
031                 .digest = {0}
032             };
033
034             /* Define variables for the Esys_Sign command's responses.*/
035             TPMT_SIGNATURE *signature;
036
037             /* Execute ESAPI command to create the signature with the
038              * created key.
039              */
040             r = Esys_Sign(ctx,
```

Code Listing 105 Authorize key usage and use the TPM ESAPI commands Esys_Sign and Esys_VerifySignature to sign and verify a previously defined digest

```
041                 loadedKeyHandle,
042                 ESYS_TR_PASSWORD,
043                 ESYS_TR_NONE,
044                 ESYS_TR_NONE,
045                 &digest,
046                 &inScheme,
047                 &hash_validation,
048                 &signature);
049
050             /* Define variables for the Esys_VerifySignature command's
051              * responses.
052              */
053             TPMT_TK_VERIFIED *validation;
054
055             /* Execute ESAPI command to validate the signature with the
056              * created key.
057              */
058             r = Esys_VerifySignature(ctx,
059                 loadedKeyHandle,
060                 ESYS_TR_NONE,
061                 ESYS_TR_NONE,
062                 ESYS_TR_NONE,
063                 &digest,
064                 signature,
065                 &validation);
```

Alternatively, the created signature can be verified with OpenSSL as shown in Code Listing 106. First an OpenSSL compliable ECC key is created from the TPM object by setting up the `TPM2_DATA` structure (l.002ff) and using the `tpm2tss_ecc_makekey` (l.009) and `EVP_PKEY_get0_EC_KEY` (l.012) commands to finally create the OpenSSL compliant key. With the command, `ECDSA_do_verify` the key can be used to verify the signature with the previously defined digest (l.015ff).

Code Listing 106 Verify signature with OpenSSL

```
001             /* Generate OpenSSL compliable ECC Key.*/
002             TPM2_DATA tpm2Data= {
003                 .userauth = authValue2,
004                 .pub = *outPublic2,
005                 .priv = *outPrivate2,
006             };
007
008             EVP_PKEY *pkey = NULL;
009             pkey = tpm2tss_ecc_makekey(&tpm2Data);
010
011             EC_KEY* eckey = NULL;
012             eckey= EVP_PKEY_get0_EC_KEY(pkey);
013
014             /* Verify Signature via OpenSSL.*/
015             ret = ECDSA_do_verify(digest.buffer, digest.size, sig, eckey);
016             switch(ret){
017                 case -1: printf("Error: ECDSA_do_verify\n"); goto error;
018                 case 0: printf("Invalid Signature!\n"); goto error;
019                 case 1: printf("Valid Signature!\n"); break;
020             }
```

6.3.2.4 Decryption of data using signature-based authorization (signatureAuthorization.c)

Note: ESAPI supports encryption with RSA and ECC keys. However, this programming example uses RSA keys for legacy reasons. It is recommended to switch to ECC for encryption in future developments. An example of an ECC key creation is provided in the example in section 6.3.2.3.

In this example, the key usage of a decryption key is authorized with a policy instead of a password. The policy requires a signature to access the decryption key. Together with the required signature key, the decryption key is embedded into the hierarchy of a primary key. The policy is created with a trial session.

For this example the following steps are carried out.

1-3. If not done yet, a primary key is created, which will be used as a parent to the to-be-created signature key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.
4-5. If not done yet, the signature key is created (not shown in Listings). This can be achieved as described in Section 6.3.2.2.2.
6. A trial session is started, which is used to create the authorization policy. (Code Listing 107).
7. The decryption key is created with the previously created policy as the authentication value (Code Listing 108).
8. Encrypt data with the decryption key that will later be decrypted using policy authorizing with a signing key (not shown in Listings).
9. Authorize key usage by fulfilling policy with signature and decrypt the cipher text (Code Listing 109).

When the primary and signature keys have been created, a trial session is started with the TPM (Code Listing 107). Within the trial session, a new policy is created by sequentially executing the desired policy commands. First, a time limit for authorization is defined (l.004ff) and afterwards the trial session is started (l.027ff). Within the trial session, an ahash value is computed and signed. The ahash is a hash value of the concatenation of defined parameters like expiration time and TPM nonce. Thus, the TPM nonce is retrieved first from the TPM (l.037), then a hash sequence is started where TPM nonce and previously set expiration time are concatenated and hashed (l.042ff). The resulting ahash is signed with the signature key (l.095ff) and the policy is constructed with the ESAPI command PolicySigned (l.114). Finally, the digest of the constructed policy is retrieved which will be used as authentication value for the decryption key (l.133ff).

Code Listing 107 A trial session is started which is used to create the authorization policy

```
021             /* Define expiration time for the later created policy in
022              * seconds. In this case ten years.
023              */
024             INT32 expiration = -(10*365*24*60*60);
025             TPM2B_DIGEST expiration2b;
026             size_t offset = 0;
027             r = Tss2_MU_INT32_Marshal(expiration, &expiration2b.buffer[0],
028               4, &offset);
029             expiration2b.size = offset;
030
031             /* Algorithm and key size for parameter encryption of the
032              * trial session.
033              */
034             TPMT_SYM_DEF symmetricTrial = {.algorithm = TPM2_ALG_AES,
035                                            .keyBits = {.aes = 128},
036                                            .mode = {.aes = TPM2_ALG_CFB}
037             };
038
```

Code Listing 107 A trial session is started which is used to create the authorization policy

```
039             /* Set caller nonce for the trial session. */
040             TPM2B_NONCE nonceCallerTrial= {
041                 .size = 20,
042                 .buffer = {11, 12, 13, 14, 15, 16, 17, 18, 19, 11,
043                            21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
044             };
045
046             /* Start the trial session with the TPM. */
047             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
048                                       ESYS_TR_NONE, ESYS_TR_NONE,
049                                       ESYS_TR_NONE,
050                                       &nonceCallerTrial,
051                                       TPM2_SE_TRIAL, &symmetricTrial,
052                                       TPM2_ALG_SHA1,
053                                       &sessionTrial);
054
055             /* Get the TPM nonce of the trial session. */
056             TPM2B_NONCE *nonceTPM;
057             r = Esys_TRSess_GetNonceTPM(ctx, sessionTrial, &nonceTPM);
058
059             /* Start a hash session with the TPM. */
060             TPMI_ALG_HASH hashAlg = TPM2_ALG_SHA1;
061             TPM2B_AUTH auth = {0};
062             r = Esys_HashSequenceStart(ctx,
063                                        ESYS_TR_NONE,
064                                        ESYS_TR_NONE,
065                                        ESYS_TR_NONE,
066                                        &auth,
067                                        hashAlg,
068                                        &sequenceHandle_handle
069                                        );
070
071             /* Include the TPM nonce into the hash session. */
072             r = Esys_SequenceUpdate(ctx,
073                                     sequenceHandle_handle,
074                                     ESYS_TR_PASSWORD,
075                                     ESYS_TR_NONE,
076                                     ESYS_TR_NONE,
077                                     (const TPM2B_MAX_BUFFER *)nonceTPM
078                                     );
079
080             /* Include the expiration time into the hash session. And hash
081              * the concatenation of TPM nonce and expiration time.
082              */
083             TPM2B_DIGEST *signed_digest;
084             TPMT_TK_HASHCHECK *validation;
085             r = Esys_SequenceComplete(ctx,
086                                       sequenceHandle_handle,
087                                       ESYS_TR_PASSWORD,
088                                       ESYS_TR_NONE,
089                                       ESYS_TR_NONE,
090                                       (const TPM2B_MAX_BUFFER *)
091                                         &expiration2b,
092                                       ESYS_TR_RH_OWNER,
093                                       &signed_digest,
094                                       &validation
095             );
096
097             /* Define the signing scheme to use.*/
```

Code Listing 107 A trial session is started which is used to create the authorization policy

```
098             TPMT_SIG_SCHEME inScheme = { .scheme = TPM2_ALG_NULL };
099
100             /* Proof that digest was created by the TPM if key is a
101              * restricted key.
102              */
103             TPMT_TK_HASHCHECK hash_validation = {
104                 .tag = TPM2_ST_HASHCHECK,
105                 .hierarchy = TPM2_RH_OWNER,
106                 .digest = {0}
107             };
108
109             /* Define variables for the command's responses.*/
110             TPMT_SIGNATURE *signature;
111
112             /* Execute ESAPI command to create the signature with the
113              * created key over the digest (TPM nonce || expiration time).
114              */
115             r = Esys_Sign(
116                 ctx,
117                 loadedSigningKeyHandle,
118                 ESYS_TR_PASSWORD,
119                 ESYS_TR_NONE,
120                 ESYS_TR_NONE,
121                 signed_digest,
122                 &inScheme,
123                 &hash_validation,
124                 &signature);
125
126             /* Execute ESAPI command to include a signed authorization
127              * in the created policy.
128              */
129             TPM2B_DIGEST cpHashA = {0};
130             TPM2B_NONCE policyRef = {0};
131             TPM2B_TIMEOUT *timeout;
132             TPMT_TK_AUTH *policySignedTicket;
133
134             r = Esys_PolicySigned(
135                 ctx,
136                 loadedSigningKeyHandle,
137                 sessionTrial,
138                 ESYS_TR_NONE,
139                 ESYS_TR_NONE,
140                 ESYS_TR_NONE,
141                 nonceTPM,
142                 &cpHashA,
143                 &policyRef,
144                 expiration,
145                 signature,
146                 &timeout,
147                 &policySignedTicket);
148             /* Get the current policy digest of the trial session. The
149              * digest is used as authentication value for the later created
150              * decryption key.
151              */
152             TPM2B_DIGEST *policySignedDigest;
153             r = Esys_PolicyGetDigest(ctx,
154                                      sessionTrial,
155                                      ESYS_TR_NONE,
156                                      ESYS_TR_NONE,
```

Code Listing 107 A trial session is started which is used to create the authorization policy

```
157                                         ESYS_TR_NONE,
158                                         &policySignedDigest);
```

After the policy digest has been retrieved within the trial session, the decryption key can be created with the policy as authentication value. Most of the key creation is done as described in 6.3.2.2.3. As shown in Code Listing 108, the policy digest is used as authentication value (l.38ff).

Code Listing 108 The decryption key is created with the previously created policy as authentication value

```
001             /* Initialize data structure for decryption key's public
002              * parameters. Among other things, the key type (RSA) is defined
003              * and some object attributes like a necessary authentication
004              * for key usage, that the key may be used as decryption key and
005              * that the key is bound to the specific TPM and hierarchy.
006              * The important part is that the authentication token for key
007              * usage is the before created policy.
008              */
009             TPM2B_PUBLIC inPublicDec = {
010                 .size = 0,
011                 .publicArea = {
012                     .type = TPM2_ALG_RSA,
013                     .nameAlg = TPM2_ALG_SHA1,
014                     .objectAttributes = (
015                                         TPMA_OBJECT_DECRYPT |
016                                         TPMA_OBJECT_FIXEDTPM |
017                                         TPMA_OBJECT_FIXEDPARENT |
018                                         TPMA_OBJECT_SENSITIVEDATAORIGIN
019                                         ),
020                     .authPolicy = {
021                          .size = policySignedDigest->size,
022                      },
023                     .parameters.rsaDetail = {
024                          .symmetric = {
025                              .algorithm = TPM2_ALG_NULL},
026                          .scheme = { .scheme = TPM2_ALG_RSAES },
027                          .keyBits = 2048,
028                          .exponent = 65537,
029                      },
030                     .unique.rsa = {
031                          .size = 0,
032                          .buffer = {},
033                      },
034                 },
035             };
036
037             /* Set previously defined policy as authentication value. */
038             memcpy(&inPublicDec.publicArea.authPolicy.buffer[0],
039                 (const BYTE *)&policySignedDigest->buffer[0],
040                policySignedDigest->size);
```

The created decryption key is then used to encrypt some data. Afterwards, as shown in Code Listing 109, the policy is used to authorize key usage and decrypt the text again. Therefore, a policy session is started first (l.002ff). Within this policy session, the steps from the trail session are recreated to reproduce the policy steps. This means that the retrieved TPM nonce (l.009) and expiration time are hashed (l.012ff) to create the ahash that again is signed (l.062ff) and finally presented within the `PolicySigned` command (l.076ff). This fulfills the policy, and key usage is authorized.

Code Listing 109 Authorize key usage by fulfilling policy with signature and decrypt the ciphertext

```
001             /* Start the policy session with the TPM. */
002             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
003                                       ESYS_TR_NONE, ESYS_TR_NONE,
004                                       ESYS_TR_NONE, &nonceCaller,
005                                       TPM2_SE_POLICY, &symmetric,
006                                       TPM2_ALG_SHA1, &session);
007
008             /* Get the TPM nonce of the policy session. */
009             r = Esys_TRSess_GetNonceTPM(ctx, session, &nonceTPM);
010
011             /* Start a hash session with the TPM. */
012            r = Esys_HashSequenceStart(ctx,
013                                       ESYS_TR_NONE,
014                                       ESYS_TR_NONE,
015                                       ESYS_TR_NONE,
016                                       &auth,
017                                       hashAlg,
018                                       &sequenceHandle_handle
019                                       );
020
021             /* Include the TPM nonce into the hash session. */
022             r = Esys_SequenceUpdate(ctx,
023                                     sequenceHandle_handle,
024                                     ESYS_TR_PASSWORD,
025                                     ESYS_TR_NONE,
026                                     ESYS_TR_NONE,
027                                     (const TPM2B_MAX_BUFFER *)nonceTPM
028                                     );
029
030             /* Include the expiration time into the hash session. And hash
031              * the concatenation of TPM nonce and expiration time.
032              */
033             r = Esys_SequenceComplete(ctx,
034                                       sequenceHandle_handle,
035                                       ESYS_TR_PASSWORD,
036                                       ESYS_TR_NONE,
037                                       ESYS_TR_NONE,
038                                     (const TPM2B_MAX_BUFFER *)&expiration2b,
039                                       ESYS_TR_RH_OWNER,
040                                       &signed_digest,
041                                       &validation
042             );
043
044             /* Define the signing scheme to use.*/
045             TPMT_SIG_SCHEME inScheme2 = { .scheme = TPM2_ALG_NULL };
046
047             /* Proof that digest was created by the TPM if key is a
048              * restricted key.
049              */
050             TPMT_TK_HASHCHECK hash_validation2 = {
051                 .tag = TPM2_ST_HASHCHECK,
052                 .hierarchy = TPM2_RH_OWNER,
053                 .digest = {0}
054             };
055
056             /* Define variables for the command's responses.*/
057             TPMT_SIGNATURE *signature3;
058
059             /* Execute ESAPI command to create the signature with the
```

Code Listing 109 Authorize key usage by fulfilling policy with signature and decrypt the ciphertext

```
060              * created key over the digest (TPM nonce || expiration time).
061              */
062             r = Esys_Sign(
063                 ctx,
064                 loadedSigningKeyHandle,
065                 ESYS_TR_PASSWORD,
066                 ESYS_TR_NONE,
067                 ESYS_TR_NONE,
068                 signed_digest,
069                 &inScheme2,
070                 &hash_validation2,
071                 &signature3);
072
073             /* Execute ESAPI command to fulfill the signature authorization
074              * policy.
075              */
076             r = Esys_PolicySigned(
077                 ctx,
078                 loadedSigningKeyHandle,
079                 session,
080                 ESYS_TR_NONE,
081                 ESYS_TR_NONE,
082                 ESYS_TR_NONE,
083                 nonceTPM,
084                 &cpHashA,
085                 &policyRef,
086                 expiration,
087                 signature3,
088                 &timeout,
089                 &policySignedTicket);
```

6.3.2.5 Write to and read from NV-Index (readWriteNVIndexPasswordAuth.c)

In this example, a region in the TPM's NV memory is defined where read and write access are secured with a required authorization.

Attention: ***The NVRAM is non-volatile memory. In this example, the defined space is freed again with "`Esys_NV_UndefineSpace`". Should the NVRAM erroneously remain occupied, e.g., due to a programming error, the TPM's NVRAM can be cleared with the program mentioned in Section 6.1.3.***

The following steps are carried out.

1. Define a region in the TPM's NV memory with the desired properties, e.g., required authentication to write to and read from the region (Code Listing 110).
2. Use the authenticated handle to write data to the previously defined memory region (Code Listing 111).
3. Use the authenticated handle to read previously written data from the defined memory region (also Code Listing 111).

As seen in Code Listing 110, first an authentication value is defined for the later defined protected memory region (l.004ff). Afterwards, within the public parameters, attributes of the memory region are defined, e.g., required read and write protection (l.015ff). Finally, the memory region is defined with the corresponding ESAPI command and the authentication value and public parameters as input (l.039ff).

Code Listing 110 Define a region in the TPM's NV memory with the desired properties

```
001             /* Define the authentication value required to authorize
002              * writing to the to-be-defined NV space.
003              */
004             TPM2B_AUTH auth = {
005                 .size = 20,
006                 .buffer={10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
007                     23, 24, 25, 26, 27, 28, 29}
008             };
009
010             /* Define public parameters for the to-be-defined NV space.
011              * Among others, define that authentication is necessary to read
012              * from and write to the NV space. Additionally, define the
013              * index size.
014              */
015             TPM2B_NV_PUBLIC publicInfo = {
016                 .size = 0,
017                 .nvPublic = {
018                     .nvIndex = TPM2_NV_INDEX_FIRST,
019                     .nameAlg = TPM2_ALG_SHA1,
020                     .attributes = (
021                         TPMA_NV_OWNERWRITE |
022                         TPMA_NV_AUTHWRITE |
023                         TPMA_NV_WRITE_STCLEAR |
024                         TPMA_NV_READ_STCLEAR |
025                         TPMA_NV_AUTHREAD |
026                         TPMA_NV_OWNERREAD
027                         ),
028                     .authPolicy = {
029                          .size = 0,
030                          .buffer = {},
031                     },
032                 .dataSize = 32,
033                 }
034             };
035
036             /* Execute ESAPI command to create the NV space according to the
037              * public parameters defined earlier.
038              */
039             r = Esys_NV_DefineSpace(ctx,
040                 ESYS_TR_RH_OWNER,
041                 ESYS_TR_PASSWORD,
042                 ESYS_TR_NONE,
043                 ESYS_TR_NONE,
044                 &auth,
045                 &publicInfo,
046                 &nvHandle_handle);
```

As seen in Code Listing 111, the already authenticated NV handle is used to write data to and read data from the memory region. First, some defined test data (l.003ff) is written to the memory with the ESAPI command (l.011ff). Afterwards, the written data again is read out (l.021ff) and the memory region is cleared (l.047ff).

Code Listing 111 Write data to and read data from the previously defined memory region

```
001             /* Define some test data to write to the NV.*/
002             UINT16 offset = 0;
003             TPM2B_MAX_NV_BUFFER nv_test_data = {
004                 .size = 20,
005                 .buffer={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6,
```

Code Listing 111 Write data to and read data from the previously defined memory region

```
006                     7, 8, 9}};
007
008             /* Execute ESAPI command to write a value to an area in NV
009              * memory that was previously defined.
010             .*/
011             r = Esys_NV_Write(ctx,
012                 nvHandle_handle,
013                 nvHandle_handle,
014                 ESYS_TR_PASSWORD,
015                 ESYS_TR_NONE,
016                 ESYS_TR_NONE,
017                 &nv_test_data, offset);
018
019             /* Execute ESAPI command to read an area in NV memory.*/
020             TPM2B_MAX_NV_BUFFER *nv_test_data2;
021             r = Esys_NV_Read(ctx,
022                 nvHandle_handle,
023                 nvHandle_handle,
024                 ESYS_TR_PASSWORD,
025                 ESYS_TR_NONE,
026                 ESYS_TR_NONE,
027                 20,
028                 0,
029                 &nv_test_data2);
030
031             if (r != TSS2_RC_SUCCESS){
032                 printf("Error: Esys_NV_Read\n");
033                 goto error;
034             }
035             else{
036                 int i = 0;
037                 while(i < nv_test_data.size){
038                     printf("%d", nv_test_data2->buffer[i]);
039                     i++;
040                 }
041                 printf(" was read from Index.\n");
042             }
043
044             /* Execute ESAPI command to remove the defined memory region
045              * from the TPM.
046              */
047             r = Esys_NV_UndefineSpace(ctx,
048                 ESYS_TR_RH_OWNER,
049                 nvHandle_handle,
050                 ESYS_TR_PASSWORD,
051                 ESYS_TR_NONE,
052                 ESYS_TR_NONE
053                 );
```

6.3.2.6 Secure Time (secureTime.c)

Note: ESAPI supports to the generation of RSA and ECC keys. However, this programming example uses an RSA key for legacy reasons. It is recommended to switch to ECC for key creation in future developments. An example of an ECC key creation is provided in the example in section 6.3.2.3.

In this example, the flexible usage of policies with the TPM2_PolicyAuthorize command with Secure Time is shown. Especially, the TPM2_PolicyAuthorize allows creating additional policies even after the key has already been created. In this use case, a policy with time constraints is used to authorize key usage. Therefore, two keys are created in the hierarchy of a primary key. One key is authorized to sign additional policies, the other is bound to TPM2_PolicyAuthorize and key usage will only be permitted by additional policies that are signed by the specific signing key. The first authorization is successful since the policies time is below the TPMs. The second authorization fails since the TPM's time is now greater than the one defined in the policy.

For this example the following steps are carried out.

1-3. If not done yet, a primary key is created, which will be used as a parent to the to-be-created signature key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.

4-5 If not done yet, the signature key is created (not shown in Listings). This can be achieved as described in Section 6.3.2.2.2.

6. Use a trial session to create a policy with PolicyAuthorize. The name of the signing key will be passed and used to sign new policies. (Code Listing 112).
7. The decryption key is created with the previously created policy as the authentication value (not shown in Listings). This can be achieved as described in Section 6.3.2.4.
8. Create a new policy with time restrictions that will be signed with the previously created signature key. Additionally, it has a time restriction of 5 seconds (Code Listing 113).
9. Use the time-restricted policy to authorize key usage. Before the key is unlocked, it will be verified that the policy is signed by the right key and that the time is still valid. (Code Listing 114).
10. For verification: Wait until the TPM clock is higher than the policy. Then a new policy session is started to authorize key usage with the time-restricted policy. Since the TPM's clock is now higher than the value in the policy, access to the key is denied (not shown in Listings).

11. Similar to Section 6.3.2.4, here a trial session is started as well to create the policy (Code Listing 112, l.017ff). First, the key name of the signature key is read out (l.028ff) and used as a parameter for the `PolicyAuthorize` command (l.047ff). Afterwards, the policy digest is retrieved (l.064ff) and is used as an authentication value for the afterwards created decryption key. The latter is not shown in the Listings.

Code Listing 112 Use a trial session to create a policy with PolicyAuthorize

```
001             /* Algorithm and key size for parameter encryption of the
002              * trial session.
003              */
004             TPMT_SYM_DEF symmetricTrial = {.algorithm = TPM2_ALG_AES,
005                                            .keyBits = {.aes = 128},
006                                            .mode = {.aes = TPM2_ALG_CFB}
007             };
008
009             /* Set caller nonce for the trial session. */
010             TPM2B_NONCE nonceCallerTrial = {
011                 .size = 20,
012                 .buffer = {11, 12, 13, 14, 15, 16, 17, 18, 19, 11,
013                            21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
014             };
015
016             /* Start the trial session with the TPM. */
017             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
018                                       ESYS_TR_NONE, ESYS_TR_NONE,
019                                       ESYS_TR_NONE, &nonceCallerTrial,
020                                       TPM2_SE_TRIAL, &symmetricTrial,
021                                       TPM2_ALG_SHA1, &sessionTrial);
022
```

Code Listing 112 Use a trial session to create a policy with PolicyAuthorize

```
023             /* Get the name of the signature key. The key name will be
024              * embedded into the policy.
025              */
026             TPM2B_NAME *nameKeySign;
027             TPM2B_NAME *keyQualifiedName;
028             r = Esys_ReadPublic(ctx,
029                 loadedSigningKeyHandle,
030                 ESYS_TR_NONE,
031                 ESYS_TR_NONE,
032                 ESYS_TR_NONE,
033                 &outPublic,
034                 &nameKeySign,
035                 &keyQualifiedName);
036
037             /* Execute ESAPI command to include the signature key name
038              * in the created policy.
039              */
040             TPM2B_DIGEST approvedPolicy = {0};
041             TPM2B_NONCE policyRef = {0};
042             TPMT_TK_VERIFIED  checkTicket = {
043                 .tag = TPM2_ST_VERIFIED,
044                 .hierarchy = TPM2_RH_OWNER,
045                 .digest = {0}
046             };
047             r = Esys_PolicyAuthorize(
048                 ctx,
049                 sessionTrial,
050                 ESYS_TR_NONE,
051                 ESYS_TR_NONE,
052                 ESYS_TR_NONE,
053                 &approvedPolicy,
054                 &policyRef,
055                 nameKeySign,
056                 &checkTicket
057                 );
058
059             /* Get the current policy digest of the trial session. The
060              * digest is used as authentication value for the later created
061              * decryption key.
062              */
063             TPM2B_DIGEST *policyAuthorizeDigest;
064             r = Esys_PolicyGetDigest(ctx,
065                                      sessionTrial,
066                                      ESYS_TR_NONE,
067                                      ESYS_TR_NONE,
068                                      ESYS_TR_NONE,
069                                      &policyAuthorizeDigest);
```

After the decryption key has been created with the `PolicyAuthorize` policy as the authentication value, any policy that is signed with the signature key embedded into the `PolicyAuthorize` policy of the decryption key can authorize key usage. Thus, a new policy is created which has a time restriction and is signed with the signature key.

First, as seen in Code Listing 113, the current time is read from the TPM (l.003ff) and 5000 ms are added (l.012). So, the time restriction will be 5 seconds. Again, a trial session is started to construct the new policy (l.026ff). The time restriction is embedded into the policy with the `PolicyCounterTimer` command that takes the expiration time and operand to compare, which is in this case “<” (l.046f). This means the current time needs to

be lower than the value in the policy. Afterwards, the policy digest is retrieved (l.052ff). The policy digest is hashed into a required ahash value (l.071ff) and then signed with the signature key (l.128ff).

Code Listing 113 Create a new policy with time restrictions that will be signed with the before created signature key

```
001             /* Get current TPM time.*/
002             TPMS_TIME_INFO *currentTime;
003             r = Esys_ReadClock(ctx,
004                                ESYS_TR_NONE,
005                                ESYS_TR_NONE,
006                                ESYS_TR_NONE,
007                                &currentTime);
008
009             /* Add 5000ms to the current TPM time. This will be the
010              * validity of the policy.
011              */
012             UINT64 newTime = currentTime->time + 5000;
013             printf("Time in policy: %" PRId64 "\n", newTime);
014
015             BYTE bytes[8];
016             bytes[0] = (newTime >> 56) & 0xFF;
017             bytes[1] = (newTime >> 48) & 0xFF;
018             bytes[2] = (newTime >> 40) & 0xFF;
019             bytes[3] = (newTime >> 32) & 0xFF;
020             bytes[4] = (newTime >> 24) & 0xFF;
021             bytes[5] = (newTime >> 16) & 0xFF;
022             bytes[6] = (newTime >> 8) & 0xFF;
023             bytes[7] = newTime & 0xFF;
024
025             /* Start a new trial session with the TPM. */
026             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
027               ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE,
028               &nonceCallerTrial,
029               TPM2_SE_TRIAL, &symmetricTrial, TPM2_ALG_SHA1,
030               &sessionTrial);
031
032             /* Execute ESAPI command to include time restriction in the
033              * to-be-created policy.
034              * Esys_PolicyCounterTimer requires an operand to compare the
035              * time with. In this case, "<" is used which means current
036              * time must be lower than policy time to valid key access.
037              */
038             UINT16 offset = 0;
039             TPM2_EO operation = TPM2_EO_SIGNED_LT;
040             TPM2B_OPERAND operandB = {
041                 .size = 8,
042                 .buffer = 0,
043             };
044             memcpy(operandB.buffer,(const uint8_t *)&bytes,operandB.size);
045
046             r = Esys_PolicyCounterTimer(ctx,
047                 sessionTrial, ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE,
048                 &operandB, offset, operation);
049
050             /* Get the current policy digest of the policy.*/
051             TPM2B_DIGEST *policyCounterTimerDigest;
052             r = Esys_PolicyGetDigest(ctx,
053                                      sessionTrial,
054                                      ESYS_TR_NONE,
```

Code Listing 113 Create a new policy with time restrictions that will be signed with the before created signature key

```
055                                     ESYS_TR_NONE,
056                                     ESYS_TR_NONE,
057                                     &policyCounterTimerDigest);
058
059             /* Since Policy Authorize uses an ahash to verify the policy,
060              * the ahash is calculated in the following.
061              * Basically, ahash is a hash of the concatenation of the
062              * approved policy (policyCounterTimerDigest) and an optional
063              * policy reference, in this case "0".
064              * aHash := H(approvedPolicy || policyRef)
065              */
066
067             /* Start a hash session with the TPM. */
068             TPMI_ALG_HASH hashAlg = TPM2_ALG_SHA1;
069             TPM2B_AUTH auth = {0};
070             ESYS_TR sequenceHandle_handle;
071             r = Esys_HashSequenceStart(ctx,
072               ESYS_TR_NONE,
073               ESYS_TR_NONE,
074               ESYS_TR_NONE,
075               &auth,
076               hashAlg,
077               &sequenceHandle_handle
078             );
079
080             /* Convert policyCounterTimerDigest to a structure the TPM can
081              * hash.
082              */
083             TPM2B_MAX_BUFFER buffer = {
084               .size = policyCounterTimerDigest->size,
085             };
086
087             memcpy(buffer.buffer,
088               (const uint8_t *)&policyCounterTimerDigest->buffer,
089               policyCounterTimerDigest->size);
090
091             r = Esys_TR_SetAuth(ctx, sequenceHandle_handle, &auth);
092
093             /* Include the buffer (converted policyCounterTimerDigest)
094              * into the hash session and complete the hashing.
095              */
096             TPM2B_DIGEST *ahash;
097             TPMT_TK_HASHCHECK *validation;
098
099             r = Esys_SequenceComplete(ctx,
100               sequenceHandle_handle,
101               ESYS_TR_PASSWORD,
102               ESYS_TR_NONE,
103               ESYS_TR_NONE,
104               &buffer,
105               ESYS_TR_RH_OWNER,
106               &ahash,
107               &validation
108             );
109
110             /* Define the signing scheme to use.*/
111             TPMT_SIG_SCHEME inScheme = { .scheme = TPM2_ALG_NULL };
112
```

Code Listing 113 Create a new policy with time restrictions that will be signed with the before created signature key

```
113             /* Proof that digest was created by the TPM if key is a
114              * restricted key.
115              */
116             TPMT_TK_HASHCHECK hash_validation = {
117               .tag = TPM2_ST_HASHCHECK,
118               .hierarchy = TPM2_RH_OWNER,
119               .digest = {0}
120             };
121
122             /* Define variables for the command's responses.*/
123             TPMT_SIGNATURE *policyCounterTimerSignature;
124
125             /* Execute ESAPI command to create the signature with the
126              * created key over the ahash.
127              */
128             r = Esys_Sign(ctx,
129               loadedSigningKeyHandle,
130               ESYS_TR_PASSWORD,
131               ESYS_TR_NONE,
132               ESYS_TR_NONE,
133               ahash,
134               &inScheme,
135               &hash_validation,
136               &policyCounterTimerSignature);
```

The new policy can be used to authorize key usage. As seen in Code Listing 114, first the policy session is started with the TPM (l.17ff). Then the TPM's internal policy digest is updated with the `PolicyCounterTimer` policy (l.026ff) and the signature of the policy digest is verified (l.033ff). The verification creates a validation ticket that is used with the `PolicyAuthorize` command to finally authorize key usage (l.047ff).

Code Listing 114 Use the time-restricted policy to authorize key usage

```
001             /* Algorithm and key size for parameter encryption of the
002              * policy session.
003              */
004             TPMT_SYM_DEF symmetric = {.algorithm = TPM2_ALG_AES,
005                                       .keyBits = {.aes = 128},
006                                       .mode = {.aes = TPM2_ALG_CFB}
007             };
008
009             /* Set caller nonce for the policy session. */
010             TPM2B_NONCE nonceCaller = {
011                 .size = 20,
012                 .buffer = {10, 12, 13, 14, 15, 16, 17, 18, 19, 11,
013                            21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
014             };
015
016             /* Start the policy session with the TPM. */
017             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
018                                       ESYS_TR_NONE, ESYS_TR_NONE,
019                                       ESYS_TR_NONE, &nonceCaller,
020                                       TPM2_SE_POLICY, &symmetric,
021                                       TPM2_ALG_SHA1, &session);
022
023             /* Update TPM's internal policy digest with the
024              * PolicyCounterTimer policy.
025              */
```

Code Listing 114 Use the time-restricted policy to authorize key usage

```
026             r = Esys_PolicyCounterTimer(ctx,
027                 session, ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE,
028                 &operandB, offset, operation);
029             /* Verify the signature of the policy to create ticket that is
030              * used within the PolicyAuthorize command.
031              */
032             TPMT_TK_VERIFIED *validationCounterTimerTicket;
033             r = Esys_VerifySignature(ctx,
034               loadedSigningKeyHandle,
035               ESYS_TR_NONE,
036               ESYS_TR_NONE,
037               ESYS_TR_NONE,
038               ahash,
039               policyCounterTimerSignature,
040               &validationCounterTimerTicket);
041
042             /* Update TPM's internal policy digest with the PolicyAuthorize
043              * policy. If both the ticket is valid and TPM time is below the
044              * policy time, access to the key is granted.
045              */
046             TPM2B_DIGEST policyRef2 = {0};
047             r = Esys_PolicyAuthorize(
048                     ctx,
049                     session,
050                     ESYS_TR_NONE,
051                     ESYS_TR_NONE,
052                     ESYS_TR_NONE,
053                     policyCounterTimerDigest,
054                     &policyRef2,
055                     nameKeySign,
056                     validationCounterTimerTicket
057                     );
058                 if (r != TSS2_RC_SUCCESS){
059                     printf("Error: PolicyAuthorize\n");
060                     goto error;
061                 }
062                 else{
063                   printf("PolicyAuthorize success.\n");
064                 }
```

6.3.2.7 Key Duplication (keyDuplication.c)

In this example, a key is duplicated, which enables the transfer of a specific key to another device or TPM. An example would be to create a backup of a key or to enable redundancy of keys for virtualized systems. Thereby, the duplicate key is reencrypted with an externally created OpenSSL key for the protected transfer.

For this example the following steps are carried out.

1. Create a policy that allows duplication. This policy will be bound to the later created duplicate key (Code Listing 116).

2-4. If not done yet, a primary key is created, which will be used as a parent to the to-be-created duplicate key. This can be achieved as described in Section 6.3.2.2.1.

5-6. Create a duplicate key (not shown in Listings).

7. Load a public RSA key generated externally with OpenSSL. This will be the new parent of the duplicate key. It will reencrypt the duplicate key (Code Listing 117).
8. Start a new policy session to authorize key duplication (Code Listing 118).

First, the OpenSSL RSA key needs to be created with the script delivered with this Application Note (`openSSLKeyGenerationScript.sh`) and shown in Code Listing 115. The script can be executed with the command "`sh openSSLKeyGenerationScript.sh`" followed by an optional argument to specify the key name.

Code Listing 115 openSSLKeyGenerationScript.sh

```
001  #!/bin/sh
002  #
003  # This script uses OpenSSL to create a 2048-Bit RSA key pair. The
004  # public key will be converted to binary in order to convert it to a
005  # TPM readable format from the correspondig C program
006  # (keyDuplication.c).
007
008  # Input file name
009  key_file=$1
010
011  # OpenSSL command to create the private 2048-Bit RSA key.
012  openssl genpkey -algorithm RSA -out ${key_file}_priv.pem -pkeyopt \
013      rsa_keygen_bits:2048 -pkeyopt rsa_keygen_pubexp:65537
014
015  # OpenSSL command to create the public RSA key from the private
016  # RSA key.
017  openssl rsa -pubout -in ${key_file}_priv.pem -out \
018      ${key_file}_pub.pem
019
020  # Read public key into variable rsa_pub_key.
021  rsa_pub_key=$(openssl rsa -pubin -inform PEM -text -noout < \
022      ${key_file}_pub.pem )
023
024  # Extract public key's hexadecimal representation (create hexdump).
025  buffer=$(echo $rsa_pub_key |
026      sed ' {
027          s/^.*Modulus[^0]*00://
028          s/ Exponent.*$//
029          s/[[:blank:]]//g
030      }')
031
032  # Convert public key's hexdump into binary format and save.
033  echo $buffer | xxd -r -p > key_pub.tpm2
```

After the key has been created with the script from Code Listing 115, a trial session is used to create a policy that allows duplicating the key (Code Listing 116). As usual, the trial session is first started (l.017ff), then the desired policy commands that build the policy are executed. Especially, the command PolicyCommandCode is relevant (l.036f) since it binds key usage to exactly the specified command, in this case key duplication. Afterwards, the policy digest is retrieved (l.049ff).

Code Listing 116 Create a policy that allows duplication

```
001          /* Algorithm and key size for parameter encryption of the
002           * trial session.
003           */
004          TPMT_SYM_DEF symmetricTrial = {.algorithm = TPM2_ALG_AES,
005                                         .keyBits = {.aes = 128},
006                                         .mode = {.aes = TPM2_ALG_CFB}
007          };
008
009          /* Set caller nonce for the trial session. */
010          TPM2B_NONCE nonceCallerTrial = {
```

Code Listing 116 Create a policy that allows duplication

```
011                 .size = 20,
012                 .buffer = {11, 12, 13, 14, 15, 16, 17, 18, 19, 11,
013                            21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
014             };
015
016             /* Start the trial session with the TPM. */
017             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
018                                       ESYS_TR_NONE, ESYS_TR_NONE,
019                                       ESYS_TR_NONE, &nonceCallerTrial,
020                                       TPM2_SE_TRIAL, &symmetricTrial,
021                                       TPM2_ALG_SHA1, &sessionTrial);
022
023             /* Bind policy to the authorization value of the authorized
024              * entity.
025              */
026             r = Esys_PolicyAuthValue(ctx,
027                                      sessionTrial,
028                                      ESYS_TR_NONE,
029                                      ESYS_TR_NONE,
030                                      ESYS_TR_NONE
031                                      );
032
033             /* Limit the authorization to a specific command code. In this
034              * case, only to duplicate the key.
035              */
036             r = Esys_PolicyCommandCode(ctx,
037                                        sessionTrial,
038                                        ESYS_TR_NONE,
039                                        ESYS_TR_NONE,
040                                        ESYS_TR_NONE,
041                                        TPM2_CC_Duplicate
042                                        );
043
044             /* Get the current policy digest of the trial session. The
045              * digest is used as authentication value for the later created
046              * key.
047              */
048             TPM2B_DIGEST *policyDigestTrial;
049             r = Esys_PolicyGetDigest(ctx,
050                                      sessionTrial,
051                                      ESYS_TR_NONE,
052                                      ESYS_TR_NONE,
053                                      ESYS_TR_NONE,
054                                      &policyDigestTrial
055                                      );
```

In the next step, the primary key and duplicate key are created as described in Sections 6.3.2.2.1 and, e.g., 6.3.2.2.2. As depicted in Code Listing 117, the public key created with OpenSSL is read (l.002ff) and copied in the public area of a TPM public parameter template (l.073ff). Afterwards, it is loaded into the TPM via ESAPI `LoadExternal` command (l.081ff).

Code Listing 117 Load a public RSA key generated externally with OpenSSL

```
001             /* Try to open key file */
002             file = fopen("key_pub.tpm2", "rb");
003
004             /* Get file length */
005             fseek(file, 0, SEEK_END);
```

Code Listing 117 Load a public RSA key generated externally with OpenSSL

```
006             fileLen=ftell(file);
007             fseek(file, 0, SEEK_SET);
008
009             /* Allocate memory */
010             buffer=(char *)malloc(fileLen+1);
011
012             /* Read file contents into buffer */
013             fread(buffer, fileLen, 1, file);
014             fclose(file);
015             /* Initialize data structure for key's authentication
016              * value.
017              */
018             TPM2B_AUTH authKeyRSA = {
019                 .size = 0,
020                 .buffer = {}
021             };
022
023             /* Initialize data structure for key's insensitive parameters.*/
024             TPM2B_SENSITIVE inSensitiveRSA = {
025                 .size = 0,
026                 .sensitiveArea = {
027                     .sensitiveType = TPM2_ALG_RSA,
028                 },
029             };
030
031             /* Set previously defined authentication value.*/
032             inSensitiveRSA.sensitiveArea.authValue = authKeyRSA;
033
034             /* Initialize template of data structure for key's public
035              * parameters.
036              */
037             TPM2B_PUBLIC inPublicRSA = {
038                 .size = 2048 + 0x1a,
039                 .publicArea = {
040                     .type = TPM2_ALG_RSA,
041                     .nameAlg = TPM2_ALG_SHA1,
042                     .objectAttributes = (TPMA_OBJECT_USERWITHAUTH |
043                                          TPMA_OBJECT_RESTRICTED |
044                                          TPMA_OBJECT_DECRYPT |
045                                          TPMA_OBJECT_SENSITIVEDATAORIGIN),
046                     .authPolicy = {
047                          .size = 0,
048                          .buffer = 0,
049                      },
050                     .parameters.rsaDetail = {
051                          .symmetric = {
052                              .algorithm = TPM2_ALG_AES,
053                              .keyBits.aes = 128,
054                              .mode.aes = TPM2_ALG_CFB,
055                          },
056                          .scheme = {
057                               .scheme = TPM2_ALG_NULL,
058                               .details.oaep.hashAlg = TPM2_ALG_SHA1,
059                           },
060                          .keyBits = 2048,
061                          .exponent = 65537,
062                      },
063                     .unique.rsa = {
064                          .size = fileLen,
```

Code Listing 117 Load a public RSA key generated externally with OpenSSL

```
065                             .buffer = *buffer,
066                         }
067                     }
068             };
069
070             /* Write the file contents (= public part of OpenSSL key) into
071              * the template data structure.
072              */
073             memcpy(inPublicRSA.publicArea.unique.rsa.buffer,
074                 (const BYTE *)buffer,fileLen);
075
076             free(buffer);
077
078             /* Execute ESAPI command to load an object that is not a
079              * TPM Protected Object, thus externally created, into the TPM.
080              */
081             r = Esys_LoadExternal(ctx,
082                                   ESYS_TR_NONE,
083                                   ESYS_TR_NONE,
084                                   ESYS_TR_NONE,
085                                   NULL,
086                                   &inPublicRSA,
087                                   ESYS_TR_RH_OWNER,
088                                   &loadedRSAKeyHandle);
```

After the external OpenSSL key is loaded, a new policy session is started to authorize key duplication (Code Listing 118). Therefore, the policy session is started first (l.017ff) and the respective policy commands are executed to authorize key usage. Afterwards key duplication is done via ESAPI command `Duplicate` with the duplicate and the externally loaded OpenSSL key handles as parameters (l.063ff).

Code Listing 118 Start a new policy session to authorize key duplication

```
001             /* Algorithm and key size for parameter encryption of the
002              * policy session.
003              */
004             TPMT_SYM_DEF policySymmetric = {.algorithm = TPM2_ALG_AES,
005                                             .keyBits = {.aes = 128},
006                                             .mode = {.aes = TPM2_ALG_CFB}
007             };
008
009             /* Set caller nonce for the policy session. */
010             TPM2B_NONCE policyNonceCaller = {
011                 .size = 20,
012                 .buffer = {11, 12, 13, 14, 15, 16, 17, 18, 19, 11,
013                            21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
014             };
015
016             /* Start the policy session with the TPM. */
017             r = Esys_StartAuthSession(ctx, ESYS_TR_NONE, ESYS_TR_NONE,
018                                       ESYS_TR_NONE, ESYS_TR_NONE,
019                                       ESYS_TR_NONE, &policyNonceCaller,
020                                       TPM2_SE_POLICY, &policySymmetric,
021                                       TPM2_ALG_SHA1, &policySession);
022
023             /* Update TPM's internal policy digest with the
024              * PolicyAuthValue policy.
025              */
026             r = Esys_PolicyAuthValue(ctx,
```

Code Listing 118 Start a new policy session to authorize key duplication

```
027                                     policySession,
028                                     ESYS_TR_NONE,
029                                     ESYS_TR_NONE,
030                                     ESYS_TR_NONE
031                                     );
032
033             /* Update TPM's internal policy digest with the
034              * PolicyCommandCode policy.
035              */
036             r = Esys_PolicyCommandCode(ctx,
037                                        policySession,
038                                        ESYS_TR_NONE,
039                                        ESYS_TR_NONE,
040                                        ESYS_TR_NONE,
041                                        TPM2_CC_Duplicate
042                                        );
043
044             /* Define variables for the command's responses. Will contain
045              * the encrypted duplicate key.
046              */
047             TPM2B_DATA encryptionKey = {
048                 .size = 0,
049                 .buffer = { }
050             };
051
052             /* Definition for the symmetric algorithm to be used for the
053              * inner wrapper.
054              */
055             TPMT_SYM_DEF_OBJECT symmetric = {.algorithm = TPM2_ALG_AES,
056                                              .keyBits = {.aes = 128},
057                                              .mode = {.aes = TPM2_ALG_CFB}};
058
059             /* Execute ESAPI command to duplicate the loaded object.*/
060             TPM2B_DATA *encryptionKeyOut;
061             TPM2B_PRIVATE *duplicate;
062             TPM2B_ENCRYPTED_SECRET *outSymSeed;
063             r = Esys_Duplicate(
064                 ctx,
065                 loadedDupKeyHandle,
066                 loadedRSAKeyHandle,
067                 policySession,
068                 ESYS_TR_NONE,
069                 ESYS_TR_NONE,
070                 &encryptionKey,
071                 &symmetric,
072                 &encryptionKeyOut,
073                 &duplicate,
074                 &outSymSeed);
```

6.3.2.8 Quote (executeQuote.c)

Note: *ESAPI supports to the generation of RSA and ECC keys. However, this programming example uses an RSA key for legacy reasons. It is recommended to switch to ECC for key creation in future developments. An example of an ECC key creation is provided in the example in section 6.3.2.3.*

In this example, the ESAPI `Quote` command is executed by the TPM. This is typically done when a remote party wants to verify the platform's integrity. Among other things, the quote consists of a signature of PCR values defined by the remote party. After the signature has been verified by the remote party, the received PCR values need to be compared to a trustworthy reference value.

To create a signature key the following steps are carried out.

1-3. If not done yet, a primary key is created, which will be used as a parent to the to-be-created restricted signature key (not shown in Listings). This can be achieved as described in Section 6.3.2.2.1.

4-5. If not done yet, a restricted signature key is created (not shown in Listings). This can be achieved as described in Section 6.3.2.2.4.

6. Execute the ESAPI command to quote PCR values and verify the created signature afterwards (Code Listing 119).

After the primary and restricted signature key are created, Code Listing 119 shows the execution and verification of the ESAPI `Quote` command. First, PCR values are selected that should be signed (l.006ff). Afterwards, the ESAPI `Quote` is executed (l.022ff). To verify the quote's signature, the attested structure needs to be converted (l.036ff) and hashed (l.049ff). In the last step, the quote's signature is verified with the hashed value (l.061ff). Typically, a remote party would verify the quote and afterwards would compare the received PCR values to a trustworthy reference value.

Code Listing 119 Execute ESAPI command to quote PCR values and verify the created signature afterwards

```
075             /* Additional input data for the quote.*/
076             TPM2B_DATA qualifyingData  = { .size = 0, .buffer = {}};
077             TPMT_SIG_SCHEME sig_scheme = { .scheme = TPM2_ALG_NULL };
078
079             /* Selected PCR values to be signed.*/
080             TPML_PCR_SELECTION pcr_selection = {
081                 .count = 2,
082                 .pcrSelections = {
083                     {
084                         .hash = TPM2_ALG_SHA1,
085                         .sizeofSelect = 3,
086                         .pcrSelect = { 0,4,0 } },
087                     {
088                         .hash = TPM2_ALG_SHA256,
089                         .sizeofSelect = 3,
090                         .pcrSelect = { 0,4,0 } },
091             }};
092
093             /* Execute ESAPI command to quote selected PCR values.*/
094             TPM2B_ATTEST *attest;
095             TPMT_SIGNATURE *signature;
096             r = Esys_Quote(ctx,
097                 loadedKeyHandle,
098                 ESYS_TR_PASSWORD,
099                 ESYS_TR_NONE,
100                 ESYS_TR_NONE,
101                 &qualifyingData,
102                 &sig_scheme,
103                 &pcr_selection,
104                 &attest, &signature);
105
106             /* Convert quoted data into TPM data structure.*/
107             TPMS_ATTEST attest_struct;
108             size_t offset = 0;
```

Code Listing 119 Execute ESAPI command to quote PCR values and verify the created signature afterwards

```
109
110             r = Tss2_MU_TPMS_ATTEST_Unmarshal(
111               (const uint8_t *)&attest->attestationData,
112               attest->size, &offset, &attest_struct);
113
114             TPM2B_MAX_BUFFER buffer = {
115             .size = attest->size
116             };
117             memcpy(buffer.buffer,(const uint8_t *)&attest->attestationData,
118               attest->size);
119
120             /* Execute ESAPI command to hash quoted data.*/
121             TPM2B_DIGEST *attestToDigest;
122             TPMT_TK_HASHCHECK *hash_validation;
123             r = Esys_Hash(ctx,
124               ESYS_TR_NONE,
125               ESYS_TR_NONE,
126               ESYS_TR_NONE,
127               &buffer,
128               TPM2_ALG_SHA1,
129               ESYS_TR_RH_OWNER,
130               &attestToDigest,
131               &hash_validation);
132
133             /* Execute ESAPI command to verify signature of quoted data.*/
134             TPMT_TK_VERIFIED *validation;
135             r = Esys_VerifySignature(ctx,
136                 loadedKeyHandle,
137                 ESYS_TR_NONE,
138                 ESYS_TR_NONE,
139                 ESYS_TR_NONE,
140                 attestToDigest,
141                 signature,
142                 &validation);
143
144         /**
145          * Integrity and authenticity of received values would be now
146          * verified. Now it has to be decided if platform is trustworthy,
147          * e.g., by comparing received PCRs values with own predefined
148          * values.
149          */
```

7 References

[1] https://www.infineon.com/cms/de/product/evaluation-boards/iridium9670-tpm2.0-linux/

[2] https://www.infineon.com/dgdl/Infineon-TPM20_Embedded_SLB_9670_AppNote-AN-v01_00-EN.pdf?fileId=5546d46265257de8016537f329595e5c

[3] https://github.com/tpm2-software/tpm2-tools/tree/master/man

[4] https://trustedcomputinggroup.org/resource/tpm-library-specification/

[5] http://www.infineon.com/tpm

[6] https://trustedcomputinggroup.org/resource/pc-client-specific-platform-firmware-profile-specification/

[7] https://trustedcomputinggroup.org/resource/tcg-tpm-v2-0-provisioning-guidance/

8 Appendix

8.1 TPM Simulator

As depicted in Figure 5 on page 14, the software simulator can be used instead of a discreet TPM with the TPM hardware drivers for development, debugging or evaluation purposes. In this case only the TPM driver need to be adapted, so that all upper software layers beginning from TPM2 Access Broker & Resource Manager can be used unchanged with the TPM simulator. Therefore, it is necessary to install the TPM2 TSS (Section 4.2) and TPM2 Access Broker & Resource Manager (TPM2 Access Broker & Resource Manager (tpm2-tabrmd) 4.3) first. However, like mentioned in the note in Section 4.3.3, enabling and starting the `tpm2-abrmd.service` should be omitted. It will be started manually with a parameter indicating to use the simulator (`--tcti=mssim`).

After installing the TPM2 TSS and TPM2 Access Broker & Resource Manager, the simulator can be downloaded and installed as depicted in Code Listing 120.

Note: The necessary dependencies for building the simulator are installed during installation of TPM2 TSS. However, if it is required to install the simulator before installing the other software, executing commands in Code Listing 2 and Code Listing 3 before continuing with Code Listing 120 is necessary.

Code Listing 120 Install and start the TPM simulator

```
001         cd /home/pi/tssSetup/
002         mkdir ibmtpm && cd ibmtpm
003         wget --no-check-certificate \
   https://sourceforge.net/projects/ibmswtpm2/files/ibmtpm1563.tar.gz
004         tar -zxvf ibmtpm1563.tar.gz
005         cd src
006         make
007         ./tpm_server
```

When the simulator is running, a currently running `tpm2-abrmd.service` might need to be stopped before starting it again manually with the simulator as interface (Code Listing 121).

Code Listing 121 Start the Access Broker & Resource Manager with simulator as interface

```
001         sudo systemctl stop tpm2-abrmd.service
002         sudo -u tss /usr/local/sbin/tpm2-abrmd --tcti=mssim
```

8.2 TCTI Loader

When linking against the ESAPI library, the TCTI Loader can be used to select a TCTI from within the code. This is useful to choose if either the TPM device or the simulator is used without explicitly using the TABRM. An example utilizing the TCTI Loader is provided with this Application Note (createPrimaryKey_TCTILDR.c). In order to compile it, the TCTI Loader library must be linked with the program. The command in Code Listing 122 can be used. The variable `OWN_PROGRAM` needs to be replaced with the name of the `.c file` that should be compiled.

Code Listing 122 Command to compile programs that use the TSS ESAPI library and the TCTI Loader

```
001         gcc OWN_PROGRAM.c -ltss2-esys -ltss2-tctildr -o OWN_PROGRAM
```

Code Listing 123 shows how to include the TCTI Loader in the code. The TCTI Loader header needs to be included (l.001) and the TCTI need to be initialized (l.006). A TCTI string can be passed to the initialization command that specifies the TCTI. Possible values are "`device:/dev/tpm0`" for the TPM device or "`mssim:host=127.0.0.1,port=2321`" for the simulator. The returned TCTI context, can be directly passed to the ESAPI initialization command (l.014). Afterwards, the TPM is initialized (l.021).

Note: When directly using the TPM device, access permissions for the device may need to be changed (e.g., `chmod a+w,a+r /dev/tpm0`) and the TABRM must be stopped (c.f., Code Listing 121, l.001).

Code Listing 123 Initialize ESAPI programming examples with TCTI Loader

```
001         #include <tss2/tss2_tctildr.h>
002
003         /* Initialize the TCTI context.*/
004         char *name = "mssim:host=127.0.0.1,port=2321";
005         TSS2_TCTI_CONTEXT *tcti_ctx = NULL;
006         r = Tss2_TctiLdr_Initialize (name, &tcti_ctx);
007         if (r != TSS2_RC_SUCCESS){
008             printf("Error: Tss2_TctiLdr_Initialize\n");
009             goto error;
010         }
011
012         /* Initialize the ESAPI context.*/
013         ESYS_CONTEXT *ctx;
014         r = Esys_Initialize(&ctx, tcti_ctx, NULL);
015         if (r != TSS2_RC_SUCCESS) {
016             printf("Error: Esys_Initialize\n");
017             return 1;
018         }
019
020         /* Startup TPM */
021         r = Esys_Startup(ctx, TPM2_SU_CLEAR);
022         if (r != TSS2_RC_SUCCESS && r != TPM2_RC_INITIALIZE) {
023             printf("Error: Esys_Startup\n");
024             return 1;
025         }
```

TCTI Loader handling is concluded by calling `Tss2_TctiLdr_Finalize` after the ESAPI_Finalize call (Code Listing 124).

Code Listing 124 Finalize TCTI Loader

```
001         Esys_Finalize(&ctx);
002         Tss2_TctiLdr_Finalize (&tcti_ctx);
```

Revision history

Reference	Description
Revision 1.2, 2020-08-10	
	Updated for FAPI Release
Revision 1.1, 2019-08-12	
	Added Quickstarter Guide Reference
Revision 1.0, 2019-03-22	
	Initial Release

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2020-08-10

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2020 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email:
dsscustomerservice@infineon.com

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.