D5.4

Report on implementation

Project number:	779391
Project acronym:	FutureTPM
Project title:	Future Proofing the Connected World: A Quantum-Resistant Trusted Platform Module
Start date of the project:	1st January, 2018
Duration:	36 months
Programme:	H2020-DS-LEIT-2017

Deliverable type:	DEM
Deliverable reference number:	DS-06-779391 / D5.4/ 1.0
Work package contributing to the deliverable:	WP 5
Due date:	June 2020 – M30
Actual submission date:	14th of July, 2020

Responsible organisation:	IFAG
Editor:	Michael Albrecht
Dissemination level:	PU
Revision:	1.0

Abstract:	This deliverable will report the public overview on the final version of VM based QR TPM, HW based QR TPM and the SW-based QR TPM.
Keywords:	Software-TPM, Virtual-TPM, Hardware-TPM, Implementation

The project FutureTPM has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779391.

Editor

Michael Albrecht (IFAG)

Contributors (ordered according to beneficiary numbers)

Andreas Wallner (IFAG), Thomas Pöppelmann (IFAG)

Christine Wright (RHUL), Daniele Sgandurra (RHUL), Fernando Sanchez Ortega (RHUL), Harry Lockyer (RHUL)

Paulo Sérgio Alves Martins (INESC-ID), Luís Fiolhais (INESC-ID), Rogério Paludo (INESC-ID)

Internal Reviewers

Sofianna Menesidou (UBITECH), Dimitris Papamartz (UBITECH)

Li Chen (SURREY), Athanasios Giannetsos (DTU)

Disclaimer

Executive Summary

The overall goal of FutureTPM is to design a Quantum-Resistant (QR) Trusted Platform Module (TPM) by designing and developing QR algorithms suitable for inclusion in a TPM; based on the selection of various families of QR crypto primitives as described in D2.1 [17]. For each primitive, detailed attention is being paid to their implementation details and performance evaluation towards their validation in the full range of TPM environments, namely hardware (hTPM), software (sTPM) and virtualization (vTPM) environments. Accompanied with the QR algorithm design and implementation, the FutureTPM project will demonstrate three use cases in secure mobile wallets and payments, activity tracking and device management, which will provide environments and applications to validate the feasibility and performance of the newly developed QR TPM in these three selected real-world systems that may be affected by the advent of quantum computing as a threat to security.

In this context, this deliverable reports the design and implementation of the quantum-resistant (QR) HW-TPM (Task 5.3), gives a consolidated overview on the final implementation of the QR SW-TPM (Task 5.1) and describes the current status of the final QR virtual-TPM implementation (Task 5.2), scheduled for M30.

Open Issues, documented in D5.1 and D5.2 are resolved, NTTRU is implemented as well as L-DAA on the promised level of optimization. The QR HW-TPM Demonstrator (D5.3) is finally available, including updated security schemes (BLISS replacing qtesla) and ready for integration- and use case testing (WP6).

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Scope and Purpose

Research on quantum computers has drawn attention from governments and industry. If, as predicted, a large-scale quantum computer becomes a reality within the next 15 years, existing public-key algorithms will be open to attack. FutureTPM is aimed at designing and developing a Quantum-Resistant (QR) Trusted Platform Module (TPM). FutureTPM’s main goal is to enable a smooth transition from current TPM environments, based on existing widely used and standardised cryptographic techniques, to systems providing enhanced security through QR cryptographic functions, including secure authentication, encryption and signing functions.

According to TPM 2.0 specifications [19], there are five different types of TPM 2.0 implementations:

- **Discrete TPMs**: they are dedicated chips that implement TPM functionality in their own tamper-resistant semiconductor package. Theoretically, they are the most secure type of TPM as, for instance, their packages are required to implement tamper resistance.
- **Integrated TPMs**: they are included as part of another chip. While they use hardware that resists software bugs, they are not required to implement tamper resistance.
- **Firmware TPMs**: these are software-only solutions that run in a CPU's trusted execution environment. Since these TPMs are entirely software solutions that run in trusted execution environments, these TPMs are more likely to be vulnerable to software bugs.
- **Software TPMs**: they are software emulators of TPMs that depend on the environment that they run in. Typically, they offer the same level of security of their execution environment, and so are vulnerable to software bugs. Therefore, they are typically used for development purposes.
- **Virtual TPMs**: they are meant to be provided by a hypervisor to allow virtual machines to share a single instance of a TPM. These TPMs rely on the hypervisor to provide them with an isolated execution environment that is hidden from the software running inside virtual machines.

TRUST ELEMENT	SECURITY LEVEL	SECURITY FEATURES	RELATIVE COST	TYPICAL APPLICATION
DISCRETE TPM	HIGHEST	TAMPER RESISTANT HARDWARE	$$$	CRITICAL SYSTEMS
INTEGRATED TPM	HIGHER	HARDWARE	$$	GATEWAYS
FIRMWARE TPM	HIGH	TEE	$	ENTERTAINMENT SYSTEMS
SOFTWARE TPM	NA	NA	¢¢	TESTING & PROTOTYPING
VIRTUAL TPM	HIGH	HYPERVISOR	¢	CLOUD ENVIRONMENT

Figure 1: Types of TPM according to TCG

FutureTPM is investigating technologies for a new generation of TPM-based solutions, including hardware (HW), software (SW) and virtualization environments, by incorporating QR cryptographic primitives. In addition, FutureTPM aims to prove and validate the applicability, usability, effectiveness and value of the QR TPM concepts, models and algorithms in real-world settings, including industry and e-commerce, which may be affected by the advent of quantum computing.

In this deliverable, we describe the current implementation status of the QR HW-TPM demonstrator; overview the concept and progress on the QR Virtual TPM; and give an update on the QR SW-TPM.

The subsequent figure gives an overview of the QR TPM environments developed within WP5. It shows that all QR TPM environments are based on the IBM SW TPM, where the QR VM-TPM and the QR SW-TPM share the same code-basis and the QR VM-TPM implements additional crypto algorithms (BIKE, SPHINS+, Rainbow), as has been described in D2.2 [10]. Furthermore, it gives an update on the algorithms that have now been implemented in D5.4 – latter are marked bold.

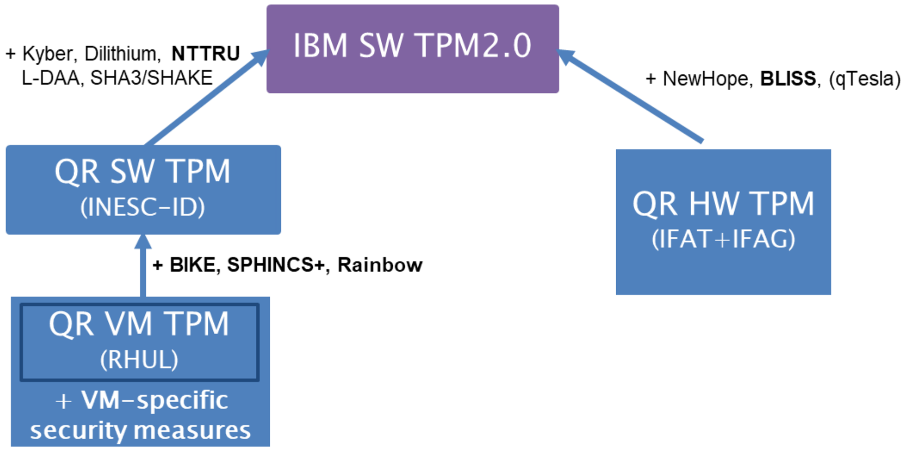

Figure 2: Overview of the QR TPM environments.

1.2 Relation to Other WPs and Deliverables

WP2 deals with the selection of which QR cryptographic primitives should be implemented in each TPM environment, both providing advice and receiving feedback from WP5. WP5 serves as the basis for the developments of the TPM environments to be demonstrated in the envisioned use cases (WP6). In particular, the QR SW-TPM will be the basis for the “Personal Activity and Health Kit Data Tracking” use case, the QR Virtual TPM will serve as basis for the “Device Management” use-case and the QR HW-TPM will be used by the “Secure Mobile Wallet and Payments” use-case.

Nevertheless it has to be stated within WP6, that the 1st round of experimentation and evaluation, was based on SW-TPM only, for all three use cases. In the current project phase, as HW TPM and vTPM become more and more available, a 2nd round of experiments will show the adaption of the use cases to their dedicated QR target platforms.

Within WP5, Task 5.1 (Implementation and Evaluation of Software QR TPM) dealt with the development of the QR SW-TPM. Task 5.2 (Implementation and Evaluation of Virtual QR TPM) and Task 5.3 (Implementation and Evaluation of Hardware QR TPM) are devoted to the design, implementation and evaluation of the QR algorithms in a VM-based TPM and a HW-TPM demonstrator, respectively. Furthermore, Task 5.4 deals with the development of a TPM Software Stack (TSS) API that covers the newly introduced QR cryptographic algorithms and, finally, in Task 5.5 a HW coprocessor to accelerate lattice operations is going to be developed.

1.3 Deliverable Structure

This deliverable is structured as follows:

- **Chapter 2** provides an update of progress on the software TPM that has been made since the submission of D5.3 in M27.
- **Chapter 3** details the concept of the second version of the quantum-resistant VM-based TPM and gives an overview of the current state of its development.
- **Chapter 4** details the design of the final version of the quantum-resistant HW-based TPM demonstrator.
- **Chapter 5** concludes the deliverable and provides a roadmap for the next steps.

The following D5.4 updates are reflected in this document:

Consolidated, public summary of TSS and TPM environments, based on D5.1-D5.3, including detailed summaries for SW- & virtual-TPM, an overview of HW-TPM and a brief description of PQC HW Acceleration

SW-TPM

- Adapted TSS to reflect PQC algorithms of QR TPM

Virtual-TPM

- Derived VM-TPM-specifics from SW-TPM implementation
- Integrated BIKE and Rainbow into QR-Libtpms
- Integrated SW-TPM/QR-Libtpms with QEMU/KVM to provide V-TPM functionalities

HW Demonstrator

- IFX to provide INDEV/UBITECH with SW simulator & HW TPM
- Use of TSS over TPC/IP as main interface for the integration
- Use HW-TPM FW compiled for PC platform for use case integration (PAY)
- Description of PQC Hardware Acceleration

Chapter 2 Update on the SW-TPM

This chapter gives an incremental update on the changes performed on the Software TPM and TPM Software Stack since the previous deliverable D5.3.

This release includes appropriate fixes for all the bugs identified during the integration and experimentation activities of WP6 [12]. It also extends the NTTRU capabilities with general data encryption and decryption. Furthermore, the previous four SW-TPM endpoints for key encapsulation and decapsulation under Kyber and NTTRU were merged into two endpoints given their functionality.

The TSS was updated accordingly to support the new endpoints for key encapsulation and decapsulation. Currently, the TSS allows the generation of encrypted salted sessions under Kyber or NTTRU. Additionally, all regression tests have been extended to support the new endpoints.

2.1 New Endpoints

In the deliverable D5.3 the NTTRU lacked functionality for general data encryption and decryption. This has been implemented and thoroughly tested in this release. Thus, following the guidelines provided in D5.1 [13] the new commands for NTTRU are:

The **TPM2_NTTRU_Encrypt** encrypts a user-provided plaintext, with the following parameters:

- **Input**: key_handle for a loaded key handle to a NTTRU public key and the user's plaintext message (the message has a maximum size of MAX_DIGEST_BUFFER).
- **Output**: a ciphertext generated by the TPM.

The **TPM2_NTTRU_Decrypt** command decrypts a ciphertext generated by TPM2_NTTRU_Encrypt, with the following parameters:

- **Input**: key_handle must reference a loaded private NTTRU key and a ciphertext generated by TPM2_NTTRU_Encrypt.
- **Output**: the TPM will decrypt the cipher object to obtain the original plaintext message.

The TSS API already supports the new NTTRU endpoint for general data encryption (nttruencrypt) and decryption (nttrudecrypt).

- nttruencrypt => a NTTRU-only command where an encryption is performed using the selected loaded key and an input is received to be encrypted. An example of its usage is:
    - ./nttruencrypt -hk LOADED_KEY_HANDLE -id IN_PLAIN -oe OUT_CIPHER
- nttrudecrypt => a NTTRU-only command where a decryption is performed using the selected loaded key and an input is received to be encrypted. Its usage is:
    - ./nttrudecrypt -hk LOADED_KEY_HANDLE -ie IN_CIPHER -od OUT_PLAIN

Additionally, new endpoints for key **encapsulation** and **decapsulation** were added to the SW-TPM and TSS replacing the previous commands **TPM2_NTTRU_Enc**, **TPM2_NTTRU_Dec**, **TPM2_Kyber_Enc**, and **TPM2_Kyber_Dec,** with the commands bellow.

The **TPM2 _Enc** command encapsulates a shared secret, with the following parameters:

- **Input**: key_handle for a loaded public key, either under Kyber or NTTRU.
- **Output**: a shared secret and a ciphertext generated by the TPM.

Respectively, **TPM2_Dec** decapsulates a shared secret, with the following parameters:

- **Input**: key_handle must reference a private loaded key and a ciphertext generated by TPM2_Enc.
- **Output**: the TPM will decapsulate the cipher object to obtain the shared_secret.

To reflect the additions of these commands to the SW-TPM the TSS API was updated by calling the appropriate methods where required and with the commands bellow for direct encapsulation and decapsulation.

- **encapsulate** => a command where an encapsulation is performed using the selected loaded key. An example of its usage is:
  - ./encapsulate –hk LOADED_KEY_HANDLE -c OUT_CIPHER -ss OUT_SHARED_SEC
- **decapsulate** => a NTTRU-only command where a decapsulation is performed using the selected loaded key and a ciphertext previously generated by the nttru_enc command. An example of its usage is:
  - ./decapsulate -hk LOADED_KEY_HANDLE -c IN_CIPHER -ss OUT_SHARED_SEC

The extension of NTTRU to support general data encryption and decryption and the new endpoint for key encapsulation and decapsulation completes all agreed upon required algorithms by the QR-TPM and advances one of the still open issued of the deliverable D5.3.

2.2 Miscellaneous SW-TPM Updates

This session presents a summary of the actions taken to fix the open issues since the last public deliverable D5.1 and also a minor fix from D5.3.

Regarding the updates since D5.1:

- Dilithium and Kyber implementation instances were both updated to their latest versions from the NIST submission (round 2) [14][15] and the changes required by the current versions are backwards-compatible with the previously described TPM stack versions in D5.1 [13].
- Two notable bugs were found during the implementation and integration in the V-TPM environment. The first related to generation of a key from a predetermined seed which it stemmed from the fact that the seed provided by the user was ignored during the key generation process. The fix entailed calling the PRNG state using the seed instead of ignoring it. The second bug was related to authenticated sessions using Kyber keys. This bug caused inconsistencies across libtpms and the SW-TPM. Fixing this bug required updating the incorrect functions. After merging theses fixes the QR-TPM stack was successfully deployed across all use cases in WP6, where its results were published in D6.3 [12].
- Support for QR-TLS was added using the TPM as a cryptographic engine. The following example demonstrates the usage of the OpenSLL engine. The command “tpm2tss-genkey” creates a key-pair which is used by the command “openssl” to generate a self-signed certificate.

  ```
  $ tpm2tss-genkey -a dilithium dilithium.tss
  $ openssl req -new -x509 -engine tpm2tss -key dilithium.tss -
  keyform engine -out dilithium.crt
  ```

  To accept TLS HTTP connections on port 8443, supported on Kyber and Dilithium, a server is instantiated with

  ```
  $ openssl s_server -cert dilithium.crt -key dilithium.tss -keyform
  engine -engine tpm2tss -accept 8443 –www
  ```

  Then a client can connect to server by executing

  ```
  $ echo "GET index.html" | openssl s_client -engine tpm2tss -connect
  localhost:8443
  ```

  Thus, a key-exchange supported on Kyber takes places, and the server authenticates itself with a Dilithium signature.

Finally, the SW-TPM was updated from D5.3 to verify the secret size before calling the general decryption methods of Kyber and NTTRU. This prevents the SW-TPM of trying to decrypt an oversized secret. A secret that does not fit into the predefined space causes the SW-TPM to crash with a stack overflow error.

2.3 Resolved and Open Issues

The following issues have been resolved since the last public release D5.1 of the SW-based TPM implementation:

- The regression test suite has been updated and there is no integrity error in the Salt tests for Kyber.
- Kyber and Dilithium were updated to the latest version submitted in NIST competition.
- There won't be a new API endpoint for SHAKE hashes. The consortium decided that it would be best to invest more time in the remaining L-DAA issues and supporting the consortium members in WP6 (this issue was originally described in D5.1).
- The modified OpenSSL broke backwards-compatibilty with other TLS ciphers when running certain regression tests. Therefore, similarly to the SHAKE endpoint, it was decided by the consortium that the failing regressions tests in the OpenSSL test suite will not be considered.

This deliverable fixed the following open issues of D5.3:

- The functionality of the NTTRU algorithm has been fully extended on the SW-TPM and TSS to support general data encryption, decryption, and salted sessions.

However, the following challenges still remain “active”:

- The L-DAA issues have not been updated. Nonetheless, the consortium is working on a new L-DAA proposal planned to overcome the issues opened in D5.1 [2]. The new implementation of L-DAA will be detailed in the release of WP2.

Chapter 3 Virtual TPM Overview

This chapter reports on the design of the first version of the Quantum-resistant VM-based Trusted Platform Module (TPM), QR Virtual-TPM or QR VM-TPM. The differences between this chapter and the corresponding chapter in D5.3 are as follows:

- Section 3.1.5 has been updated to report that:
  - We have resolved the problem with the TIS buffer size and have tested some TPM commands on the V-TPM with performance measurements taken.
  - We have successfully integrated SW-TPM/QR-Libtpms with QEMU/KVM and tested Kyber and Dilithium using the “Alice and Bob” protocol as described in the D5.1 document.
- Section 3.2.4 has been updated to report that we have successfully compiled BIKE into QR-Libtpms
- Section 3.2.5 has been updated to report that we have commenced the testing of SPHINCS+, Rainbow and BIKE but we have encountered a technical issue.
- Section 3.3 (Future Work) has been removed because the content has been integrated into Section 3.2.5.
- Appendix D has been updated to add details of integration of BIKE into QR-Libtpms.
- Appendix E has been added to report an issue with TSS library when invoking QR functionalities.

3.1 QR Virtual-TPM Architecture

3.1.1 Choice of QEMU

Several methods of VM-TPM have been proposed and developed, however, none of them entirely fits with the requirements of the FutureTPM framework. A comparison of VM-TPM feasibility has already been done in the literature [18], and an overview will be given in the remaining sections of this chapter. Three implementation activities on two systems have been proposed, with emulations on **Xen** and **QEMU** (short for Quick EMUlator) and a pass-through for Xen. [10] QEMU is the chosen emulator, and KVM is the chosen virtualization layer. In particular, QEMU pass-through makes use of the HW-TPM of the host, and strongly links a single guest to the host. Therefore, when used as a pass-through, this approach is infeasible in a cloud-based implementation where multiple guests are offered their own TPM functionality. Equally, it makes migration of guests infeasible, as the HW-TPM state cannot be migrated. Instead, both of these functionalities are possible when, instead of using the pass-through functionality, QEMU uses a TPM emulator (SW-TPM) as backend. QEMU with TPM emulation currently exists, in particular by exploiting Libtpms[1] (see next section for more information) to emulate a TPM device, along with a modified TPM Interface Specification (TIS) on the guest BIOS. Migration is possible in this situation as the TPM state is saved along with the VM image. Each guest has its own emulated TPM, and therefore states cannot be shared between VMs.

[1] https://github.com/stefanberger/libtpms

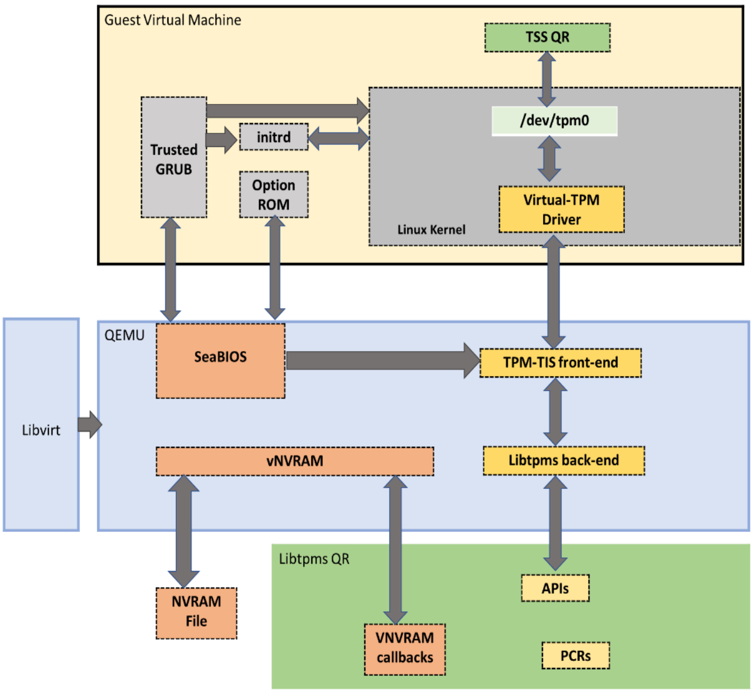

Figure 3: Architecture of V-TPM

3.1.2 Choice of QR-Libtpms

The libtpms is a library to support emulation of TPM 1.2 and TPM 2.0. It is implemented independently of a backend for storage and an interface for communication. It is thus a flexible library for the integration of TPM functionality into hypervisors, primarily into QEMU. A Libtpms with QR cryptographic functionality included (QR-Libtpms) is used in this chosen architecture. This QR-Libtpms includes implementations of Dilithium, Kyber and L-DAA. We have identified the modifications that have been made to the Libtpms for the purpose of adding new algorithms, i.e. SPHINCS+, Rainbow and BIKE.

3.1.3 Implementation of Virtual TPM

In this architecture (see Figure 3), a Virtual Machine (VM) is connected using a driver to a TPM emulator (based on QR-Libtpms) by using TPM-TIS buffer. Each VM contains a TSS QR, which is the modified version of the TSS of IBM with the new functions which have QR-algorithms added in the virtual TPM code such as Kyber and Dilithium.

The functions (`createprimarykey`, `signed`, *etc*) executed by the TSS, are captured by the Virtual-TPM driver. The Virtual-TPM driver is created by the hypervisor QEMU when we power on the VM with this command:

```
qemu-system-x86_64 -display sdl -accel kvm   -m 1024 -boot d -bios
bios-256k.bin -boot menu=on   -chardev
socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock  -tpmdev
emulator,id=tpm0,chardev=chrtpm  -device tpm-tis,tpmdev=tpm0
virtualmachine.img
```

This command creates in the VM these two components:

- `/dev/tpm0` → TPM Device Driver
- `/dev/tpmrm0` → multi-process entry point.

Please note that the TPM emulator must be started before trying to access it through QEMU. This TPM emulator implements two components:

1. Command *channel* for transferring TPM commands and responses.
2. Control *channel* over which we can use to reset, modify or initialize the TPM state, among other things.

As previously described, it is mandatory to create an instance of the TPM emulator before QEMU tries to create the Virtual-TPM driver. The next command is an example of an instance of the TPM emulator which is using *UNIX socket* for the communication:

```
mkdir /tmp/mytpm1

./swtpmsocket --tpmstate dir=/tmp/mytpm1 --tpm2 –ctrl
type=unixio,path=/tmp/mytpm1/swtpm-sock --log level=20
```

This emulator is connected to the driver virtual TPM by using TIS.

The consortium was testing the Virtual TPM architecture with QR-Libtpms code (that includes only Dilithium, Kyber and L-DAA). Initially when using the QR algorithms in the QR-Libtpms, the communication between the VM and the Libtpms did not work properly. To resolve this issue, we have modified the TIS buffer size in Kernel of the VM and QEMU. See Section 3.1.5.

3.1.4 Using SW-TPM within V-TPM

In the following, we describe how to use the SW-TPM and V-TPM.

1. **<u>In the same host</u>**

To use the IBM TSS for TPM 2.0 directly with SW-TPM in the same host, we need to use the following commands:

Start SW-TPM in one terminal:

```
mkdir /tmp/myvtpm

swtpm socket --tpmstate dir=/tmp/myvtpm --tpm2 --ctrl type=tcp,port=2322
\

   --server type=tcp,port=2321 --flags not-need-init
```

Execute indicative operations with the QR-TSS stack in another terminal:

```
export TPM_COMMAND_PORT=2321 TPM_PLATFORM_PORT=2322 \

  TPM_SERVER_NAME=localhost TPM_INTERFACE_TYPE=socsim \

  TPM_SERVER_TYPE=raw

tssstartup

tsspcrread -ha 10

count 1 pcrUpdateCounter 21
```

```
digest length 32
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

To reset SW-TPM run the following command:

```
swtpm_ioctl -i --tcp :2322
```

2. **Using a TSS in a VM and V-TPM**

Start SW-TPM in one terminal in the host machine:

```
mkdir /tmp/mytpm1
./swtpmsocket --tpmstate dir=/tmp/mytpm1 --tpm2 –ctrl
type=unixio,path=/tmp/mytpm1/swtpm-sock --log level=20
```

Each VM has installed a TSS QR which is a modified version of the TSS of IBM with new functions with the aim of checking the QR algorithms added in the virtual TPM such as Kyber and Dilithium.

The functions are executed by the QR-TSS are captured by the Virtual-TPM driver, which is created by the hypervisor QEMU when we power on the VM with this command:

```
qemu-system-x86_64 -display sdl -accel kvm -m 1024 -boot d -bios bios-
256.bin -boot menu=on -chardev
socket, id=chrtpm, path=/tmp/mytpm1/swtpm-sock -tpmdev
emulator, id=tpm0, chardev=chrtpm -device tpm-tis, tpmdev=tpm0
virtualmachine.img
```

The consortium has created a Dockerfile to be used to test in local the SW-TPM and QR-Libtpms and QR-TSS. The technical steps to create a virtual environment using a Dockerfile are available in a separate whitepaper.

3.1.5 QR V-TPM

There were issues with the TIS buffer size which had to be edited to allow larger cryptographic keys to be allocated correctly. This was amended by editing the following file:

```
./qemu/hw/tpm/ tmp_tis.h
```

where `TPM_TIS_BUFFER_MAX` is the variable which needs to be changed.

The buffer size had to be changed to 200 KB in order to accommodate the largest cryptographic keys being used by the consortium. In making the file, the flag --ignore-errors needs to be included to allow the file to be compiled, otherwise an `unsigned int` errors can occur. It should be noted that the latest version of GCC needs to be installed on the device before QEMU is made[2].

After successful compilation, the modified QEMU was created and the other steps were made to boot the QEMU image with the V-TPM running, commands can be issued to the V-TPM, some issues with performance were noted as `tss startup` command took slightly longer as previous with 0.003817601 seconds time elapsed to initiate the command. The increased buffer size should address the previous issue with regards to the keys being larger than the allocated buffer size. Key generation commands worked with increased key size with no major impact on the performance.

Some TPM commands where tested on the V-TPM to see the functionality and the how well they worked the tests where benchmarked with the following *perf.* The results are shown in Table 1..

Command	Purpose	Time Elapsed
tssstartup	Initiates startup to the V-TPM	0.005564852 seconds
tssstartauthsession	Starts session with the V-TPM	0.004963542 seconds
tsshierarchychangeauth	Prevents rouge users changing policy of the V-TPM	0.004920123 seconds
tssdictionaryattackparameters	Simulates a dictionary-based attack against the V-TPM	0.004927550 seconds
tssgetcapability	Used top display what the TMP can do such as display what algorithms it can handle and use.	0.003924411 seconds

Table 1: V-TPM Performance Results

The V-TPM has been integrated with the QR-SWTPM with the use of a Docker container by allowing KVM to run inside it for the V-TPM to correctly work. Running the V-TPM within Docker requires some extra libraries to enable KVM to run within the container so the V-TPM can run in the hypervisor[3]. Table 2: Testing and Timings of the V-TPM within the Docker Container shows the timing results of running the commands in the container. These commands are based on the Alice and Bob protocol described in the D5.1 document.

Command	Timing
`/.kyberencrypt -hk 80000001 -id test.txt -oe enc.bin`	real 0m0.257s user 0m0.142s

[2] It can be noted during compilation, some (unrelated) errors may still occur, and you may be asked to submit a bug report, but this does not affect the V-TPM.
[3] See: https://blog.scottlowe.org/2017/11/24/using-docker-machine-kvm-libvirt/

	sys 0m0.107s
./sign -hk 80000001 -dilithium -if enc.bin -os sig.bin -pwdk dilithium	real 0m0.265s user 0m0.136s sys 0m0.120s
./load -hp 80000000 -ipr kyber_priv.bin -ipu kyber_pub.bin -pwdp sto	real 0m0.257s user 0m0.139s sys 0m0.111s

Table 2: Testing and Timings of the V-TPM within the Docker Container

Figure 4 and Figure 5 show the output and the timing of two QR commands issued within a VM.

```
[root@f18059be96a3 utils]# time ./kyberencrypt -hk 80000001 -id test.txt -oe enc.bin

real    0m0.258s
user    0m0.135s
sys     0m0.115s
```

Figure 4: Output of Kyber Encrypting a File in a VM

```
sys     0m0.107s
[root@f18059be96a3 utils]# time ./sign -hk 80000001 -dilithium -if enc.bin -os sig.bin -pwdk dilithium
in.inScheme.scheme = 45

real    0m0.265s
user    0m0.136s
sys     0m0.120s
```

Figure 5: Output of Signing Operation on a File in a VM

We have found an issue with TSS library when invoking QR functionalities. See Appendix E for more details. This issue is of non-deterministic nature, so it affects the functionalities only during some of the runs, while overall the V-TPM functionalities are fine.

3.2 QR Cryptographic Algorithms in V-TPM

The consortium has followed the steps described in the D5.1 document on adding new algorithms.

The integration plan is aimed at adding the following QR cryptographic algorithms into QR-Libtpms:

- SPHINCS+
- Rainbow
- BIKE

3.2.1 SPHINCS+

SPHINCS+ is a stateless variation of the stateful XMSS hash-based signature scheme. The distinguishing characteristic of this signature scheme is that its security is based only on the hardness of symmetric primitives. The signature size is around 30KB, which is acceptable in many scenarios. The main downside of the scheme is that it is slow and resource-intensive. In applications where speed and, to a lesser extent, signature size are not very important and there is a general consensus within the cryptographic community that it will be standardized by NIST [21].

3.2.1.1 SPHINCS+ Chosen Parameter Set

The consortium has examined six parameter sets put forward by the SPHINCS+ working group. For a security level, there are tradeoffs between signature size (s) and speed (f). These are shown in Table 3.[1]

SPHINCS+	n	h	d	log(t)	k	w	bitsec	Sec level	Sig bytes
-128s	16	64	8	15	10	16	133	1	8 080
-128f	16	60	20	9	30	16	128	1	16 976
-192s	24	64	8	16	14	16	196	3	17 064
-192f	24	66	22	8	33	16	194	3	35 664
-256s	32	64	8	14	22	16	255	5	29 792
-256f	32	68	17	10	30	16	254	5	49 216

Table 3: SPHINCS+ Parameter Sets

For each sec level, one size-optimised (ending on 's' for "small") and one speed-optimised (ending on 'f' for "fast") parameter set.

The consortium has chosen to use the "-128f" parameter set which has 128 bitsec and security level 1 (second row from the top).

The SPHINCS+ "-128f" has the following parameter set:

n : the security parameter = 16
h : the height of the hypertree = 60
d : the number of layers in the hypertree = 20
t : the number of leaves of a FORS tree
log(t) = 9
k : the number of trees in FORS = 30
w : the Winternitz parameter = 16
Bits per second = 128
Signature in bytes = 16976

It makes sense to choose security level 1 because NIST has 5 security categories for security evaluation of PQC algorithms, and NIST recommends focusing on categories 1, 2 and 3 (see Table 4).[17]

NIST Category	Description
1	Attack with similar complexity of those required for breaking AES-128
2	Attack with similar complexity of those required for collision search SHA256/SHA3-256
3	Attack with similar complexity of those required for breaking AES-192
4	Attack with similar complexity of those required for collision search SHA384/SHA3-384
5	Attack with similar complexity of those required for breaking AES-256

Table 4: NIST Categories

In other words, SPHINCS+ implementation has:

- Public Key Size = 32 Bytes
- Secret Key Size = 64 Bytes
- Signature Size = 16976 Bytes
- Signed Message Size = Signature Size + Message Size

To implement SPHINCS+, the consortium has used the standard codebase from the SPHINCS+ site: https://sphincs.org/ The SPHINCS+ code was unit tested. The unit test results demonstrated that SPHINCS+ key generation, signature generation and verification are working correctly which provided the assurance that the SPHINCS+ code is fit for purpose. To implement SPHINCS+ in SW-TPM and TSS, the consortium has made the relevant code changes. For more details, see Appendix A.

3.2.2 *Rainbow*

Rainbow is a multivariate digital signature scheme. It is a generalization of the Unbalanced Oil and Vinegar (UOV) structure. This design allows parameterizations that are more efficient at the cost of additional algebraic structure. The Rainbow signature scheme was analyzed to be EUF-CMA secure utilizing a hash construction with a random salt.

Since the original Rainbow signature scheme was published in 2005, the scheme has been studied with various parameters. NIST commented that the spectrum of Rainbow parameters allows for optimization in a diverse array of use cases. In the NIST second round submission, the key generation algorithm for the original Rainbow scheme was improved, the nine parameter sets in the first round submission was narrowed down to three sets. Two variants of Rainbow signatures were proposed in order to make a trade-off in key size and performance.

It is also commented by NIST that a further benefit of Rainbow is that it has also been studied in other contexts, including in lightweight applications. Overall, Rainbow has the advantages of being simple and easy to implement and very fast, but it has the limitation of having large key sizes [22].

3.2.2.1 Rainbow Chosen Parameter Set

In the NIST first round submission, there were nine parameter sets. In the NIST second round submission, the key generation algorithm for the original Rainbow scheme was improved, the nine parameter sets in the first round submission was narrowed down to three sets. These are shown in Table 5.[22]

NIST Security Category	Standard Rainbow		Cyclic Rainbow		Compressed Rainbow	
	| pk | KB	| sk | KB	| pk | KB	| sk | KB	| pk | KB	| sk |
I/II	149.0	93.0	58.1	93.0	58.1	64B
III/IV	710.6	511.4	206.7	511.4	206.7	64B
V/VI	1,705.5	1,227.1	491.9	1,227.1	491.9	64B

Table 5: Rainbow Parameter Sets

Rainbow Signature sizes are: 48B, 140B, 184B [22]

The consortium has chosen to implement the Standard (or Classic) Rainbow parameter set for NIST Security Category I/II in order to be consistent with SPHINCS+ "-128f" parameter set which has security level 1.

In other words, Rainbow implementation has:

- Public Key Size = 149000 Bytes
- Secret Key Size = 93000 Bytes
- Signature Size = 184 Bytes (Maximum)
- Signed Message Size = Signature Size + Message Size

To implement Rainbow, the consortium has used the Rainbow codebase from the NIST site: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions. The Rainbow code was unit tested. The unit test results demonstrated that Rainbow key generation, signature generation and verification are working correctly which provided the assurance that the Rainbow code is fit for purpose. To implement Rainbow in SW-TPM and TSS, the consortium has made the relevant code changes. For more details, see Appendix B.

3.2.3 BIKE

3.2.3.1 BIKE Chosen Parameter Set

BIKE (Bit Flipping Key Encapsulation) is a suite of algorithms for key encapsulation based on quasi-cyclic moderate density parity-check (QC-MDPC) codes that can be decoded using bit flipping decoding techniques.[9] Key Encapsulation mechanism (KEM) is composed of 3 algorithms: [23]

- GEN – outputs a public encapsulation key pk and a private decapsulation key sk,
- ENCAPS – takes as input an encapsulation key pk and outputs a ciphertext c and a symmetric key K, and
- DECAPS – takes as input a decapsulation key sk and a cryptogram c and outputs a symmetric key K or a decapsulation symbol (unless using implicit rejection).

The BIKE suite consists of 3 variants: BIKE-1, BIKE-2 and BIKE-3. Each variant offers different performance trade-offs [23].

- BIKE-1 has fast, inversion-less key generation and larger public keys (2 blocks).
- BIKE-2's key generation is more expensive but public keys are smaller. BIKE-3's decapsulation invokes the decoding algorithm on a "noisy" syndrome.
- BIKE-3 is fundamentally distinct from BIKE-1 and BIKE-2, mainly in terms of security and security-related aspects like choice of parameters.

To be consistent with SPHINCS+ and Rainbow, the consortium has chosen to implement BIKE-1 at NIST Security Level 1. The BIKE-1 implementation has: [23]

- Public Key Size = 2541 Bytes
- Secret Key Size = 249 Bytes
- Cipher Text Size = 2541 Bytes (Maximum)

To implement BIKE, the consortium has used the standard codebase from the BIKE site: https://bikesuite.org/ The BIKE code was unit tested. The unit test results demonstrated that BIKE key generation, encapsulation and decapsulation are working correctly which provided the assurance that the BIKE code is fit for purpose. For more details, see Appendix C.

3.2.4 Integration of SPHINCS+, Rainbow and BIKE into QR-Libtpms

To integrate SPHINCS+, Rainbow and BIKE into QR-Libtpms, the SPHINCS+, Rainbow and BIKE source files, the modified SW-TPM and TSS files were copied into the relevant QR-Libtpms folders and successfully compiled. For more details, see Appendix D.

The newly added algorithms (SPHINCS+, Rainbow and BIKE) have been successfully integrated into the SW-TPM with a Docker container with reference to D5.1 section 3.2.2.

3.2.5 Open Issues

We have started testing the integrated functionalities offered by SPHINCS+, Rainbow and BIKE, successfully integrated into SW-TPM/QR-Libtpms (see Section 3.2) in the context of the V-TPM similarly as above (using the "Alice and Bob" protocol). Because TPM is designed to be "algorithm agile", the SW-TPM and TSS code have been changed to point to run the new algorithms and no changes were made to network commands such as "swtpm".

When configuring the SW-TPM, the “swtpm” command is needed for the user to send commands to the SW-TPM (see Figure 6).

Without this command, no commands can be sent via tss2 and the newly added algorithms cannot be tested. We found an issue when setting the flags “not-need-init”. The issue found is a “SetBit” memory issue which has become present when running the SW-TPM, once the command has a terminal response with an error message:

Entering failure mode, mode 4 location SetBit line 98 (see Figure 6)

This was investigated and there were no changes to this from the Kyber integration to the newly added algorithms. A post was made on the Github repository with regards to this issue and its ticket is still waiting for a response.

```
[root@3c3d4a8a85cd ~]# swtpm socket --tpmstate dir=/tmp/myvtpm --tpm2 --ctrl type=tcp,port=2322,bindaddr=172.17.0.2 -
-server type=tcp,port=2321,bindaddr=172.17.0.2 --flags not-need-init
libtpms/tpm2: Entering failure mode; code: 4, location: SetBit line 98
Segmentation fault (core dumped)
[root@3c3d4a8a85cd ~]#
```

Figure 6: "swtpm socket" command

Chapter 4 HW-TPM Demonstrator Overview

This chapter reports on the design of the quantum-resistant HW TPM (QR HW-TPM).

4.1 QR HW-TPM Demonstrator

The herein described HW-TPM demonstrator is based on a fork of the open-source implementation of the SW-TPM 2.0 produced by IBM and Microsoft running bare-metal on a Cortex-M3 CPU synthesized for an FPGA evaluation platform. It comes with added support for new QR algorithms, namely the key encapsulation mechanism (KEM) NewHope [1] and the digital signature scheme (DSS) BLISS [4].

The following discusses the architecture of the HW-TPM demonstrator, the implementation of a minimal OS environment necessary to run the modified QR SW-TPM on the FPGA, the implementation of the QR algorithms BLISS and NewHope on the HW-TPM demonstrator. Moreover, an outline for concepts on hardware acceleration of cryptographic primitives is provided.

The main reasons for the chosen architecture are:

- the ability to integrate the accelerator that will be built as part of D5.5
- the advantage to not be bound by the limitations posed by existing platforms, especially with regards to available memory
- the analysis and testing possibilities for all details of the platform. This would allow for more detailed lessons learnt and guidelines on the considerations for possible integrated HW-based TPM implementations as will be generated during WP6.

4.2 FPGA Evaluation Platform

The ARM DesignStart Eval platform is used for the evaluation of the QR HW-TPM. It is optimized for the DesignStart MPS2+ board and well-suited for rapid prototyping. The MPS2+ board is supplied with fixed encrypted FPGA implementations of all the Cortex-M processors integrated into a prototype System-on-Chip (SoC). In addition, DesignStarts includes the encrypted netlist of an ARM Cortex-M3 processor with modifiable source code for a SoC with useful peripherals like PSRAM, Ethernet, Audio, VGA, SPI and GPIO. A simplified block diagram of the system is provided in Figure 7. The Cortex-M3 is closely related to the ARM Secure Core (SC) SC300, which also implements the ARMv7 instruction set and additional security functions. The SC300 is used in commercially available security microcontrollers. Usage of the prototyping SoC thus allows deriving general statements about the resource requirements of the TPM software and implementations of cryptographic software (e.g. memory footprint, execution time, etc.) for SC300/Cortex-M3. Due to the similarities between SC300 and Cortex-M3 and with the accessibility of the platform, without a non-disclosure agreement (NDA), it is assessed as a suitable prototyping platform. When an unmodified bitstream containing the prototype SoC with the ARM Cortex-M3 processor is loaded to the FPGA together with program code (i.e., a hex file), it directly allows evaluation of the resource requirements and performance of the software implementation. Moreover, it is possible to modify the source code of the prototype SoC subsystem. This way coprocessor implementations can be integrated into the SoC and then get loaded onto the FPGA. For this the Advanced High-performance Bus (AHB) already contains a spare save that can be easily replaced by an accelerator (see Figure 4). Of course, a DesignStart environment cannot be shipped as a HW-based TPM solution due to the cost of the FPGA and hardware requirements of the relatively powerful SoC. An actual QR HW-TPM does most likely not require Ethernet, Audio, or VGA. Thus, DesignStart's main purpose is to provide an evaluation platform that is accessible and easy to program and use but that still allows to derive information and data relevant for actual hardware implementations.

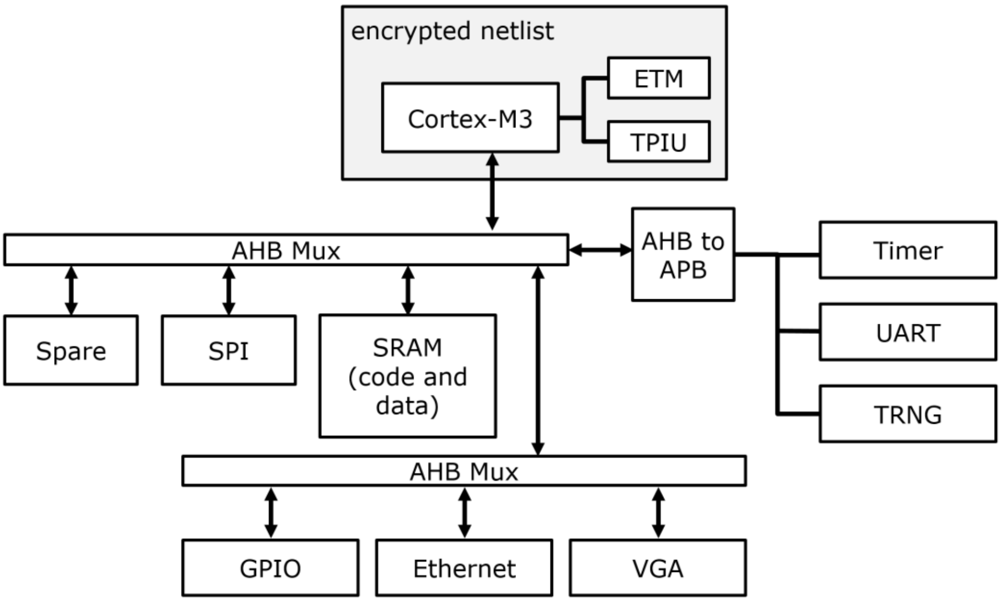

Figure 7: Simplified system overview on the DesignStart SoC

4.3 Architecture of the HW-TPM Demonstrator

As already pointed out, an FPGA platform running a typical embedded Processor is used to build the QR HW-TPM. To build its firmware, a fork of the IBM SW-TPM extended with QR algorithms (the "QR SW-TPM") is being used. Several adaptions were necessary to make this fork run in a bare metal environment – we will point them out subsequently. A general overview on this is given in Figure 8. As with the other QR TPM variants developed within FutureTPM, communication takes place via TCP/IP. Thus, the IBM SW-TPM running on the FPGA development board has been supplemented by a minimal operating system (OS) environment comprised of a TCP/IP stack and a scheduler that switches between the TCP/IP stack and the QR SW-TPM, which is depicted in Figure 9. For evaluation purposes we make use of the Ethernet Stack as it is easier to integrate than SPI. The Ethernet peripheral is part of the DesignStart FPGA design and low-level drivers are available.

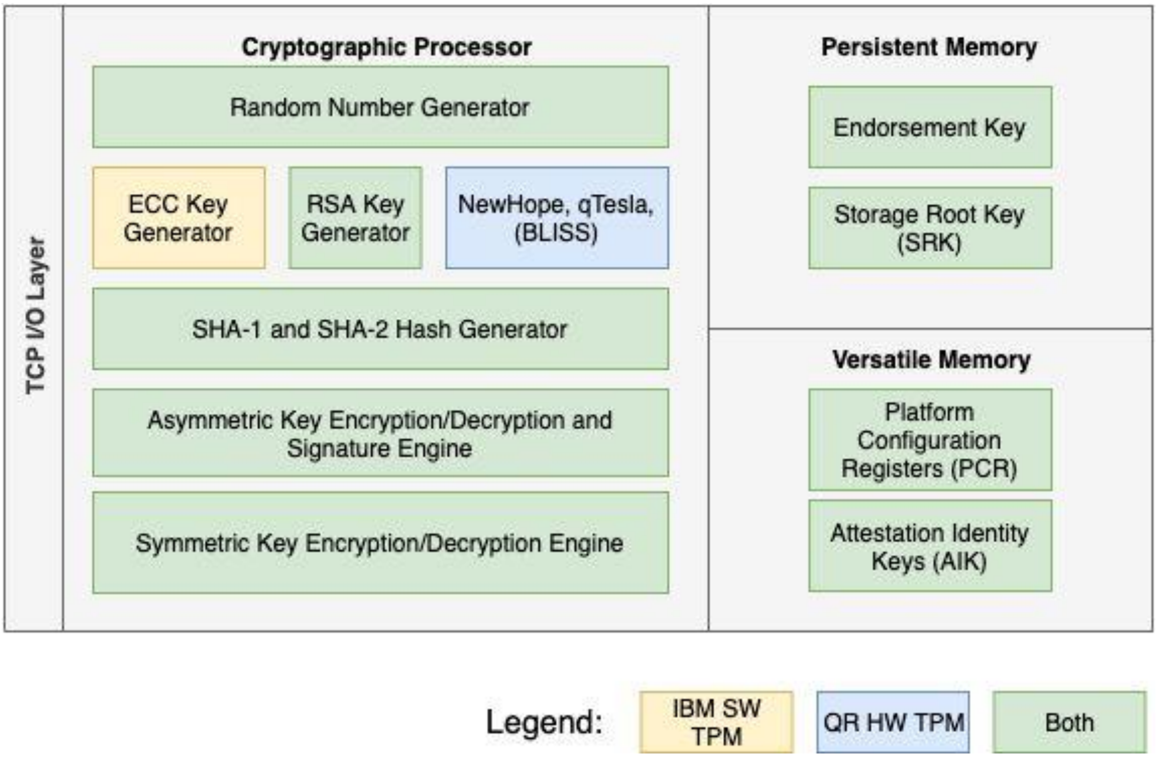

Figure 8: General overview of the QR HW-TPM and comparison to the IBM SW-TPM

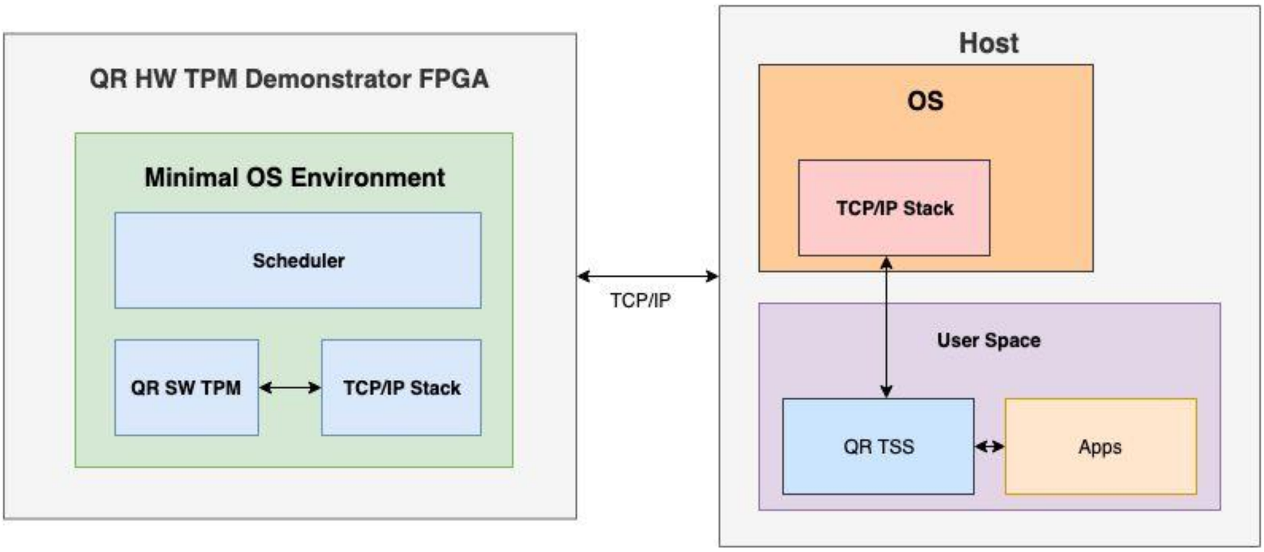

Figure 9: Block diagram of the QR HW-TPM demonstrator

To get this running, several adaptations were necessary:

- The IBM SW-TPM had to be ported to the FPGA platform. Most importantly, as the IBM SW-TPM depends on the OpenSSL library for cryptographic operations and OpenSSL cannot be built for the chosen FPGA platform, the OpenSSL library had to be replaced by the WolfCrypt[4] open-source cryptographic library.

[4] https://www.wolfssl.com/products/wolfcrypt-2/

- As TCP/IP stack the so-called BSD-socket-based *A Light-weight TCP/IP stack (lwIP)*[5] was chosen. Again, this implementation had to be ported to the FPGA platform. Moreover, it was necessary to adapt the IBM SW-TPM implementation to this TCP/IP stack and to intertwine both through callbacks.

- Because the lwIP stack needs to execute housekeeping functionality regularly, a minimalistic scheduler was implemented that allows for the IBM SW-TPM and the lwIP stack to run in parallel as two threads.

- Moreover, Ethernet drivers were ported to the chosen FPGA platform and compilation errors in the IBM SW-TPM implementation were corrected.

- A method to support non-volatile memory (NVM) emulation was implemented: As the used FPGA board is not equipped with NVM, encapsulation keys have been defined statically within the QR SW-TPM source code, which is sufficient for the purpose of demonstration.

Apart of these fundamental developments, the HW-TPM demonstrator has – similar to the other FutureTPM TPM platforms – been extended with the QR algorithms NewHope and BLISS. Details on that will be given subsequently.

4.4 Basic Operation of the QR HW-TPM Demonstrator

As already outlined, the communication with the QR HW TPM demonstrator is similar to the one described in D5.1 for the SW-TPM. It requires the usage of sockets to establish a TCP/IP connection between the QR SW-TPM running on the FPGA platform and a client. Through this connection, the QR HW TPM mimics the physical TPM command transmission interface (TCTI) layer found in the physical TPM.

After receiving data, the QR HW-TPM demonstrator executes the requested command by validating the session, its internal state, the command code, and its key handles. Then, the remaining data is forwarded to the Command Dispatcher where the unmarshalling functions are selected from the decoded command code and its data is unmarshalled. Finally, the command is executed. If there is any data to be returned to the client, the same procedure is applied in reverse order. The marshalling functions are obtained from the command code; the data is marshalled and then sent back to the client through the same TCP/IP connection.

4.5 New Endpoints

To support new QR algorithms, the following endpoints have been implemented in the QR HW-TPM demonstrator.

- **NewHope512-CCA**:
  - Key Generation: to generate a NewHope key one must use the standardized TPM2_Create, TPM2_CreateLoaded and TPM2_CreatePrimary functions, already provided by the TPM2 specification, with the TPM_ALG_NEWHOPE value and the corresponding security mode *k*.
  - TPM2_NEWHOPE_Enc encapsulates a shared secret, with the following parameters:
    - Input: *key_handle* for a loaded key handle to a NewHope public key.
    - Output: a *shared secret* and a *ciphertext* generated by the TPM.

[5] https://savannah.nongnu.org/projects/lwip/

- TPM2_NEWHOPE_Dec decapsulates a shared secret, with the following parameters:
  - Input: *key_handle* must reference a private loaded NewHope key and a *ciphertext* generated by TPM2_NEWHOPE_Enc.
  - Output: the TPM will decapsulate the cipher object to obtain the *shared_secret*.

- **BLISS-I**[6]:
  - Key Generation: to generate a BLISS key one must use the standardized TPM2_Create, TPM2_CreateLoaded and TPM2_CreatePrimary functions, already provided by the TPM2 specification, with the TPM_ALG_BLISS value and the corresponding security mode *mode*.
  - Signing: to sign a digest using the QR HW-TPM, the user must use the following functions: TPM2_Sign and TPM2_Quote.
  - Signature Verification: to verify a BLISS signature, the user must use the TPM2_VerifySignature function.

4.6 How to add new algorithms

As the HW-TPM demonstrator is based on a fork of the IBM SW-TPM, the strategy discussed in D5.1 for the addition of new algorithms applies here.

4.7 Using the HW-TPM demonstrator

The QR HW-TPM demonstrator FPGA board offers an RJ45 Ethernet interface. However, note that its TCP/IP implementation does not support the network management protocol DHCP. Thus, its address is fixed to 192.168.0.16.

[6] The BLISS variant GALACTICS was used because of its improvements over the original BLISS paper in that sampling is constant time, improving security. https://eprint.iacr.org/2019/511.pdf

After connecting the FPGA board to the power supply, the board automatically boots the QR SW-TPM and listens on the same TCP/IP ports as the IBM SW-TPM, that is, port 2321 for the command server and port 2322 for the platform server. Like the IBM SW-TPM, it is addressable by the TSS implementations. The basic structure is depicted in Figure 10.

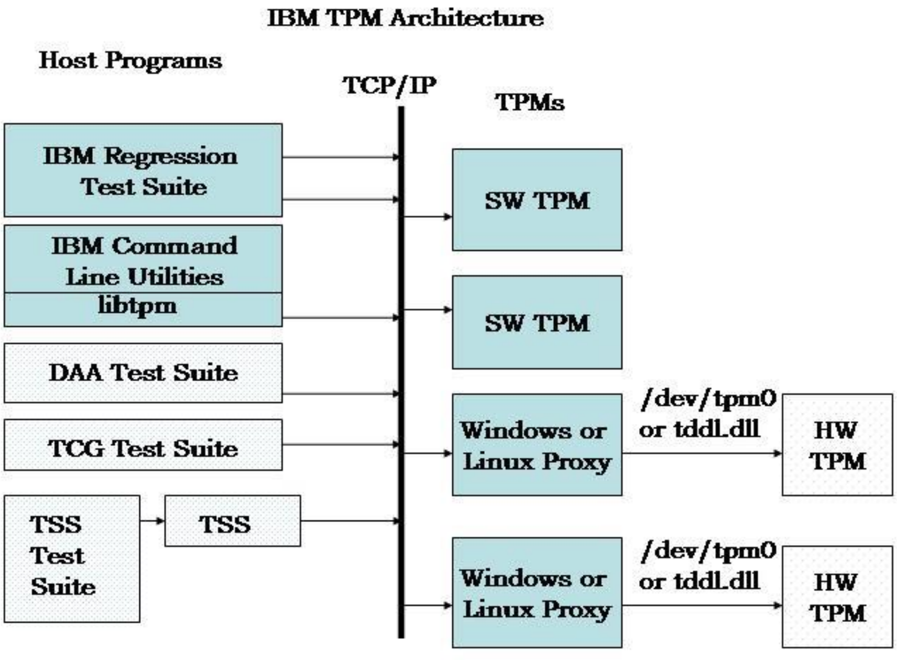

Figure 10: Architecture of IBM's TPM2.0 SW implementation
(from http://ibmswtpm.sourceforge.net/).

4.8 Example

This subsection will give an example, where Alice (the QR HW-TPM) communicates with Bob (a local QR SW-TPM) to transmit an encrypted and authenticated message using the QR HW-TPM demonstrator. The idea behind this example is motived by the following: Unlike key agreements, that are well-known from the world of classical cryptography (e.g. Diffie-Hellman key agreement [3]), where both participating parties contribute to a same extent to the exchanged secret, KEMs like NewHope are more like a one-way road. There, the sender fully determines the exchanged secret. However, using several runs of an IND-CCA-secure KEM like NewHope, we can mimic an authenticated (perfect-forward secret) key agreement:

First, Alice and Bob obtain long-term KEM keys and exchange their public keys. Second, Alice uses NewHope to send a secret s1 to Bob using Bob's long-term public key. Likewise, Bob sends a secret s2 to Alice encrypted under her long-term public key. The third step ensures the connection enjoys perfect forward secrecy. Alice generates a NewHope ephemeral key-pair and sends the resulting public-key to Bob. Bob then sends another secret s3 to Alice using Alice's ephemeral key. At the end, Alice and Bob can create a session key by hashing the concatenation of s1, s2 and s3. Using this session key, Alice encrypts a message, additionally signs it using her BLISS private key and sends the ciphertext together with the signature to Bob. Bob checks the signature and on success decrypts the ciphertext using the previously agreed session key.

Figure 11 depicts the communication diagram.

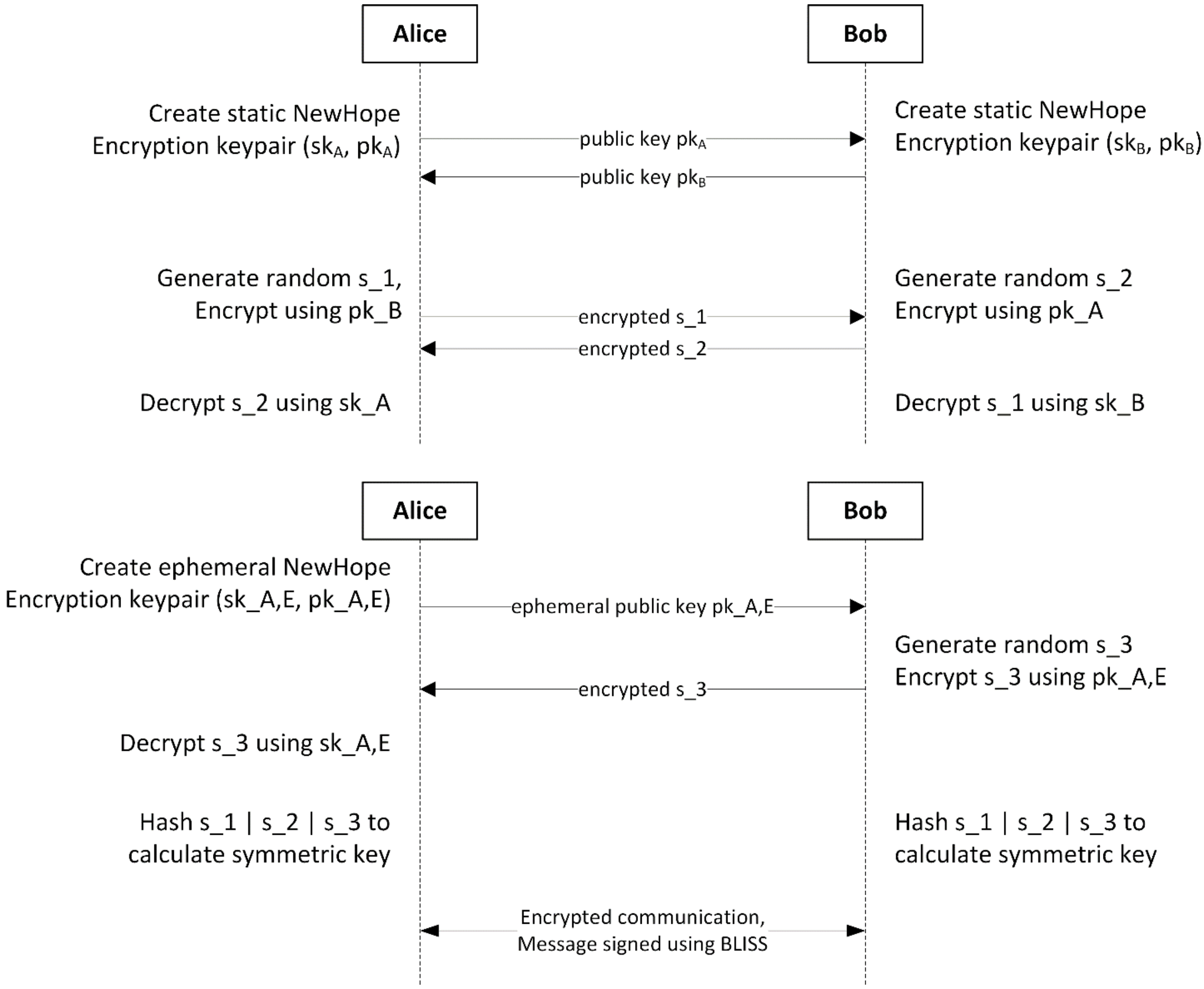

Figure 11: Communication Diagram between Alice and Bob for setup and repeated usage steps

The use-case is run using the TSS command line binaries, extended to support the PQ algorithms.

4.9 TSS Adaptations

As described above, the TSS is the piece of software on a client that will communicate with the TPM. As such a modified TSS that supports NewHope and BLISS was needed for the QR HW-TPM Demonstrator.

Similar to the approach chosen for the QR HW-TPM Demonstrator that is based on an open-source implementation published by IBM, the TSS is split into a library part that handles communication, and individual binaries that each implement calling of a single TPM command.

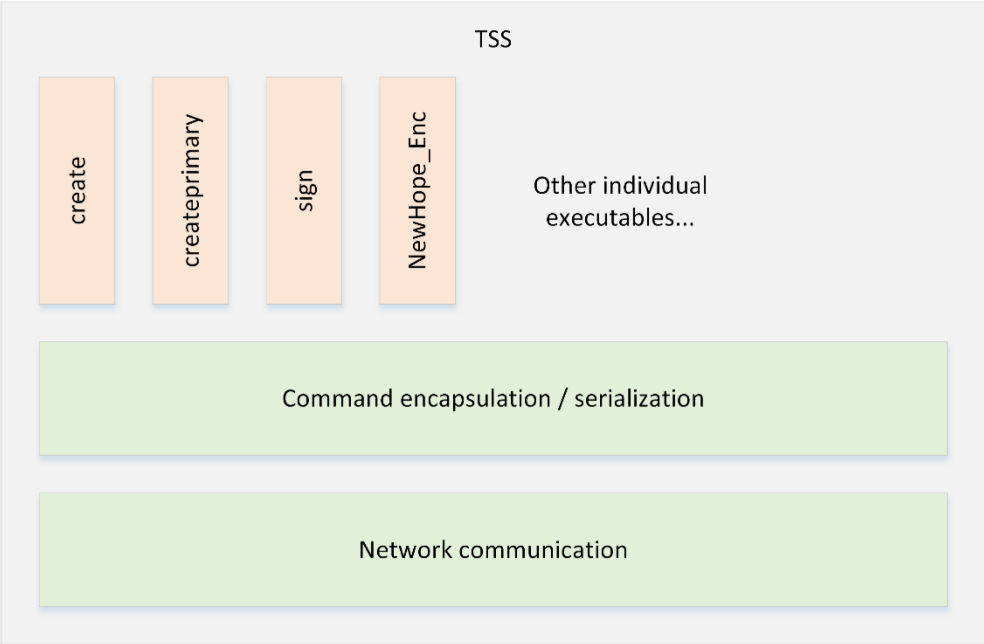

Figure 12 TSS architecture / parts

The communication part of the TSS could remain the same it already implements the TPM command transmission interface used by the QR HW-TPM Demonstrator.

The changes that were needed comprise mostly of support for the new algorithm IDs that are used to select the NewHope or BLISS algorithm. In detail this means:

- The *create* (calling TPM2_Create), *createprimary* (TPM2_CreatePrimary) and *loadexternal* (TPM2_CreateLoaded) executables needed to be extended by command line switches to use the algorithm IDs for NewHope and BLISS, and accept the according response packets. Data structures sent by the TPM are stored 1:1 in files for any output data.
- Similarly, *sign* (TPM2_Sign) and *verify* (TPM2_Verify) needed to be extended to allow usage of the BLISS algorithm ID.
- The NewHope integration is built in two new commands, TPM2_NEWHOPE_Enc and TPM2_NEWHOPE_Dec. For these two commands two new executables *NewHope_Enc* and *NewHope_Dec* where added. Both commands take parameters in accordance with other executables to write their output parameters to files or the standard output.

With these changes, all commands and command parameters that were added to the QR HW-TPM demonstrator can be used. To do so one has to define the TPM_SERVER_NAME environment variable to be set to the IP address of the demonstrator:

```
$ export TPM_SERVER_NAME="192.168.0.16"
```

With this setting, any of the executables can be called to send the according command to the QR HW-TPM demonstrator, as seen above in the description of the example use-case.

4.10 Hardware Acceleration of PQC

PQC routines can be accelerated in hardware with various degrees of flexibility. We classify hardware-based PQC accelerators in the dimensions: coupling, algorithmic flexibility, and extent of acceleration. In Figure 13 an abstract microcontroller architecture is provided. The CPU subsystem comprises a registerfile and is connected to a high performance main bus. The main bus is used to read and write the non-volatile memory (NVM) for code and permanently stored data and the SRAM for temporary data. A peripheral bus connects peripherals with lower performance requirements, like a timer or Universal Asynchronous Receiver Transmitter (UART). The grey components describe various possible options for the placement of hardware-based PQC accelerators. From the locality depicted in Figure 8 the level of coupling an accelerator has to the main CPU can be derived.

- High coupling - Instruction Set Extension (ISE): With ISE new instructions are added to the microarchitecture of the CPU that can be used to speedup certain operations of PQC schemes. Usually, the added instructions are stateless to ease integration and have a low overhead as a large part of the CPU subsystem is reused. However, when the CPU executes the instructions from the ISE it cannot perform other tasks. Exemplarily, an ISE may comprise of additional arithmetic instructions for the computation of the NTT butterfly or instructions that facilitate Gaussian sampling.
- Medium coupling - Co-Pro: The CPU can be extended by tightly coupled cryptographic co-processors (Co-Pro) that are optimized for PQC. Such co-processors typically have direct access to the registerfile and may also contain an internal registerfile and state. This direct access leads to decreased memory transfer times. They can thus operate on larger datasets and may reuse hardware of the Core. A Co-Pro could for example implement a full NTT accelerator or SHA-3 core.
- Loose-coupling - Accelerator on Bus: An accelerator for PQC may be added to the main bus of the microcontroller or to a peripheral bus if data transfer time is of lower importance. Similar to other peripherals the CPU transfers data from the registerfile/SRAM into the memory address space of the peripheral, starts an operations and then periodically controls the operation, feeds new data or just waits until the operation is finished. After an operation is finished the CPU obtains the result and copies it into the registerfile/SRAM. A peripheral on the system bus can usually execute complex operations independently of the CPU on internal memory. This frees the CPU for other tasks, e.g., I/O and allows integration into a SoC system with minimal dependencies on other subsystems. A loosely coupled accellerator may implement a full NTT accelerator, a SHA-3 core, or even a monolithic implementation of a full PQC scheme (e.g., NewHope).

Another dimension is the level of algorithmic flexibility the accelerator offers. A hardware-based accelerator may support acceleration for

- a single parameter set of a particular scheme (e.g., NewHope-512),
- all parameter sets of a particular scheme (e.g., NewHope),
- several parameter sets of a particular class of schemes (e.g., a binomial sampler for Kyber, NewHope, Saber),
- or may implement functions that can accelerate a wide variety of schemes efficiently.

Moreover, the extent of acceleration is an important additional dimension. An accelerator may

- implement a subfunction of a building block (e.g., a butterfly or a part of the SHA-2 round function),
- one complete building block (e.g., the SHA-3, NTT, or binomial sampling) or multiple building blocks, or
- a full scheme (e.g., NewHope).

With respect to related work, the highest performance is usually achieved by accelerators that are loosely coupled, accelerate a single parameter set and implement a full scheme. In such cases the accelerators can have their own high performance internal memory and are not constrained by the data path of the CPU (e.g., a 32-bit memory bus).

For a QR HW TPM, the coupling of the accelerator is assessed to be of minor importance and depends mostly on the required performance and achievable cost position. The most important property is the flexibility to accelerate at least several parameter sets of a particular class of schemes or wide variety of schemes. This is because of the uncertainties of the NIST competition and additional need for TCG to approve algorithms.

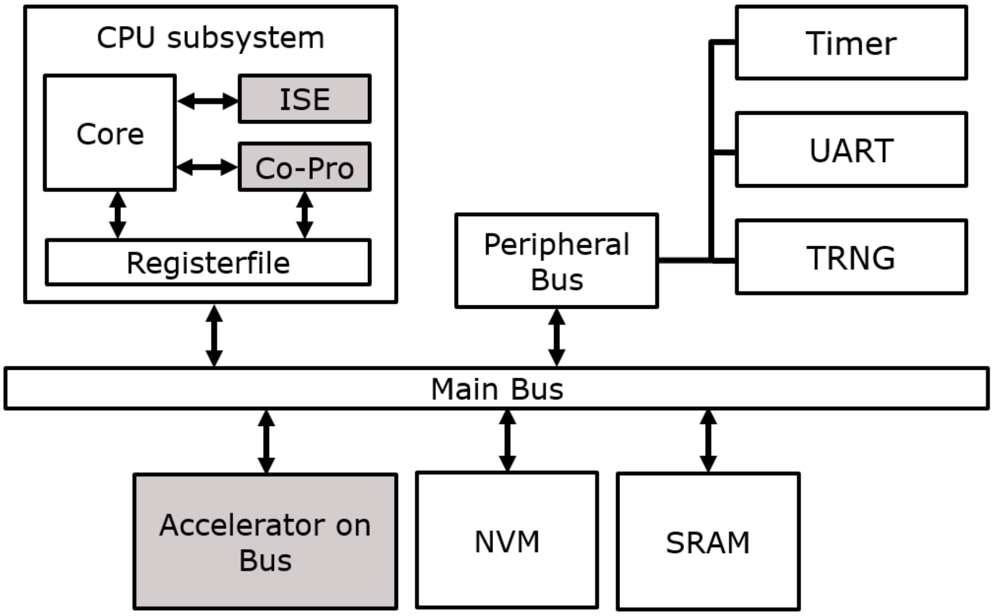

Figure 13 Option for hardware acceleration of PQC

Chapter 5 Conclusions and Next Steps

In this deliverable, we have discussed the concept of the Quantum-Resistant Virtual Trusted Platform Module, which is now ready in a first version by M30. Moreover, we have given an update to the final state of the Quantum-Resistant Software Trusted Platform Module and an overview of the architecture of the final version of the Quantum-Resistant Hardware Trusted Platform Module demonstrator.

The next steps will be to finalize the implementation of the QR Virtual TPM (Task 5.2) and further optimization of L-DAA (Task 5.5). The TPM SW Stack (Task 5.4) is updated to support all PQC algorithms implemented in the three QR TPM environments. In Task 5.5, the HW QR crypto-coprocessor is implemented to accelerate the Ring-LWE schemes NewHope and BLISS in HW. Finally, performance evaluations and testing are done.

<u>Outlook to D5.5 Deliverables</u>
The final deliverable document in M33 will provide a HW-TPM Coprocessor implementation, final optimizations for the SW-TPM environment and last updates on vTPM integration and testing results.

SW-TPM
- Possibly Bug fixes and optimization (esp. L-DAA), after T5.4

vTPM
- Possibly late results on Sphinx+, Bike, Rainbow integration and testing, after T5.4

HW-Coprocessor Demonstrator
- Implementation of HW-Coprocessor demonstrator
- Evaluation of cryptographic coprocessor and crypto performance testing

In case of very late technical implementation results after M33, updates will be documented in the final deliverable document of WP2 at M36.

List of Abbreviations

Abbreviation	Translation
AHB	Advanced High Performance Bus
BIOS	Basic Input Output System
DHCP	Dynamic Host Configuration Protocol
DSS	Digital Signature Scheme
FPGA	Field Programmable Gate Array
HW	Hardware
NDA	Non-disclosure Agreement
KEM	Key Encapsulation Mechanism
NVM	Non-Volatile Memory
NVRAM	Non-Volatile Random Access Memory
PCR	Platform Configuration Registers
QEMU	Quick EMUlator
QR	Quantum Resistant
SC	Secure Core
SH	Shell
SoC	System on Chip
SW	Software
TCTI	TPM command transmission interface
TIS	TPM Interface Specification
TPM	Trusted Platform Module
TSS	TPM Software Stack
UC	Use-case
VM	Virtual Machine
VNC	Virtual Network Computing
WP	Work Package

Bibliography

[1] E. Alkim, L. Ducas, T. Poppelmann and P. Schwabe, “Post-quantum Key Exchange - A New Hope,” in USENIX Security Symposium, Vancouver, Canada, 2016.

[2] L. Chen, N. E. Kassem, A. Lehmann and V. Lyubashevsky, “A Framework for Efficient Lattice-Based DAA,” in 1st Workshop on Cyber-Security Arms Race (CYSARM’19), London, United Kingdom, 2019.

[3] W. Diffie and M. E. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information Theory. 22 (6), p. 644–654, 1976.

[4] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler and D. Stehlé, “Dilithium - Submission to the NIST post-quantum project,” 30 03 2019. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round2.pdf

[5] Bindel, N., Akleylek, Alkim, E., Barreto, P., Buchmann, J., Eaton, E., . . . Zanon, G. (2019, 26 4). qTesla - Submission to the NIST post-quantum project. Retrieved from NIST: https://qtesla.org/wp-content/uploads/2019/04/qTESLA_round2_04.26.2019.pdf

[6] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler and D. Stehlé, “CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM,” in 2018 IEEE European Symposium on Security and Privacy, EuroS&P, London, UK, 2018.

[7] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler and D. Stehlé, “Dilithium - Submission to the NIST post-quantum project,” 30 03 2019. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round2.pdf.

[8] N. El Kassem, L. Chen, R. El Bansarkhani, A. El Kaafarani, J. Camenisch, P. Hough, P. Martins and L. Sousa, “More efficient, provably-secure direct anonymous attestation from lattices,” Future Generation Computer Systems, Volume 99, pp. 425-258, 2019

[9] TCG, “TCG Algorithm Registry,” 7 2 2018. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCG-_Algorithm_Registry_Rev_1.27_FinalPublication.pdf

[10] T.F. Consortium, “D2.2 – Second Report on New QR Cryptographic Primitives,” 2019

[11] T.F. Consortium, “D5.2 Second version of implementation”, Christian Hanser, 2019

[12] T.F. Consortium,“D6.3 Demonstrators Implementation Report – First Release”, Sotiris Koussouris, 2018

[13] T.F. Consortium, “D5.1 First version of implementation”, Daniele Sgandurra, 2017

[14] NTTRU: “Truly Fast NTRU Using NTT”, Vadim Lyubashevsky and Gregor Seiler, Cryptology ePrint Archive, Report 2019/040, 2019, https://eprint.iacr.org/2019/040

[15] NIST Round2 submission: CRYSTALS-KYBER, Peter Schwabe and Roberto Avanzi and Joppe Bos and Leo Ducas and Eike Kiltz and Tancrede Lepoint and Vadim Lyubashevsky and John M. Schanck and Gregor Seiler and Damien Stehle, 2019, https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/CRYSTALS-Dilithium-Round2.zip ;

[16] NIST Round2 submission: CRYSTALS-DILITHIUM, Vadim Lyubashevsky and Leo Ducas and Eike Kiltz and Tancrede Lepoint and Peter Schwabe and Gregor Seiler and Damien Stehle, 2019, https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/CRYSTALS-Dilithium-Round2.zip

[17] T.F. Consortium, “D2.1 First Report on New QR Cryptographic Primitives”, Gagliatdoni, T., 2018

[18] Cucurull, J and Guasch, S.: “Virtual TPM for a secure cloud: fallacy or reality?” Scytl Secure Electronic Voting, 2014.

[19] TCG, "Trusted Platform Module 2.0: A Brief Introduction", 2015 [Online]. Available at: https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf

[20] Aragon, N., Barreto P. S. L. M., Bettaieb S., Bidoux L., Blazy, O., Deneuville J., Gaborit, P., Gueron, S., Güneysu, T., Melchor, C. A., Misoczki, R., Persichetti, E., Sendrier, N., Tillich. J., Vasseur, V. and Zémor, G., "*BIKE: Bit Flipping Key Encapsulation", Round 2 Submission* 2019. [Online]. Available at: https://bikesuite.org/files/round2/spec/BIKE-Spec-Round2.2019.03.30.pdf

[21] Aumasson J., Bernstein D. J., Dobraunig C., Eichlseder M., Fluhrer S., Gazdag S., Hülsing A., Kampanakis P., Kölbl S., Lange T., Lauridsen M. M., Mendel F., Niederhagen R., Rechberger C., Rijneveld J. and Schwabe P., *SPHINCS+ Submission to the NIST post-quantum project*, 2019

[22] Ding J., Chen M., Petzoldt A., Schmidt D. and Yang B. Y., "*Rainbow", The 2nd NIST Standardization Conference for Post-Quantum Cryptosystems*, 2019.

[23] Aragon, N., Barreto P. S. L. M., Bettaieb S., Bidoux L., Blazy, O., Deneuville J., Gaborit, P., Gueron, S., Güneysu, T., Melchor, C. A., Misoczki, R., "*BIKE: Bit Flipping Key Encapsulation", Round 2 Submission*.

[24] T.F. Consortium, "D5.3 Final version of implementation", Michael Albrecht, 2020

Appendices

Appendix A: SPHINCS+ Implementation

To implement SPHINCS+, the consortium has used the standard codebase from the SPHINCS+ site: https://sphincs.org/ The SPHINCS+ source files were then renamed with a prefix of “sphincsplus” in order to make them identifiably belonging to SPHINCS+ when ported into QR-Libtpms. The “#Include” statements within the source files were also appropriately renamed.

The sphincsplus-Makefile was modified (due to name changes) and executed. It produces an executable, sphincsplus-PQCgenKAT_sign. Before integrating into QR-Libtpms, the SPHINCS+ code was unit tested by executing sphincsplus-PQCgenKAT_sign.

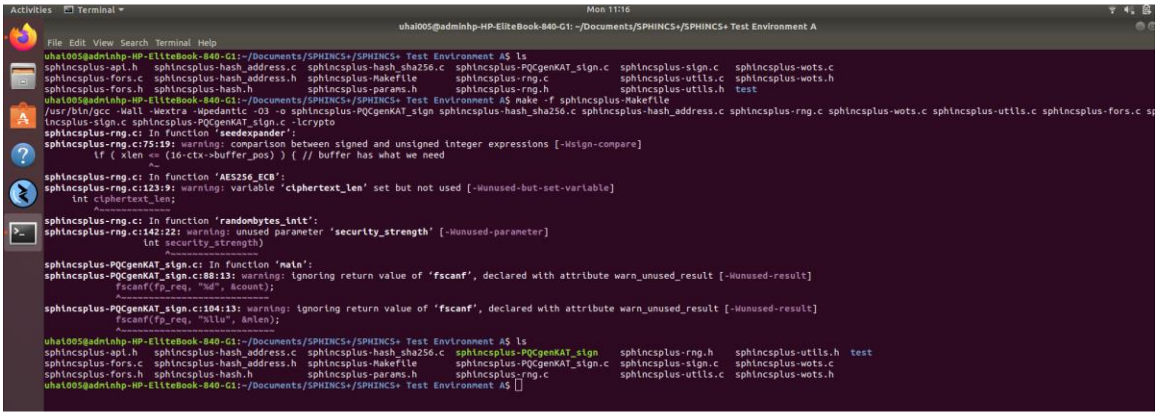

Figure 14: SPHINCS+ Compilation

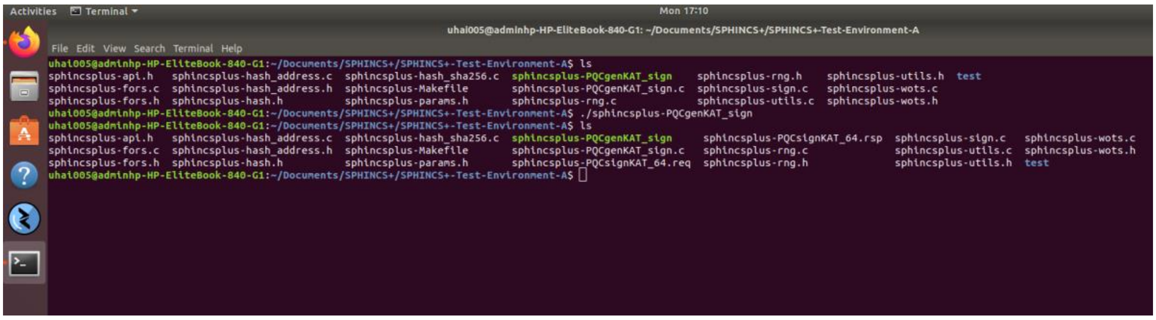

Figure 15: SPHINCS+ Unit Test

The unit test produced a response file, sphincsplus-PQCsignKAT_64.rsp which contains 100 test result records, each with a different message length. The following 2 figures show the SPHINCS+ unit test results for record 0 to 4 and for record 95 to 99.

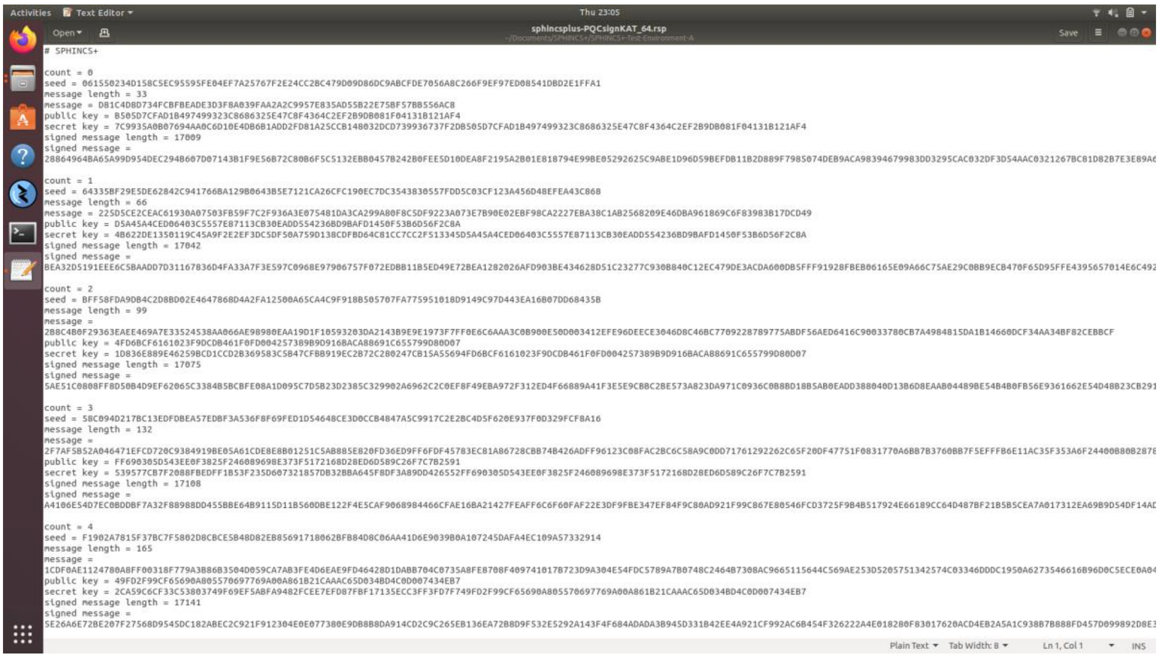

Figure 16: SPHINCS+ Unit Test Results (Record 0 to 4)

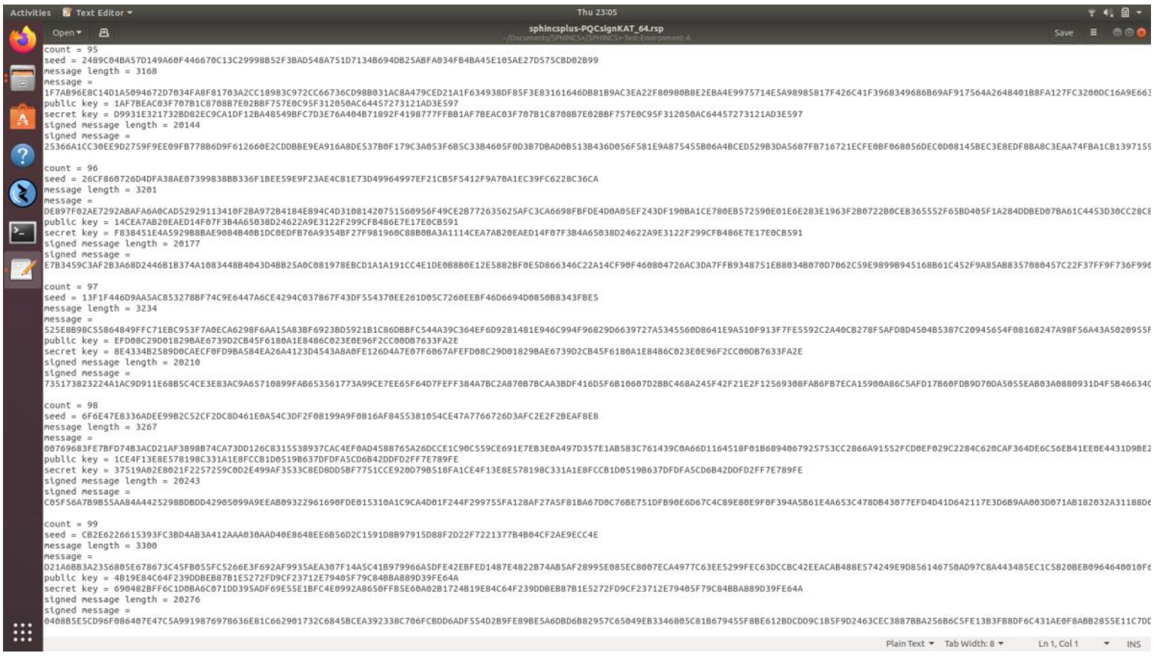

Figure 17: SPHINCS+ Unit Test Results (Record 95 to 99)

The unit test results demonstrated that SPHINCS+ key generation, signature generation and verification are working correctly which provided the assurance that the SPHINCS+ code is fit for purpose.

Following the instructions given in the D5.1 [13] document on how to add new algorithms to TPM, the consortium has made the relevant code changes to SW-TPM for the following: AlgorithmCap.c, CryptUtil.c, Implementation.h, InternalRoutines.h, Makefile-common, Marshal.c, Marshal_fp.h, TpmTypes.h, Unmarshal.c and Unmarshal_fp.h.

In addition, two new files were created: CryptSphincsPlus.c and CryptSphincsPlus_fp.h.

CryptSphincsPlus.c has three important functions:

1. Key Generation
2. Signature Generation

3. Verification of Signature

The code was taken from sphincsplus-sign.c. The interfaces were then modified to adhere to the TPM specification:

Key Generation

```
CryptSphincsPlusGenerateKey(
        // IN/OUT: The object structure in which the key is created.
        OBJECT              *sphincsplusKey,
        // IN: if not NULL, the deterministic RNG state
        RAND_STATE          *rand
        );
```

Signature Generation

```
CryptSphincsPlusSign(
        TPMT_SIGNATURE  *sigOut,
        OBJECT          *key,       // IN: key to use
        TPM2B_DIGEST    *hIn        // IN: the digest to sign
        );
```

Verification of Signature

```
CryptSphincsPlusValidateSignature
    TPMT_SIGNATURE      *sig,       // IN: signature
    OBJECT              *key,       // IN: public modulus
    TPM2B_DIGEST        *digest     // IN: The digest being validated
    );
```

The consortium has made similar code changes to TSS. In the Utils folder: certify.c, create.c, createloaded.c, createprimary.c, loadexternal.c, objecttemplates.c, quote.c, sign.c, tssmarshal.c, tssprint.c, Unmarshal.c and verifysignature.c. In the Utils/ibmtss folder: Implementation.h, TPM_Types.h, tssmarshal.h, tssprint.h and Unmarshal_fp.h.

Appendix B: Rainbow Implementation

To implement Rainbow, the consortium has used the Rainbow codebase from the NIST site: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions. The Rainbow source files were then renamed with a prefix of "rainbow" in order to make them identifiably belonging to Rainbow when ported into QR-Libtpms. The "#Include" statements within the source files were also appropriately renamed.

The rainbow-Makefile was modified (due to name changes) and executed. It produced four executables: rainbow-PQCgenKAT_sign, rainbow-genkey, rainbow-gensignature and rainbow-verify. Before integrating into QR-Libtpms, the Rainbow code was unit tested by executing rainbow-PQCgenKAT_sign. The test produced a response file, rainbow-PQCsignKAT_92960.rsp, which contains 100 test result records, each with a different message length. The following two figures show the Rainbow unit test results for record 0 to 3 and for record 96 to 99 respectively.

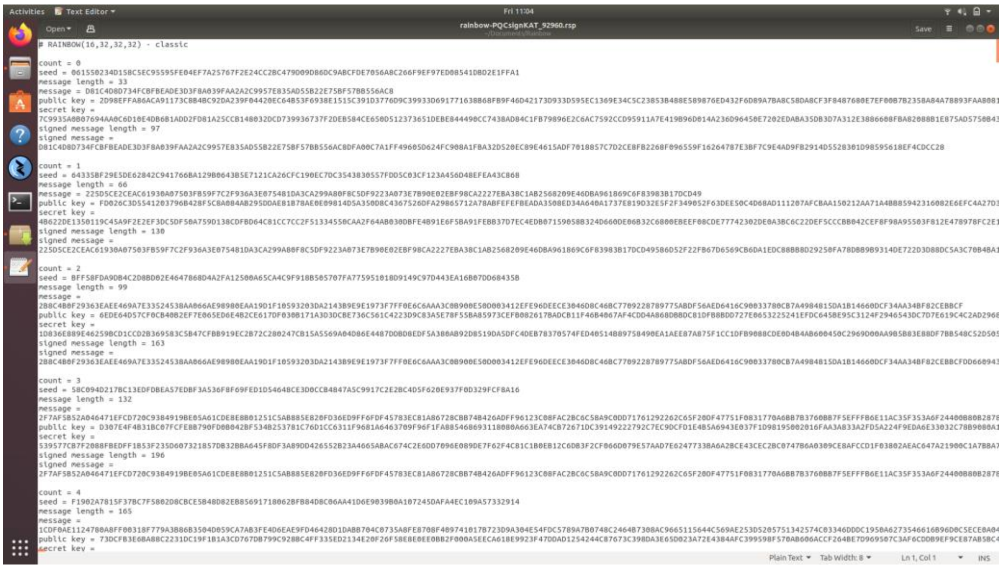

Figure 18: Rainbow Unit Test Results (Record 0 to 3)

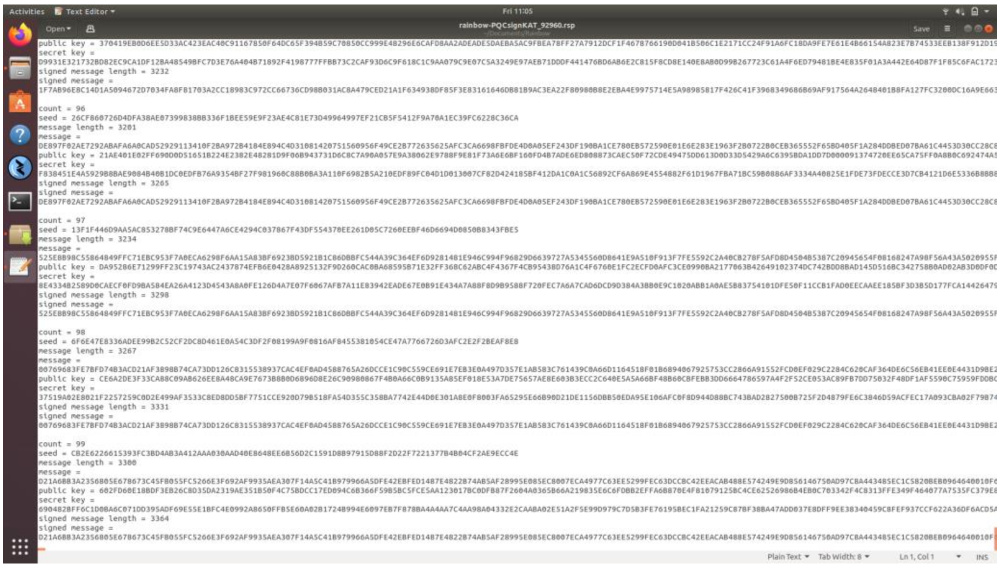

Figure 19: Rainbow Unit Test Results (Record 96 to 99)

An additional test was done by performing the following steps:

1. Create a message file for signing.
2. Create a signature file.
3. Execute rainbow-genkey with parameters [pk_file_name] [sk_file_name].
4. Execute rainbow-gensignature with parameters [sk_file_name] [file_to_be_signed].
5. Copy the signature generated into the signature_file.
6. Execute rainbow-verify with parameters [pk-file-name] [signature_file_name] [message_file_name].

See the below figure for steps 3, 4 and 6.

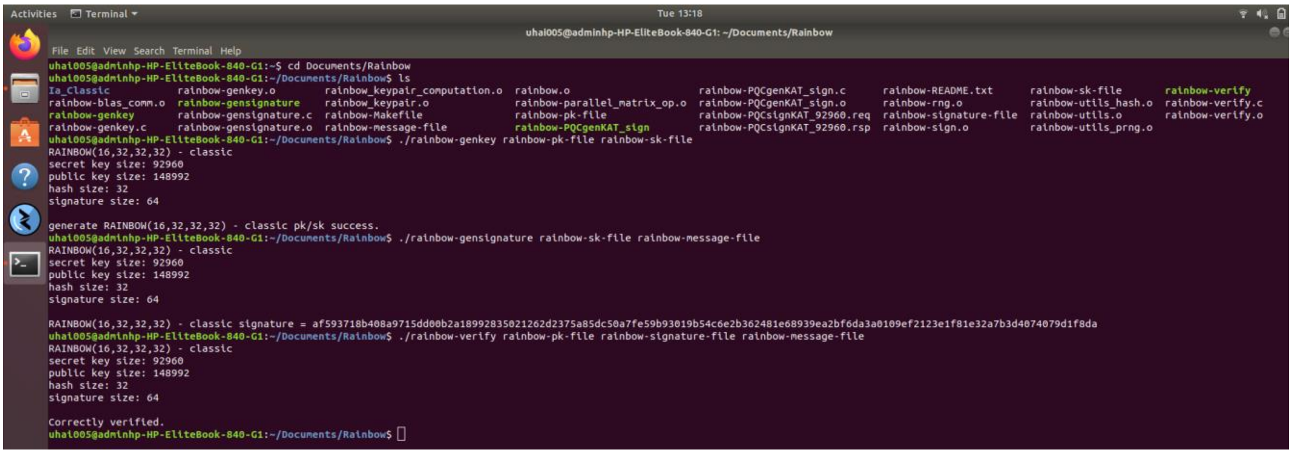

Figure 20: Rainbow Unit Test - Key Generation, Signature Generation and Verification

The unit test results demonstrated that Rainbow key generation, signature generation and verification are working correctly which provided the assurance that the Rainbow code is fit for purpose.

Using the SW-TPM code that has been modified to include SPHINCS+, the consortium has made further changes to include Rainbow: AlgorithmCap.c, CryptUtil.c, Implementation.h, InternalRoutines.h, Makefile-common, Marshal.c, Marshal_fp.h, TpmTypes.h, Unmarshal.c and Unmarshal_fp.h.

In addition, two new files were created: CryptRainbow.c and CryptRainbow_fp.h.

CryptRainbow.c has three important functions:

- Key Generation
- Signature Generation
- Verification of Signature

The code was taken from rainbow-sign.c. The interfaces were then modified to adhere to the TPM specification:

Key Generation

```
CryptRainbowGenerateKey(
        // IN/OUT: The object structure in which the key is created.
        OBJECT          *rainbowKey,
        // IN: if not NULL, the deterministic RNG state
        RAND_STATE      *rand
        );
```

Signature Generation

```
CryptRainbowSign(
        TPMT_SIGNATURE  *sigOut,
        OBJECT          *key,       // IN: key to use
        TPM2B_DIGEST    *hIn        // IN: the digest to sign
        );
```

Verification of Signature

```
CryptRainbowValidateSignature(
    TPMT_SIGNATURE      *sig,       // IN: signature
    OBJECT              *key,       // IN: public modulus
    TPM2B_DIGEST        *digest     // IN: The digest being validated
    );
```

Using the TSS code that has been modified to include SPHINCS+, the consortium has made further changes to include Rainbow. In the Utils folder: certify.c, create.c, createloaded.c, createprimary.c, loadexternal.c, objecttemplates.c, quote.c, sign.c, tssmarshal.c, tssprint.c, Unmarshal.c and verifysignature.c. In the Utils/ibmtss folder: Implementation.h, TPM_Types.h, tssmarshal.h, tssprint.h and Unmarshal_fp.h.

Appendix C: BIKE Implementation

To implement BIKE, the consortium has used the standard codebase from the BIKE site: https://bikesuite.org/. The BIKE source files were then renamed with a prefix of 'BIKE-' in order to make them identifiably belonging to BIKE when ported into QR-Libtpms. The "#Include" statements within the source files were also appropriately renamed.

The BIKE-Makefile was modified (due to name changes) and executed (see Figure 10). It produced four executables: bike-nist-kat. Before integrating into QR-Libtpms, the BIKE code was unit tested by executing bike-nist-kat.

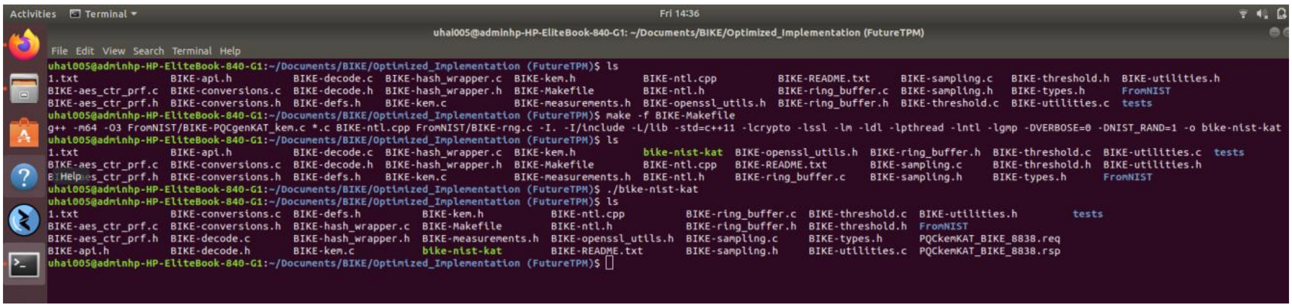

Figure 21: BIKE Unit Test

The test produced a response file, PQCkemKAT_BIKE_8838.rsp, which contains 100 test result records. The following two figures show the BIKE unit test results for record 0 to 4 and for record 96 to 99 respectively.

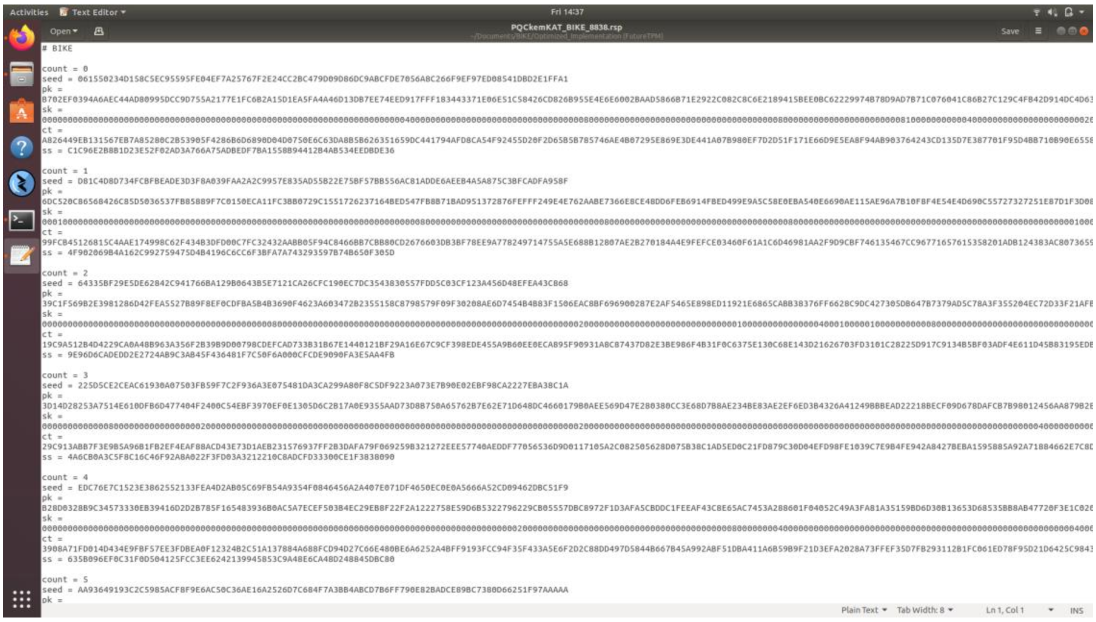

Figure 22: BIKE Unit Test Results (Record 0 to 4)

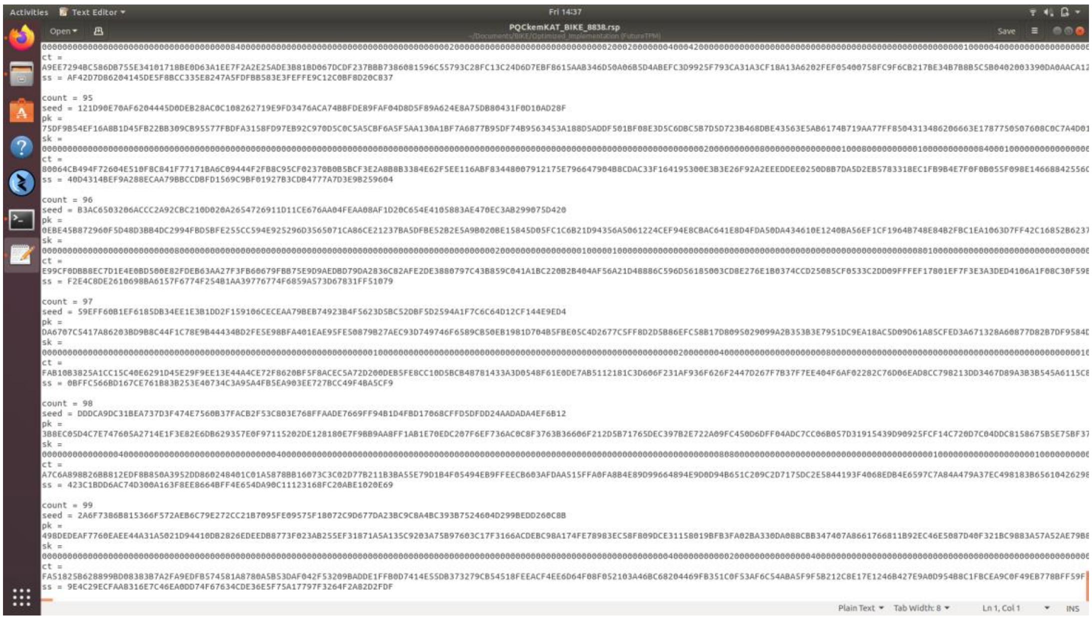

Figure 23: BIKE Unit Test Results (Record 95 to 99)

The unit test results demonstrated that BIKE key generation, encapsulation and decapsulation are working correctly which provided the assurance that the BIKE code is fit for purpose.

Using the SW-TPM code that has been modified to include SPHINCS+ and Rainbow, the consortium has made further changes to include BIKE: AlgorithmCap.c, AsymmetricCommands.c, CommandAttributeData.h, CommandDispatchData,h, Commands.h, CryptUtil.c, Implementation.h, InternalRoutines.h, Makefile-common.txt, Marshal.c, Marshal_fp.h, Object.c, TpmTypes.h, Unmarshal.c and Unmarshal_fp.h.

In addition, six new files were created: CryptBIKE.c, CryptBIKE_fp.h, BIKE_Dec_fp.h, BIKE_Decrypt_fp.h, BIKE_Enc_fp.h and BIKE_Encrypt_fp.h

CryptBIKE.c has five important functions:

- Key Generation
- Encapsulation
- Decapsulation
- Encryption
- Decryption

The code was taken from BIKE-kem.c. The interfaces were then modified to adhere to the TPM specification:

Key Generation

```
CryptBIKEGenerateKey(
        // IN/OUT: The object structure in which the key is created.
        OBJECT              *BIKEKey,
        // IN: if not NULL, the deterministic RNG state
        RAND_STATE          *rand
        );
```

Encapsulation

```
CryptBIKEEncapsulate(
        // IN: The object structure which contains the public key used in
        // the encapsulation.
        TPMT_PUBLIC                     *publicArea,
        // OUT: the shared key
        TPM2B_BIKE_SHARED_KEY           *ss,
        // OUT: the cipher text
        TPM2B_BIKE_CIPHER_TEXT          *ct
        );
```

Decapsulation

```
CryptBIKEDecapsulate(
        // IN: The object structure which contains the secret key used in
        // the decapsulation.
        TPMT_SENSITIVE                  *sensitive,
        // IN: BIKE security mode
        TPMT_BIKE_SECURITY              k,
        // IN: the cipher text.
        TPM2B_BIKE_CIPHER_TEXT          *ct,
        // OUT: the shared key.
        TPM2B_BIKE_SHARED_KEY           *ss
    );
```

Encryption

```
CryptBIKEEncrypt(
        // OUT: The encrypted data
        TPM2B_BIKE_ENCRYPT              *cOut,
        // IN: The object structure in which the key is created.
        OBJECT                          *BIKEKey,
        // IN: the data to encrypt
        TPM2B                           *dIn
    );
```

Decryption

```
CryptBIKEDecrypt(
        // OUT: The decrypted data
        TPM2B                           *cOut,
        // IN: The object structure in which the key is created.
        OBJECT                          *BIKEKey,
        // IN: the data to decrypt
        TPM2B_BIKE_ENCRYPT              *dIn
    );
```

Using the TSS code that has been modified to include SPHINCS+ and Rainbow, the consortium has made further changes to include BIKE. In the Utils folder: CommandAttributeData.c, Commands.c, Commands_fp.h, create.c, createloaded.c, createprimary.c, loadexternal.c, objecttemplates.c, objecttemplates.h, tssauth20.c, tssmarshal.c, and Unmarshal.c. In the Utils/ibmtss folder: Implementation.h, Parameters.h, TPM_Types.h, tssmarshal.h and Unmarshal_fp.h.

Appendix D: SPHINCS+, Rainbow and BIKE Integration Into QR-Libtpms

1. The SPHINCS+, Rainbow and BIKE source files were copied to ./Docker_libtpmsQR/libtpms_inescID/src/tmp2/crypto/openssl folder:

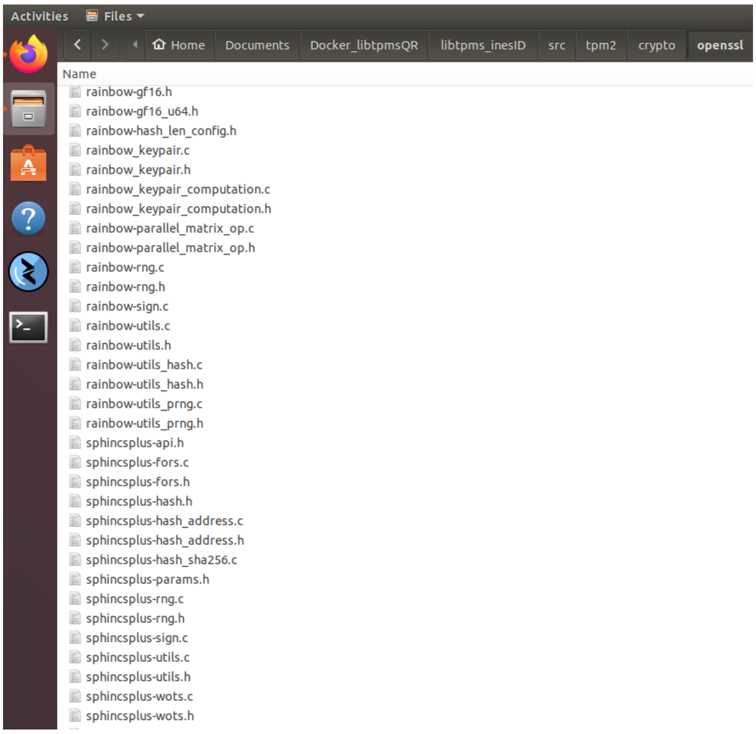

Figure 24: SPHINCS Code Copied Into QR-Libtpms

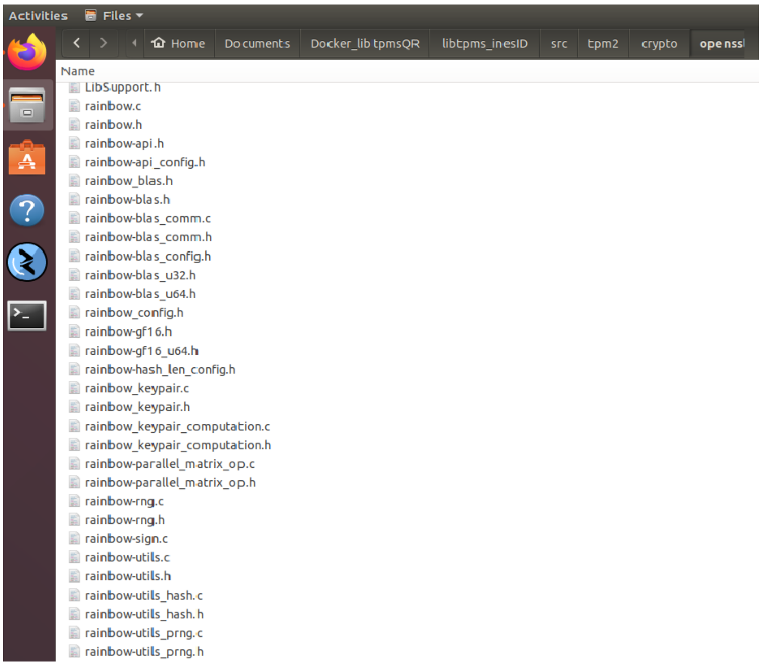

Figure 25: Rainbow Code Copied Into QR-Libtpms

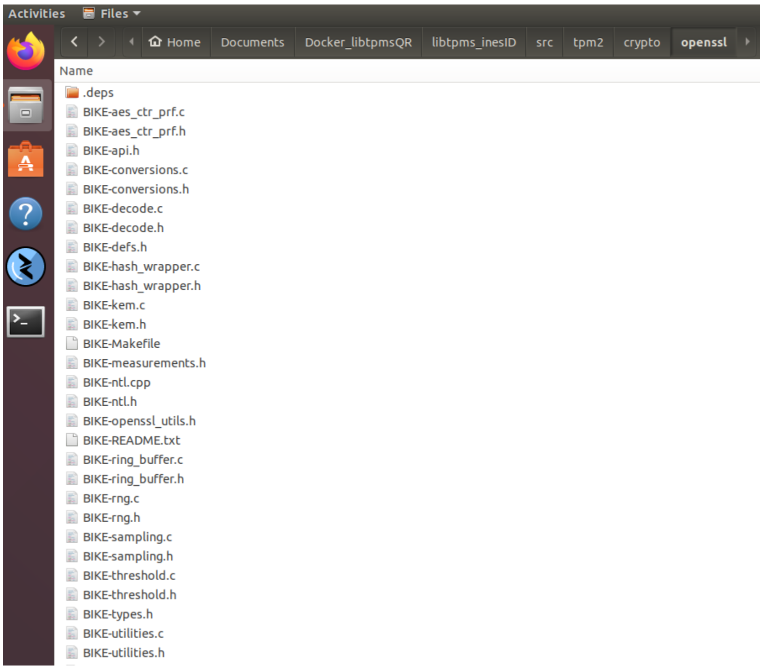

Figure 26: BIKE Code Copied Into QR-Libtpms

2. CryptSphincsPlus.c, CryptRainbow.c, CryptBIKE.c and CryptUtil.c were copied to ./Docker_libtpmsQR/libtpms_inescID/src/tmp2/crypto/openssl folder:

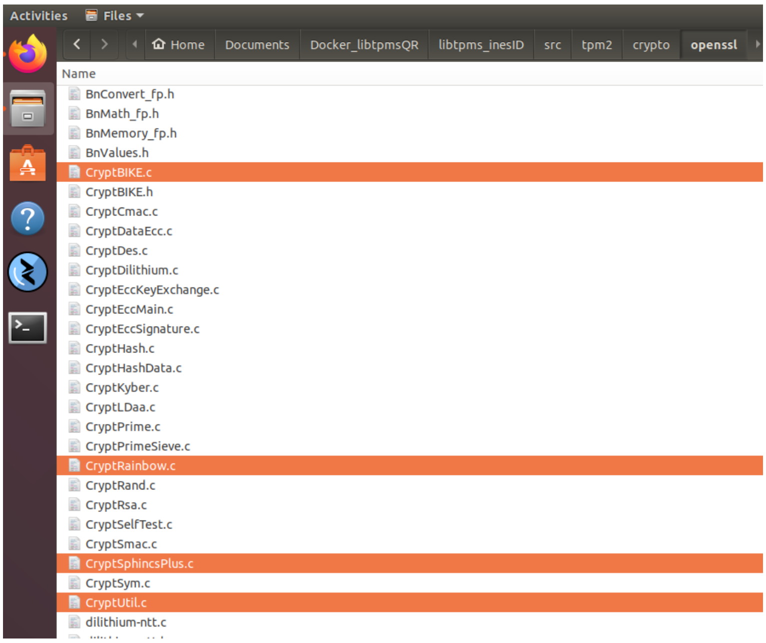

Figure 27: CryptSphincsPlus.c, CryptRainbow.c, CryptBIKE.c and CryptUtil.c Copied Into QR-Libtpms

3. CryptSphincsPlus_fp.h, CryptRainbow_fp.h and CryptBIKE_fp.h were copied to ./Docker_libtpmsQR/libtpms_inescID/src/tmp2/crypto folder:

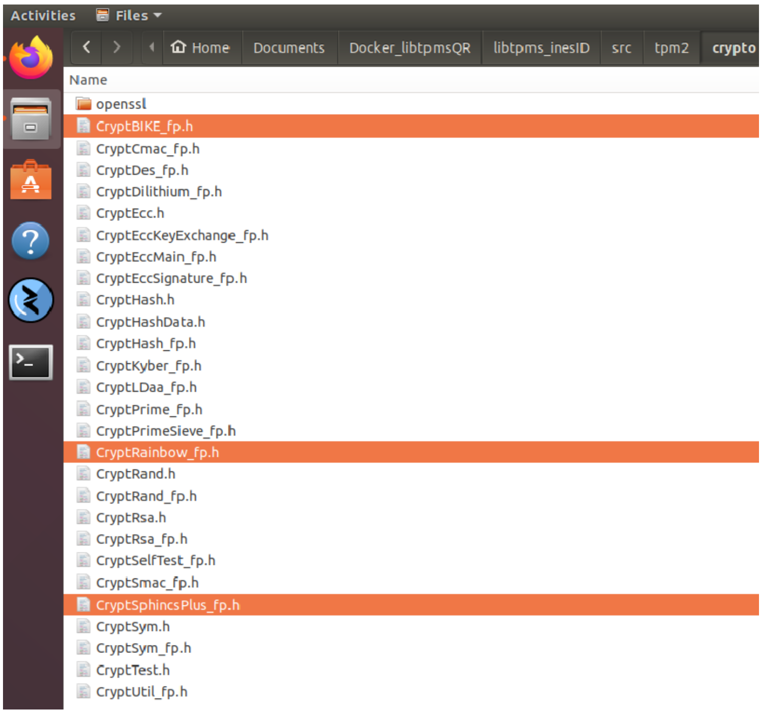

Figure 28: CryptSphincsPlus_fp.h, CryptRainbow_fp.h and CryptBIKE_fp.h Copied Into QR-Libtpms

4. The following new BIKE files were copied to ./Docker_libtpmsQR/libtpms_inescID/src/tmp2 folder: BIKE_Dec_fp.h, BIKE_Decrypt_fp.h, BIKE_Enc_fp.h and BIKE_Encrypt_fp.h.

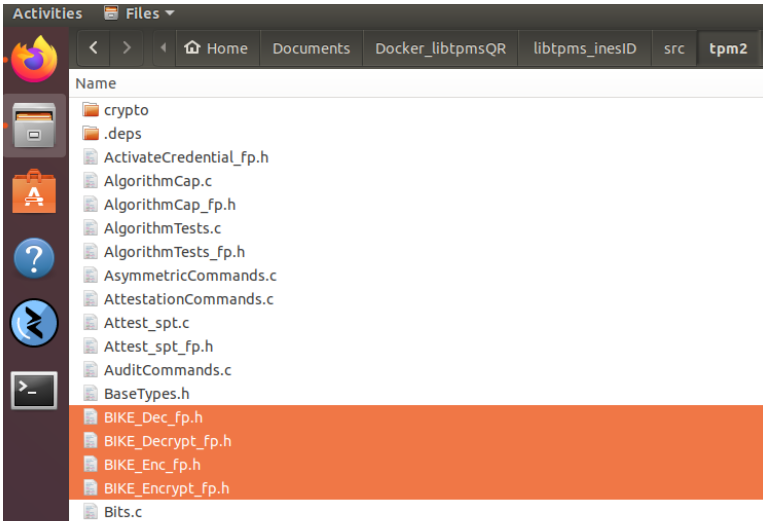

Figure 29: New BIKE files Copied Into QR-Libtpms

5. The SW-TPM code (modified to include SPHINCS+, Rainbow and BIKE) were copied to ./Docker_libtpmsQR/libtpms_inescID/src/tmp2 folder: AlgorithmCap.c, AsymmetricCommands.c, CommandAttributeData.h, CommandDispatchData.h, Cryptutil.c Implementation.h, InternalRoutines.h, Marshal.c, Marshal_fp.h, Object.c, TpmTypes.h, Unmarshal.c and Unmarshal_fp.h.

6. The TSS code in the utils folder (modified to include SPHINCS+, Rainbow and BIKE) were copied to ./Docker_libtpmsQR/tpm_sim/tss/utils folder: CommandAttributeData.c, Commands.c, Commands_fp.h, certify.c, create.c, createloaded.c, createprimary.c, loadexternal.c, objecttemplates.c, objecttemplates.h, quote.c, sign.c, tssauth20.c, tssmarshal.c, tssprint.c, Unmarshal.c and verifysignature.c.

7. The TSS code in the utils/ibmtss folder (modified to include SPHINCS+, Rainbow and BIKE) were copied to ./Docker_libtpmsQR/tpm_sim/tss/utils/ibmtss folder: Implementation.h, Parameters.h, TPM_Types.h, tssmarshal.h, tssprint.h and Unmarshal_fp.h.

8. The following new BIKE files were copied to ./Docker_libtpmsQR/tpm_sim/tss/utils/ibmtss folder: BIKE_Dec_fp.h, BIKE_Decrypt_fp.h, BIKE_Enc_fp.h and BIKE_Encrypt_fp.h.

9. Three Makefiles in the ./Docker_libtpmsQR/libtpms_inesID/src folder have been modified to include SPHINCS+, Rainbow and BIKE. They are:

    - Makefile
    - Makefile.am
    - Makefile.in

10. Build the docker file by running the command: **sudo docker build . -t swtpm_qr -f ./Dockerfile**

The following two figures show that SPHINCS+, Rainbow and BIKE were successfully compiled into QR-Libtpms:

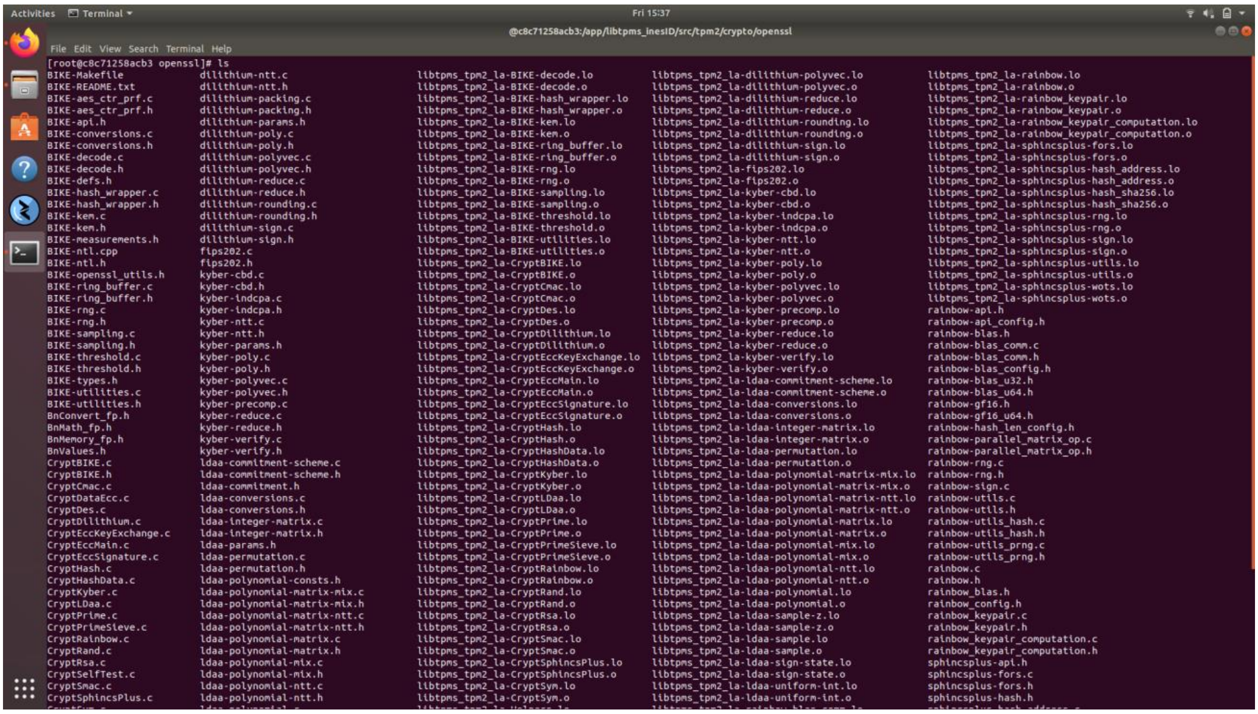

Figure 30: SPHINCS+, Rainbow and BIKE Compiled Into QR-Libtpms

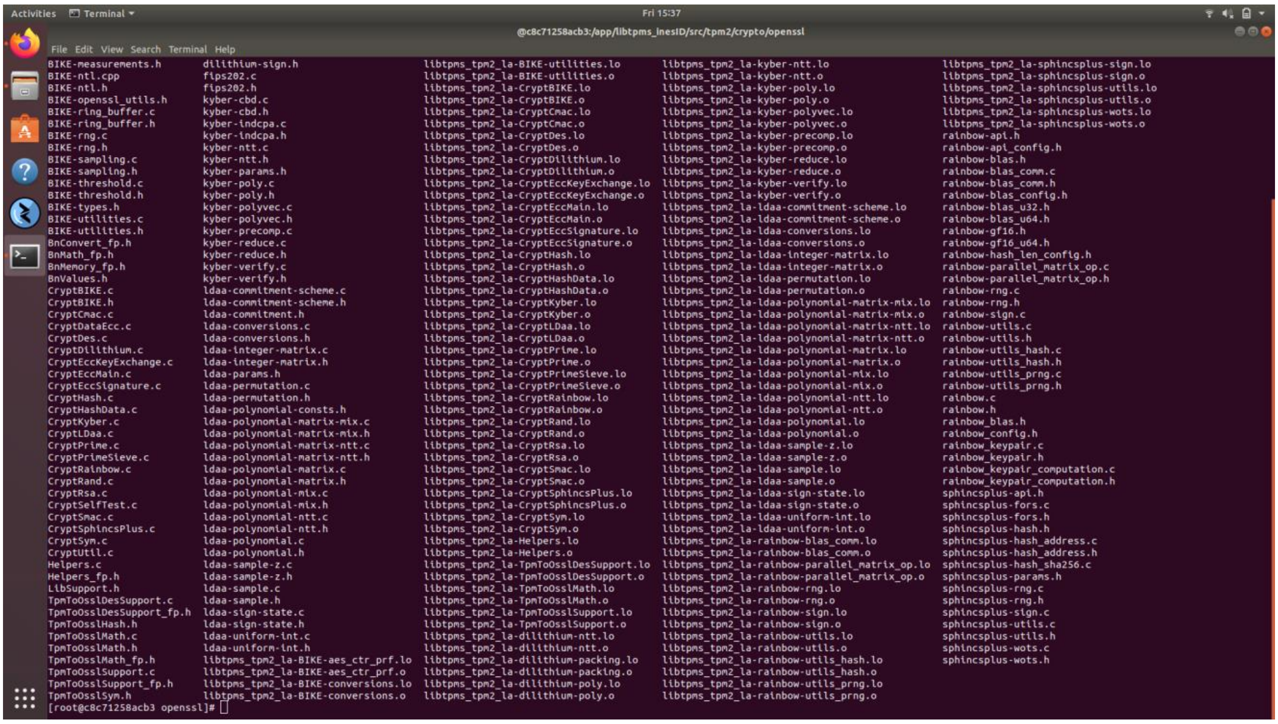

Figure 31: SPHINCS+, Rainbow and BIKE Compiled Into QR-Libtpms

Appendix E: TSS2 Issues Found In V-TPM

We have found an issue with TSS2 in that sometimes it is not possible to send commands to the V-TPM. In particular, some commands do work on the V-TPM as found, such as `get_capability`, but anything which requires some key generation to be uses won't return anything. This issue might happen when making TSS2 from source, and this is required for the QR algorithms to be implemented. A ticket regarding the issue found has been raised GitHub repo of the project. With the Docker/V-TPM integration, this is sometime problematic to use as V-TPM may crash during its install and some network/transportation issues. Another example of this issue would be that sometimes the V-TPM docker image won't verify the keys as the received length is drastically shorter than the sent length.

When running `tssverifysignature` this issue was found, as shown in the following:

```
./verifysignature -hk 80000002 -dilithium -if enc.bin -is sig.bin -v
verifysignature: Hashing message file enc.bin with halg 000b
verifysignature: Copying hash
 verifysignature: hash length 32
 b4 23 58 e6 7a 84 95 10 9a 5a d4 e1 7e d0 c7 53
 51 9f 99 ee 3e 06 7e 88 21 fb 12 29 47 dd 60 a9
TSS_Command_PreProcessor: Input parameters
        keyHandle TPM_HANDLE 80000002
        digest length 32
        b4 23 58 e6 7a 84 95 10 9a 5a d4 e1 7e d0 c7 53
        51 9f 99 ee 3e 06 7e 88 21 fb 12 29 47 dd 60 a9
        signature
        sigAlg TPM_ALG_DILITHIUM
        TPMU_SIGNATURE selection 002d not implemented
TSS_Execute20: Command 00000177 marshal
TSS_Execute_valist: Step 1: initialization
TSS_Execute_valist: Step 5: command encrypt
TSS_Sessions_GetDecryptSession: Found 0 decrypt sessions at 0
TSS_Execute_valist: Step 6 calculate HMACs
TSS_Execute_valist: Step 7 set command authorizations
TSS_Execute_valist: Step 8: process the command
TSS_AuthExecute: Executing TPM2_VerifySignature
TSS_Socket_Open: Opening localhost:2321-raw
TSS_Socket_SendCommand: TPM2_VerifySignature
 TSS_Socket_SendCommand length 2792

 80 01 00 00 0a e8 00 00 01 77 80 00 00 02 00 00
```

```
[...]
 5a d4 e1 7e d0 c7 53 51 9f 99 ee 3e 06 7e 88 21
 fb 12 29 47 dd 60 a9 03
 TSS_Socket_ReceiveCommand length 10
 80 01 00 00 00 0a 00 00 02 db
TSS_Socket_Close: Closing localhost-raw
verifysignature: failed, rc 000002db
```

As said before, this issue is of non-deterministic nature, so it affects the functionalities only during some of the runs, while overall the V-TPM functionalities are fine. We hope the developers of TSS will fix this issue soon.