D5.1

First version of 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 / D5.1/ DRAFT | 1.0
Work package contributing to the deliverable:	WP 5
Due date:	June 2019 – M18
Actual submission date:	1st July, 2019

Responsible organisation:	RHUL
Editor:	Daniele Sgandurra (RHUL)
Dissemination level:	PU
Revision:	1.0

Abstract:	This deliverable will report the status of the first version of SW-based QR TSS and QR TPM.
Keywords:	Software TPM, Implementation, TSS

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

Editor

Daniele Sgandurra (RHUL)

Christian Hanser (Infineon)

Contributors (ordered according to beneficiary numbers)

Luís Fiolhais, Paulo Martins and Leonel Sousa (INESC-ID)

Internal Reviewers

Christian Hanser (Infineon), Roberto Sassu (Huawei)

Executive Summary

The 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. The algorithm design will be accompanied with implementation and performance evaluation, namely hardware, software and virtualization environments. Use cases in online banking, activity tracking and device management will provide environments and applications to validate the FutureTPM framework. This deliverable reports the status of the design and implementation of the first version of the software-based quantum-resistant TPM, called *Software TPM*. In addition, we describe how the TPM Software Stack (TSS) has been updated to support the new QR cryptographic primitives. The deliverable also provides indication on how to extend both the software TPM and TSS with further algorithms.

Table of Contents

Chapter 1 Introduction 1
1.1 Scope and Purpose 1
1.2 Relation to Other WPs and Deliverables...3
1.3 Deliverable Structure 3
Chapter 2 Key Performance Indicators...4
Chapter 3 Software TPM Overview..6
3.1 Software TPM 6
3.2 Basic Operations of the Software TPM 6
3.2.1 New Endpoints 6
3.2.2 How to Add New algorithms 8
3.3 TPM Software Stack (TSS)...11
3.3.1 New Commands..12
3.3.2 How to Add New algorithms 14
3.4 Using the Software TPM 14
3.4.1 Example 15
3.5 Open Issues...18
Chapter 4 Conclusions...20
List of Abbreviations 21
Bibliography...22

List of Figures

Figure 1: Types of TPM according to TCG ("Trusted Platform Module 2.0 2
Figure 2: Overview of Software TPM 3
Figure 3: Communication Diagram between Alice and Bob 15
Figure 4: Software TPM Workflow of Commands 18

List of Tables

Table 1: Key Performance Indicators for Software TPM 5

Chapter 1 Introduction

In this Chapter we recall the scope and purpose of this deliverable, we describe its relationship with other work packages (WPs) and deliverables, and finally we outline the deliverable's structure.

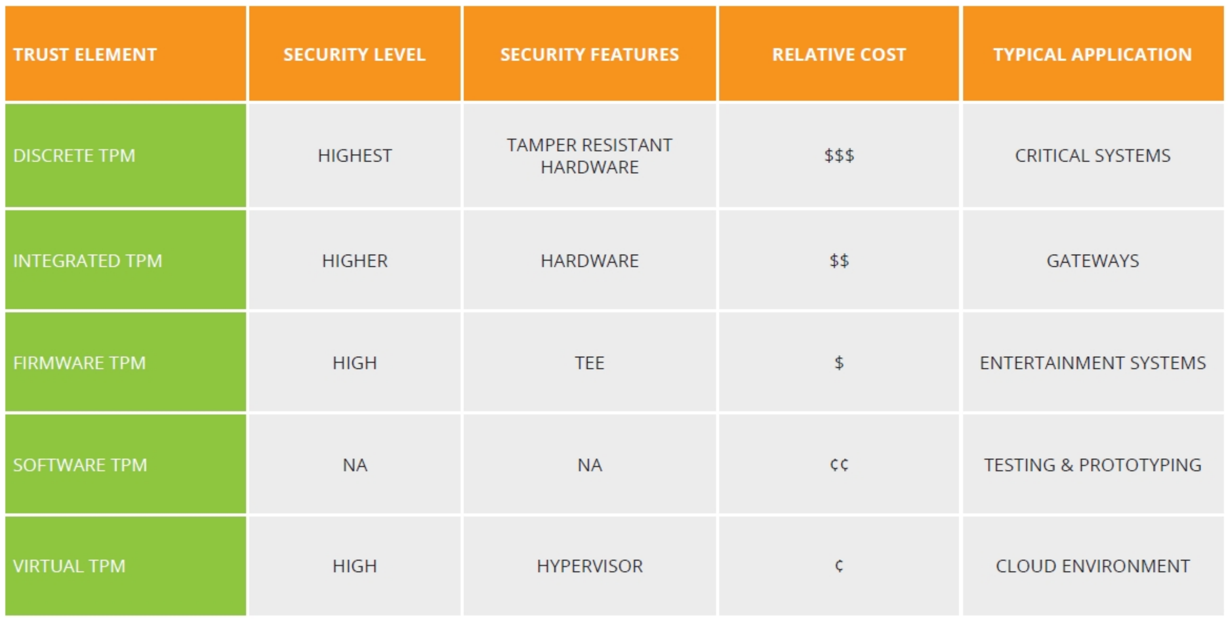
1.1 Scope and Purpose

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 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, 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 some form of 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 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. They rely on the hypervisor to provide them with an isolated execution environment that is hidden from the software running inside virtual machines to secure their code from the software in the virtual machines.

Figure 1 shows the security level, security features, relative cost and typical applications of the different types of TPMs.



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 ("Trusted Platform Module 2.0[1]

FutureTPM is investigating technologies for a new generation of TPM-based solutions, including hardware, software and virtualization environments, by incorporating robust and physically-secured 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 will describe the current implementation of the Software TPM. A high-level view of the current architecture of the Software TPM is illustrated in Figure 2. The components in yellow colour are inherited from the IBM Software TPM 2.0 implementation, the light blue components are the new components implementing the QR functionalities, and the green components shows those components that have been updated to also work in the new QR environments. The new and updated functionalities will be described in more detail in Chapter 3.

[1] A Brief Introduction", Available at: https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf)

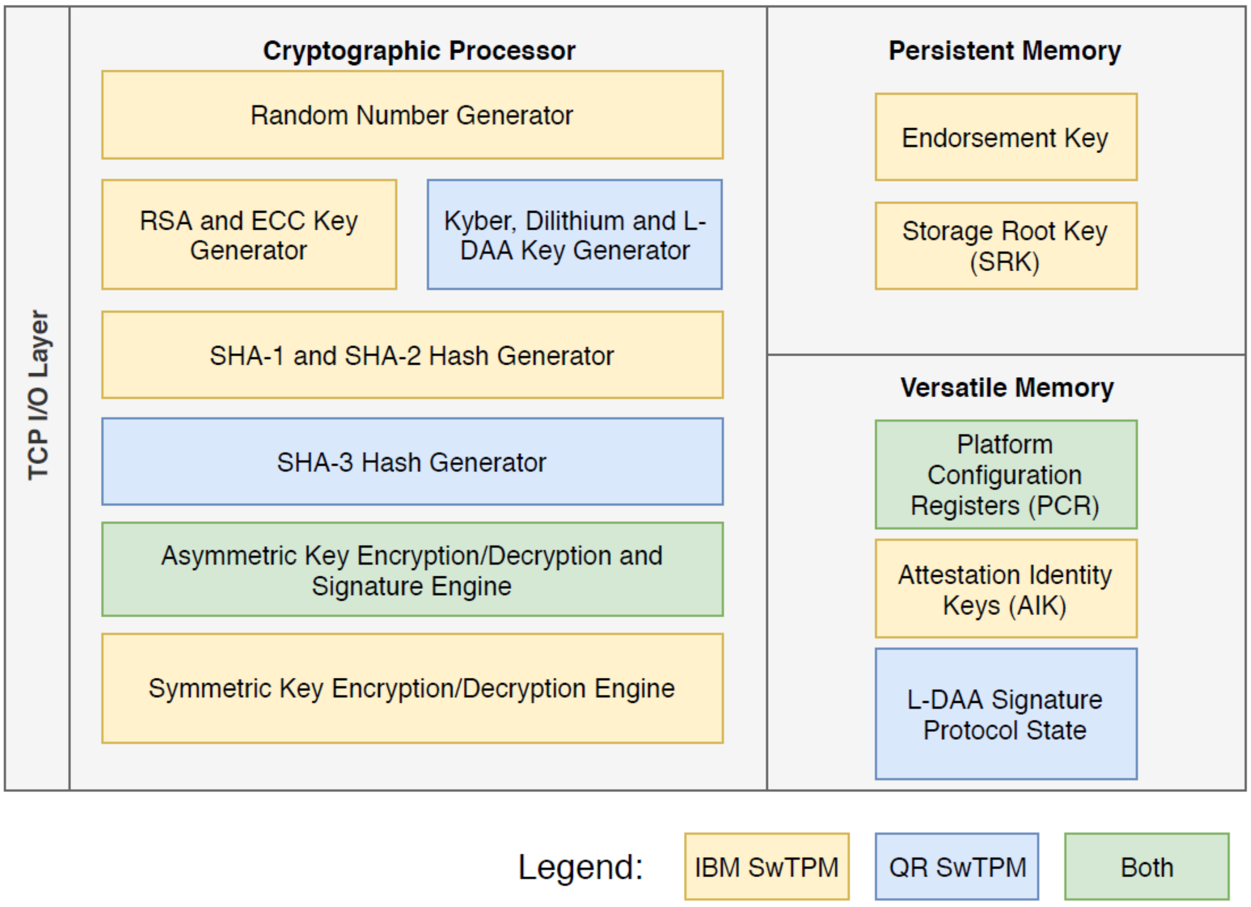

TCP I/O Layer
Cryptographic Processor
Random Number Generator
RSA and ECC Key Generator
Kyber, Dilithium and L-DAA Key Generator
SHA-1 and SHA-2 Hash Generator
SHA-3 Hash Generator
Asymmetric Key Encryption/Decryption and Signature Engine
Symmetric Key Encryption/Decryption Engine
Persistent Memory
Endorsement Key
Storage Root Key (SRK)
Versatile Memory
Platform Configuration Registers (PCR)
Attestation Identity Keys (AIK)
L-DAA Signature Protocol State
Legend:
IBM SwTPM
QR SwTPM
Both

Figure 2: Overview of Software TPM

1.2 Relation to Other WPs and Deliverables

Within WP5, Task 5.1 (Implementation and Evaluation of Software QR TPM) is devoted to the design, implementation and evaluation of the QR algorithms which were selected in WP2 in a software TPM emulator. This task is essential for developing the application layer of the TPM. The main output of this task is this deliverable (D5.1). In addition, the application layer code will be reused by the virtualized TPM (Task 5.2) and the hardware TPM (Task 5.3). This task is dependent on WP2, where the cryptographic algorithms have been defined. Furthermore, Task 5.4 deals with the development of a Trusted Software Stack (TSS) API that covers the newly introduced QR cryptographic algorithms. The outputs of these future activities will be also documented in D5.2, D5.3 and D5.4.

1.3 Deliverable Structure

This deliverable is structured as follows:

- **Chapter 2** provides the Key Performance Indicators (KPI) for the Software, Virtual and Hardware TPM.
- **Chapter 3** reports on the design of the first version of the quantum-resistant software-based Trusted Platform Module (TPM). We also provide additional pointers on how the software-based TPM may be used and extended with further algorithms. Moreover, we describe the changes to the TPM Software Stack (TSS) necessary to support the new cryptographic primitives, and we explain how the TSS may be further extended.
- **Chapter 4** concludes the deliverable.

Chapter 2 Key Performance Indicators

One of the metrics to measure the success of FutureTPM is the extent to which it can replace the current TPM standard, whilst providing quantum-resistance. While metrics such as memory requirements can be readily measured from cryptographic specifications, like those featured on NIST's post-quantum standardization effort, other metrics require concrete implementation. In the following, we identify the key performance indicators that can be specifically obtained from the Software, Virtual and Hardware TPMs. These metrics will enable us to perform an evaluation of: *i)* the impact quantum-resistance has on the execution time of cryptographic primitives; *ii)* the ability of the FutureTPM specification to remain secure upon the possibility of successful cryptanalysis of one of its primitives; *iii)* the ability to support different security levels according to the targeted application; and *iv)* the extensibility of implementations. There metrics are discussed in more detail in the following.

i) Performance metrics will measure the impact quantum-resistance has on key-exchange mechanisms, public-key encryption schemes, signature schemes, and direct anonymous attestation. For each of the following primitives, a TPM command, or combination of, which is representative of that operation, will be systematically chosen for each implementation and its execution time will be measured:

- Key exchange mechanism:
  - Key generation
  - Key encapsulation
  - Key decapsulation
- Public-key encryption schemes:
  - Key generation
  - Encryption
  - Decryption
- Signature schemes:
  - Key generation
  - Signature generation
  - Signature verification
- Direct anonymous attestation:
  - Opening
  - Key generation
  - Signature generation

Moreover, as far as possible, reference execution times will be given for each operation for their counterparts standardized in TPM 2.0.

ii) Since FutureTPM implementations will include recent cryptographic schemes, which have not yet undergone intensive scrutiny, there is a non-negligible risk of cryptanalysis developments towards some of them. A second metric will measure the number of different supported security assumptions for each of the following schemes:

- Key exchange mechanism
- Public-key encryption scheme
- Signature scheme
- Direct anonymous attestation

A qualitative evaluation of how the schemes supported on different security assumptions may share computational resources and thus reduce their combined implementation costs will also be provided. We define in Table 1 a target value for the aggregate number of supported schemes for each TPM implementation medium.

iii) As previously recalled, for each scheme, we will provide the number of supported cryptographic parameters, by measuring the adaptability of FutureTPM implementations to applications with different security and performance requirements. We expect all TPM implementations to support multiple parameters, as described in Table 1.

iv) A qualitative evaluation of the level of extensibility of implementations will be given, describing, for instance, the coverage level of regression tests, to ensure that future functionalities will not break currently implemented security primitives. All the developed systems will be made adequately extensible, as foreseen in Table 1.

KPIs/ Acceptance Criteria	**Software-based TPM**	**Hardware-based TPM**	**Virtual-based TPM**
Functionalities (number of algorithms implemented/supported)	*>=5*	*>= 3*[2]	*>= 5*[3]
Support for cryptographic parameters (*algorithm-agile parameters*)	*Yes*	*Yes*[4]	*Yes*[5]
Testing (usability, test cases, regression testing)	*Yes*	*Yes*	*Yes*

Table 1: Key Performance Indicators for Software TPM

[2] BLISS, NewHope and qTesla.
[3] The Virtual TPM will use the Software TPM algorithms with the added support for virtual environments.
[4] Hardware PM will implement NewHope 512 and qTesla2048.
[5] The Virtual TPM will use the Software TPM algorithms with the added support for virtual environments.

Chapter 3 Software TPM Overview

This chapter reports on the design of the first version of the quantum-resistant software-based Trusted Platform Module (TPM), Software TPM, giving pointers on how it may be used and extended with further algorithms. Moreover, the changes to the TPM Software Stack (TSS) necessary to support the new cryptographic primitives are described, and it is explained how the TSS may be further extended in the future.

3.1 Software TPM

The herein described Software TPM is a fork of the open-source implementation of TPM 2.0 produced by IBM and Microsoft with added support for new quantum resistant (QR) algorithms and hashes, namely Kyber (Bos, et al., 2018), Dilithium (Ducas, et al., 2019), Lattice-based Direct Anonymous Attestation (LDAA) (El Kassem, et al., 2019), and SHA3 (NIST, 2015).

3.2 Basic Operations of the Software TPM

The communication with the Software TPM requires the usage of sockets to establish a TCP connection between a client and the server. Through this connection, the Software TPM mimics the physical TPM command transmission interface (TCTI) layer found in the physical TPM.

After receiving data, the Software TPM 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 connection.

3.2.1 New Endpoints

To support the new QR algorithms, the following endpoints were added to the Software TPM:

- **Kyber**:
    - Key Generation: to generate a Kyber key one must use the standardized TPM2_Create, TPM2_CreateLoaded and TPM2_CreatePrimary functions, already provided by the TPM2 specification, with the TPM_ALG_KYBER value and the corresponding security mode *k*.
    - TPM2_KYBER_Enc encapsulates a shared secret, with the following parameters:
        - Input: *key_handle* for a loaded key handle to a Kyber public key.
        - Output: a *shared secret* and a *ciphertext* generated by the TPM.
    - TPM2_KYBER_Dec decapsulates a shared secret, with the following parameters:
        - Input: *key_handle* must reference a private loaded Kyber key and a *ciphertext* generated by TPM2_KYBER_Enc.
        - Output: the TPM will decapsulate the cipher object to obtain the *shared_secret*.
    - TPM2_KYBER_Encrypt encrypts a user-provided plaintext, with the following parameters:
        - Input: *key_handle* for a loaded key handle to a Kyber 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.

- TPM2_KYBER_Decrypt decrypts a ciphertext generated by TPM2_KYBER_Encrypt, with the following parameters:
  - Input: *key_handle* must reference a loaded private Kyber key and a *ciphertext* generated by TPM2_KYBER_Encrypt.
  - Output: the TPM will decrypt the cipher object to obtain the original plaintext *message*.

- **Dilithium**:
  - Key Generation: to generate a Dilithium key one must use the standardized TPM2_Create, TPM2_CreateLoaded and TPM2_CreatePrimary functions, already provided by the TPM2 specification, with the TPM_ALG_DILITHIUM value and the corresponding security mode *mode*.
  - Signing: to sign a digest using the modified SW QR TPM the user must use the following functions: TPM2_Sign and TPM2_Quote.
  - Signature Verification: to verify a Dilithium signature the user must use the TPM2_VerifySignature function.
- **LDAA**:
  - The variable notation used to describe the LDAA inputs and outputs is the same as in (El Kassem, et al., 2019).
  - Key Generation: to generate an LDAA key one must use the standardized TPM2_Create, TPM2_CreateLoaded and TPM2_CreatePrimary functions, already provided by the TPM2 specification, with the TPM_ALG_LDAA value and the corresponding security parameter *mode*.
  - TPM2_LDAA_Join performs a join request, with the following parameters:
    - Input: a reference to a loaded LDAA private and public entity (*key_handle*), the session ID (*sid*), the unique submission identifier (*jsid*), and the Issuer's nonce (*nonce*) and basename (bsn_I).
    - Output: the session link token (*nym*) and the proof of knowledge (π).
  - TPM2_LDAA_SignProceed allows the TPM to proceed with the calculation of the commitments, with the following parameters:
    - Input: a reference to a loaded LDAA private entity (*key_handle*) and the session identifier (*sid*).
  - TPM2_LDAA_CommitTokenLink initiates the LDAA sign commit procedure, with the following parameters:
    - Input: a reference to a loaded LDAA private entity (*key_handle*), the session identifier (*sid*), and the Verifier's basename (*bsn*).
    - Output: the commit token link (*nym*), the error polynomial (p_e), and the Verifier's basename polynomial (p_{bsn}).
  - The three functions TPM2_SignCommit [1,2,3] process the commits for the session, with the following parameters:
    - Input: a reference to a loaded LDAA private entity (*key_handle*), the session identifier (*sid*), the unique subsession identifier (*ssid*), the basename (*bsn*), the selector for the sign states (*sign_state*), the error polynomial and the basename polynomial generated in TPM2_LDAA_CommitTokenLink (p_e, p_{bsn}), and the seed to generate the shared polynomial matrix in the NTT domain between the Host and the TPM (*seed*). Additionally, the TPM2_SignCommit1 function requires the transposed Issuer's Polynomial Matrix *issuer_at_ntt*.

The reasoning behind re-generating the polynomial matrix shared between the TPM and the Host is that the matrix is too big to be transferred between the Host and TPM for each sign commit. Using maximum security parameters, the size of the matrix is in the order of GBs. As such, it was selected to generate smaller portions of the matrix inside the TPM only when necessary through a fixed seed value provided by the Host.

    - Output: the session identifier (*sid*), the unique subsession identifier (*ssid*), and the resulting commitment *commit*.

  - TPM2_LDAA_SignProof replies to the challenges issued by the Host, with the following parameters:
    - Input: a reference to a loaded LDAA private entity (*key_handle*), the session identifier (*sid*), the selector for the sign states (*sign_state_sel*), the generated challenge by the host (*sign_state_type*), the signature states generated in the host (*R1, R2*).
    - Output: the resulting sign states (*R1, R2*), and part of the secret values generated by the TPM2_LDAA_SignCommit commands (*sign_group*).

- **SHA3/SHAKE:**
  - The addition of the new SHA3s and SHAKEs hashes did not result in new endpoints. Their addition extended the following default endpoints: TPM2_Certify, TPM2_Create, TPM2_CreatePrimary, TPM2_Hash, TPM2_LoadExternal, TPM2_Sign, TPM2_VerifySignature, TPM2_EventSequenceComplete, TPM2_GetCommandAuditDigest, TPM2_GetSessionAuditDigest, TPM2_GetTime, TPM2_HashSequenceStart, TPM2_HMAC, TPM2_HMAC_Start, TPM2_LoadExternal, TPM2_Import, TPM2_NV_Certify, TPM2_PCR_Allocate, TPM2_PCR_Event, TPM2_PCR_Extend, TPM2_PCR_Read, TPM2_PolicyPCR, TPM2_PolicySigned, TPM2_Quote, and TPM2_StartAuthSession.

3.2.2 How to Add New algorithms

The above-described software TPM may be extended in the future to support new cryptographic primitives, potentiating the diversification of security assumptions in a post-quantum world. The addition of a new algorithm to the Software TPM is a multi-step process with wide modifications to the codebase. Therefore, this subsection focuses solely on describing the addition of asymmetric algorithms used in encryption/decryption, key encapsulation/decapsulation, and signing and signature verification, since these are the most relevant primitives to achieve quantum-resistance. Furthermore, for simplicity’s sake, this subsection defines a single example algorithm “Empire” which can perform all the operations.

The first step to add an algorithm to the Software TPM is to define it in *Implementation.h* and attribute an ID to it:

```
#define ALG_EMPIRE_VALUE          0x002F
#define TPM_ALG_EMPIRE           (TPM_ALG_ID)(ALG_EMPIRE_VALUE)
```

The ID chosen for this example was selected as the smallest unused ID of the TCG registry (TCG, 2018) Since the Empire algorithm supports encapsulation, decapsulation, encryption and decryption, these operations need to be also defined and associated with their respective IDs.

```
typedef TPM2B_KYBER_CIPHER_TEXT TPM2B_EMPIRE_CIPHER_TEXT;
typedef TPM2B_KYBER_ENCRYPT TPM2B_EMPIRE_ENCRYPT;
typedef TPM2B_KYBER_PUBLIC_KEY TPM2B_EMPIRE_PUBLIC_KEY;
typedef TPM2B_KYBER_SECRET_KEY TPM2B_EMPIRE_SECRET_KEY;
typedef TPM2B_KYBER_SHARED_KEY TPM2B_EMPIRE_SHARED_KEY;
typedef TPM2B_DILITHIUM_SIGNED_MESSAGE TPM2B_EMPIRE_SIGNED_MESSAGE;
typedef TPMS_SIG_SCHEME_DILITHIUM TPMS_SIG_SCHEME_EMPIRE;
typedef TPMT_DILITHIUM_SCHEME TPMT_EMPIRE_SCHEME;
typedef TPMS_SIGNATURE_DILITHIUM TPMS_SIGNATURE_EMPIRE;
typedef TPMS_DILITHIUM_PARMS TPMS_EMPIRE_PARMS;
```

Again, the IDs chosen for each operation and type were selected as the smallest unused IDs. Finally, the *TPM_CC_LAST* variable is updated to reflect the addition of the new commands. In this case, its new definition is 0x1A8. The new command entries are also added to the *LIBRARY_COMMAND_ARRAY_SIZE* definition.

The *s_algorithms* array in *AlgorithmCap.c* is updated so that the TPM2_GetCapability returns the correct information about the supported algorithms in the TPM.

CryptCreateObject is the global function where any key creation is performed. To create new Empire keys *CryptCreateObject* in *CryptUtil.c* is updated with a new case entry for the Empire algorithm that calls the concrete Empire key creation function. Then, in the same file, Empire entries are added to the *CryptStartup*, *CryptInit*, *CryptSecretEncrypt, CryptSecretDecrypt*, *CryptIsAsymAlgorithm*, *CryptIsAsymSignScheme* and *CryptAsymDecryptScheme* functions. The *CryptStartup* and *CryptInit* functions can be safely ignored if the new algorithm does not perform any operation at startup or initialization of the TPM. For the signature portion, a new Empire case entry needs to be added to the *CryptSign* and *CryptValidateSignature* functions. Similarly, to key creation, both functions are called when using the default TPM2_Sign and TPM2_VerifySignature commands, respectively. Any private code which is added in this step must be exported to *InternalRoutines.h*.

The new public data types for Empire reside in *TpmTypes.h.* Already defined datatypes may be used but it is strongly recommended that at least a different name be used in order to improve code readability. In Empire's case the new definitions are:

The respective new types are added to the following unions: *TPMU_SIG_SCHEME, TPMU_ASYM_SCHEME, TPMU_SIGNATURE, TPMU_PUBLIC_ID, TPMU_PUBLIC_PARMS, TPMU_SENSITIVE_COMPOSITE,* and *TPMU_ENCRYPTED_SECRET*. Then, union marshalling and unmarshalling functions are created and/or updated in *Marshal.c* and *Unmarshal.c*. If Empire only supported signing and verification, this would be the final step.

The new Empire encapsulation, decapsulation, encryption and decryption commands which were defined earlier can now be implemented. First the interfaces are defined for each command, e.g., for the Empire_Encrypt command the input, output, and its function interfaces can be defined in the file *Empire_Encrypt_fp.h* as:

```
typedef struct {
   TPMI_DH_OBJECT keyHandle;
   TPM2B_MAX_BUFFER      message;
} Empire_Encrypt_In;

typedef struct {
   TPM2B_EMPIRE_ENCRYPT          outData;
} Empire_Encrypt_Out;

TPM_RC
TPM2_Empire_Encrypt(
                    Empire_Encrypt_In     *in, // IN: input parameter list
                    Empire_Encrypt_Out    *out // OUT: output parameter list
                    );
```

All command definitions must be added to *Commands.h*. Then, the interfaces are exported to *CommandDispatchData.h*. Following the previous example one would implement the Empire Encrypt command descriptor as:

```
 #include "Empire_Encrypt_fp.h"
typedef TPM_RC  (Empire_Encrypt_Entry)(
                                        Empire_Encrypt_In  *in,
                                        Empire_Encrypt_Out *out
                                        );
typedef const struct {
   Empire_Encrypt_Entry     *entry;
   UINT32            inSize;
   UINT32            outSize;
   UINT32            offsetOfTypes;
   UINT32            paramOffsets[1];
   BYTE              types[5];
} Empire_Encrypt_COMMAND_DESCRIPTOR_t;
Empire_Encrypt_COMMAND_DESCRIPTOR_t _Empire_EncryptData = {
   /* entry  */        &TPM2_Empire_Encrypt,
   /* inSize */        (UINT32)(sizeof(Empire_Encrypt_In)),
   /* outSize */       (UINT32)(sizeof(Empire_Encrypt_Out)),
   /* offsetOfTypes */   offsetof(Empire_Encrypt_COMMAND_DESCRIPTOR_t, types),
   /* offsets */       {(UINT32)(offsetof(Empire_Encrypt_In, message))},
   /* types */         {TPMI_DH_OBJECT_H_UNMARSHAL,
                                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
           END_OF_LIST,
           TPM2B_EMPIRE_ENCRYPT_P_MARSHAL,
           END_OF_LIST}
};
#define _EMPIRE_EncryptDataAddress (&_Empire_EncryptData)
```

If needed, the missing marshalling and unmarshalling functions are defined in *MarshalArray* and *UnmarshalArray* in the same file, and the command descriptors are added to *s_CommandDataArray*.

At this stage the implementation for each Empire command can be defined in *AsymmetricCommands.c* (TPM2_Empire_Encrypt, etc). Finally, the *s_ccAttr* array in *CommandAttributeData.h* can be updated with the new Empire commands. The Empire algorithm is now fully implemented.

3.3 TPM Software Stack (TSS)

The herein described TSS is a fork of the open-source implementation of TSS 2.0 by IBM and Microsoft with added support for new QR algorithms and hashes, namely Kyber (Bos, et al., 2018), Dilithium (Ducas, et al., 2019), LDAA (El Kassem, et al., 2019), qTesla (Bindel, et al., 2019), NewHope (Alkim, Ducas, Poppelmann, & Schwabe, 2016), and SHA3 (NIST, 2015).

The new QR-TSS is not backwards-compatible with other implementations of the Software TPM and has only been tested with the previously described Software TPM. Furthermore, the previously described Software TPM does not possess endpoints for the qTesla and NewHope algorithms.

This section describes the new commands and how to further extend the TSS with new algorithms.

3.3.1 New Commands

New commands were added to the TSS in order to support the new endpoints in the Software TPM:

- **Kyber**
  - create / createprimary => default commands to create keys inside the TPM. To create a Kyber key the user can execute them with the *-kyber* flag with a specific security mode *k*;
  - loadexternal => default command to load a key which may or may not have been created by the TPM. To load a Kyber key the *-kyber* flag must be used;
  - kyber_enc => a Kyber-only command where an encapsulation is performed using the selected loaded key. An example of its usage is:

    ./kyber_enc -hk LOADED_KEY_HANDLE -c OUT_CIPHER -ss OUT_SHARED_SEC
  - kyber_dec => a Kyber-only command where a decapsulation is performed using the selected loaded key and a ciphertext previously generated by the kyber_enc command. An example of its usage is:

    ./kyber_dec -hk LOADED_KEY_HANDLE -c IN_CIPHER -ss OUT_SHARED_SEC
  - kyberencrypt => a Kyber-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:

    ./kyberencrypt -hk LOADED_KEY_HANDLE -id IN_PLAIN -oe OUT_CIPHER
  - kyberdecrypt => a Kyber-only command where a decryption is performed using the selected loaded key and an input is received to be encrypted. Its usage is:

    ./kyberdecrypt -hk LOADED_KEY_HANDLE -ie IN_CIPHER -od OUT_PLAIN
- **Dilithium**
  - create / createprimary => default commands to create keys inside the TPM. To create a Dilithium key the user can execute them with the *-dilithium* flag with a specific security mode;
  - loadexternal => default command to load a key which may or may not have been created by the TPM. To load a Dilithium key use the *-dilithium* flag;
  - sign => default command to sign and hash a file inside the TPM. To sign the message using the Dilithium algorithm the *-dilithium* flag must be used;
  - verifysignature => default command to verify a signature inside the TPM. To verify a Dilithium signature the *-dilithium* flag must be used.
- **LDAA**
  - create / createprimary => default commands to create keys inside the TPM. To create an LDAA key the user can execute them with the *-ldaa* flag with a specific security mode *lmode*;

- ldaa_join => LDAA specific command to start an LDAA session inside the TPM by performing the join procedure for a new session ID, a new unique session ID and the Issuer's basename. Its usage is:

  ./ldaa_join -hk LOADED_KEY_HANDLE -sid IN_NEW_SESSION_ID -jsid IN_UNIQUE_SESSION_ID -bsn IN_ISSUER_BASENAME -onym OUT_JOIN_LINK_TOKEN

- ldaa_signproceed => LDAA specific command to be used by the Host to allow the TPM to start the signing procedure. Its usage is:

  ./ldaa_signproceed -hk LOADED_KEY_HANDLE -sid IN_SESSION_ID

- ldaa_committokenlink => LDAA specific command to initiate the signing commit procedure for a specific session and Issuer's basename. Its usage is:

  ./ldaa_committokenlink -hk LOADED_KEY_HANDLE -sid IN_SESSION_ID -bsn IN_ISSUER_BASENAME -onym OUT_TOKEN_LINK -ope OUT_ERR_POLY -opbsn OUT_BSN_POLY

- ldaa_signcommit [1,2,3] => LDAA specific command to process the commit[1,2,3] inside the TPM for a respective session, a basename, the Issuer's basename polynomial, the error polynomial, a selected sign state, and the seed value to generate the shared B matrix in the NTT domain. The ldaa_signcommit1 command requires an additional parameter from the issuer: the transposed A matrix in the NTT domain. Their usages are:

  ./ldaa_signcommit1 -hk LOADED_KEY_HANDLE -sid IN_SESSION_ID -bsn IN_BASENAME -seed IN_SEED_VALUE_IN_HEX -sign [0-7] -iatntt IN_FILE_BIN -ipe IN_ERR_POLY -ipbsn IN_BSN_POLY -ocomm OUT_COMMIT

  /ldaa_signcommit[2,3] -hk LOADED_KEY_HANDLE -sid IN_SESSION_ID -bsn IN_BASENAME -seed IN_SEED_VALUE_HEX -sign [0-7] -ipe IN_ERR_POLY -ipbsn IN_BSN_POLY -ocomm OUT_COMMIT

- ldaa_signproof => LDAA specific command to reply to the challenges issued by the Host

  ./ldaa_signproof -hk LOAD_KEY_HANDLE -sid IN_SESSION_ID -signT IN_RESPONSE_TYPE -sign [0-7] -isign1 IN_1ST_HOST_SIGN_STATE -isign2 IN_2ND_HOST_SIGN_STATE -osign1 OUT_SIGN_RESULT1 -osign2 OUT_SIGN_RESULT_2 -ogroup OUT_PORTION_SECRET_VALUES

- **SHA3/SHAKE**
  - Similarly, to the hash additions in the Software TPM, the addition of SHA3 and SHAKE did not result in new commands in the TSS, but the default commands were updated to support the new hashing algorithms. These commands are: *certify, create, createprimary, eventsequencecomplete, getcommandauditdigest, getsessionauditdigest, hash, gettime, hashsequencestart, hmac, hmacstart, importpem, loadexternal, nvcertify, pcrallocate, pcrevent, pcrread, policypcr, policymaker, policysigned, quote, sign, startauthsession,* and *verifysignature.* In all command instances the new hash algorithms can be used through either the *-halg* or the *-nalg* flag. For example, the command to create a SHA3-512 of some data is:

    ./hash -halg sha3-512 -ic test_data

3.3.2 How to Add New algorithms

Adding algorithms to the TSS follows a procedure similar to the one described in Section 3.2.2, but applied to different files. The new TPM types are added to *ibmtss/TPM_Types.h*, the marshalling functions are defined in *tssmarshal.c*, the unmarshalling functions in *Unmarshal.c*, the algorithms in *ibmtss/Implementation.h*, the command attribute data is added to *CommandAttributeData.h*, command interfaces are exported to *Commands_fp.h* and to *ibmtss/Parameters.h*, and the command dispatch data is included in *tssauth20.c*. All command interfaces must be copied to the *ibmtss/* folder.

The existing commands can be updated to add support for Empire (*loadexternal*, *create*, *createprimary*, *sign* and *verifysignature*) and new commands can be created (*empire_enc*, *empire_dec*, *empireencrypt* and *empiredecrypt*).

Finally, it is strongly encouraged to add regression tests for the new algorithm in the *regtests/* folder, including key creation in *initkeys.sh*, signature creation and verification in *testsign.sh*, and more specific Empire tests in *testempire.sh*. The general regression test suite script, *reg.sh*, should also be updated with a new entry for Empire.

3.4 Using the Software TPM

Building the Software TPM requires a C compiler (e.g., gcc, clang) and the make tool. To build the Software TPM the user should navigate into its *src/* folder and run the *make* command. The compilation step should produce no warnings and a single binary called *tpm_server*. The user can now execute the server binary. By default, the *tpm_server* runs its command server on port 2321 and its platform server on 2322. The user may change the command and platform ports through the *-port* flag, where the selected port will be used for the command server and the selected port plus one will be used for the platform server, *e.g.*, the command:

```
$ ./tpm_server -port 4000
```

runs the Software TPM command server on port 4000 and the platform server on 4001. The user can obtain more information about the tool by running:

```
$ ./tpm_server -h
```

Similar to the Software TPM, the TSS requires a C compiler and the make tool. To build the TSS the user should navigate to the *utils/* folder and run the *make* command. The compilation step should produce no warnings and several binaries are created.

To assert the correct operation of the Software TPM the full regression test suite might be run. To do so, it is firstly advised to verify that the *tpm_server* is running in a separate terminal with the default command and platform ports. The regression test suite resides in the same folder as the TSS binaries and is called *reg.sh*. To execute the full test suite the following command should be run with:

```
$ ./reg.sh -a
```

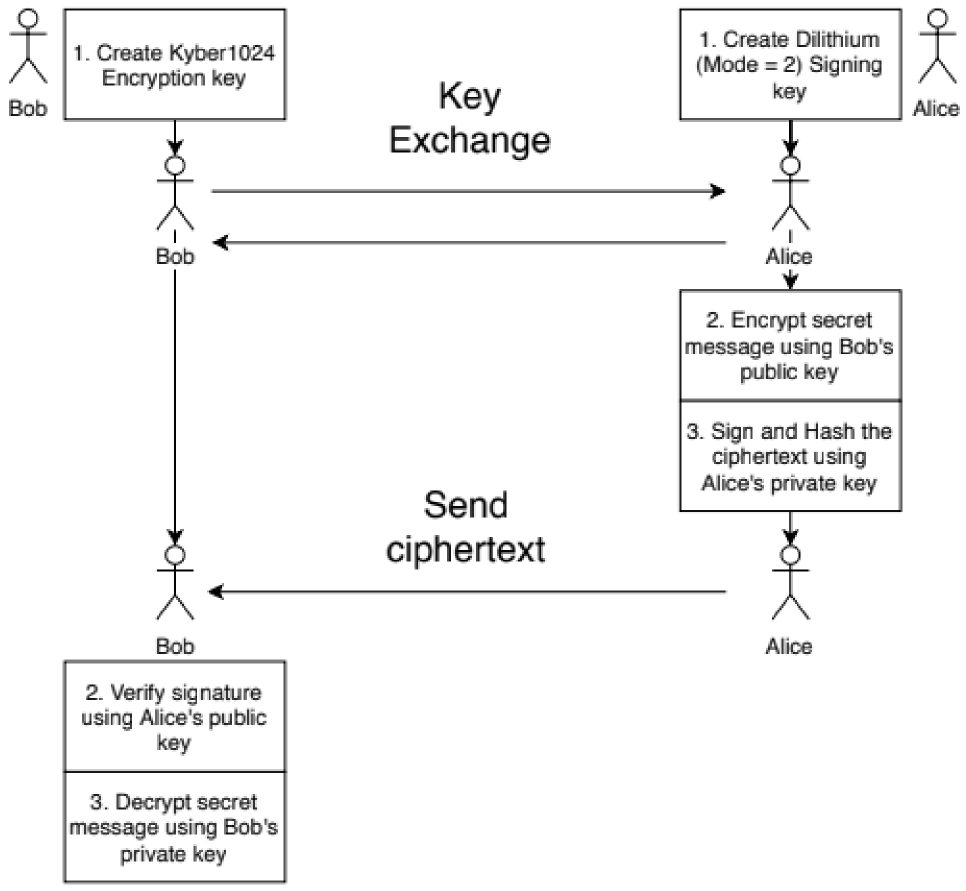
At the end of the tests the output should show:

```
Success - 33 Tests 0 Warnings
```

Running the test suite is extensive, thorough, and time-consuming. However, this is a recommended practice when performing modifications to the Software TPM.

3.4.1 Example

This subsection will walkthrough an example, where Alice communicates with Bob to transmit an encrypted and authenticated message using the TPM. Figure 3 depicts the communication diagram.



Bob
1. Create Kyber1024 Encryption key
Key Exchange
1. Create Dilithium (Mode = 2) Signing key
Alice
Bob
Alice
2. Encrypt secret message using Bob's public key
3. Sign and Hash the ciphertext using Alice's private key
Send ciphertext
Bob
Alice
2. Verify signature using Alice's public key
3. Decrypt secret message using Bob's private key

Figure 3: Communication Diagram between Alice and Bob

This scenario description assumes that the user has four shells opened, two for Bob and two for Alice. For each user, one shell is opened in the Software TPM *src/* folder and the other in the TSS *utils/* with all the binaries compiled. For the sake of clarity, the Software TPM shell shall be referred to as TPM-SH and the TSS shell as TSS-SH from this point onward. Note that to run two instances of the Software TPM in the same machine both must use different command and platform ports. For the sake of clarity Alice's TSS and Software TPM run on command port 2325 and on interface port 2326. To make use of these ports all of Alice's commands in the TSS are prefixed with TPM_COMMAND_PORT=2325 TPM_PLATFORM_PORT=2326.

```
$ ./tpm_server
```

Both Alice and Bob in their respective TPM-SHs run the server

```
$ ./startup -c
```

and then in their TSS-SHs, start up the TPM.

It should be noted that running this command is always necessary after having started the server.

Bob and Alice create a primary key each using Kyber784 (*k=3*) with a platform key hierarchy (-*hi*), a password *sto* (-*pwdk*), an output ticket file *pritk.bin* (-*tk*) and a creation hash file name *prich.bin* (-*ch*):

```
$ ./createprimary -kyber k=3 -hi p -pwdk sto -tk pritk.bin -ch prich.bin
Handle 80000000
```

The *createprimary* command returns a key handle under which keys can be created. The signing and encryption keys can now be created by each party. Alice uses the Dilithium algorithm for signatures and Bob uses Kyber1024 for encryption/decryption.

Alice, in TSS-SH, creates a signing (-*si*) Dilithium key under the Kyber784 primary key (-*hp 80000000* with password -*pwdp sto*) fixed to this TPM (-*kt f*) and to this parent key (-*kt p*), stores the public key in *dil_pub.bin* (-*opu*) and private key in *dil_priv.bin* (-*opr*) with the password *dilithium* (-*pwdk*)

```
$ ./create -hp 80000000 -si -dilithium mode=2 -kt f -kt p -opr dil_priv.bin -opu dil_pub.bin -pwdp sto
-pwdk dilithium
$ ./create -hp 80000000 -kyber k=4 -den -kt f -kt p -opr kyber_priv.bin -opu kyber_pub.bin -pwdp
sto -pwdk kyber
```

Bob, in TSS-SH, creates a Kyber1024 decryption (-*den*) key under the Kyber784 primary key with the same properties, and stores the public key in kyber*_pub.bin* (-*opu*) and private key in kyber*_priv.bin* (-*opr*) with the password *kyber* (-*pwdk*).

Bob and Alice then exchange their respective public keys through another authenticated channel.

The message which Alice will encrypt and sign is “My super secret. Please don’t share.” Alice will encrypt the message using Bob’s public Kyber1024 key, thus it needs to be loaded. Still in TSS-SH, Alice runs:

```
$ ./loadexternal -hi p -ipu kyber_pub.bin
Handle 80000001
```

and performs the encryption:

```
$ ./kyberencrypt -hk 80000001 -id test.txt -oe enc.bin
```

where -*id* is the file which contains the secret string and -*oe* is the file which contains the

```
$ ./flushcontext -ha 80000001
```

encryption result. Since Alice no longer needs Bob’s encryption public key, it is safe to flush it:

Alice then loads the Dilithium private and public keys under the Kyber784 primary key.

```
$ ./load -hp 80000000 -ipr dil_priv.bin -ipu dil_pub.bin -pwdp sto
Handle 80000001
```

Finally, the ciphertext (*-if enc.bin*) is signed and hashed using the loaded Dilithium key (*-hk 80000001 -dilithium*) and the signature is stored in *sig.bin* (*-os*):

```
$ ./sign -hk 80000001 -dilithium -if enc.bin -os sig.bin -pwdk dilithium
```

Bob receives the ciphertext and Alice's signature, but before proceeding must load its own Kyber1024 private and public key, as well as Alice's public key:

```
$ ./load -hp 80000000 -ipr kyber_priv.bin -ipu kyber_pub.bin -pwdp sto
Handle 80000001
$ ./loadexternal -hi p -ipu dil_pub.bin
Handle 80000002
```

Bob verifies Alice's signature in its TSS-SH:

```
$ ./verifysignature -hk 80000002 -dilithium -if enc.bin -is sig.bin
```

After successfully authenticating that the ciphertext was sent by Alice, Bob proceeds to decrypt Alice's message:

```
$ ./kyberdecrypt -hk 80000001 -ie enc.bin -od dec.bin -pwdk kyber
```

Bob can now flush the keys from the TPM:

```
$ ./flushcontext -ha 80000001
$ ./flushcontext -ha 80000002
```

Finally, Figure 4 shows the workflow of all the commands in the current implementation of the Software TPM.

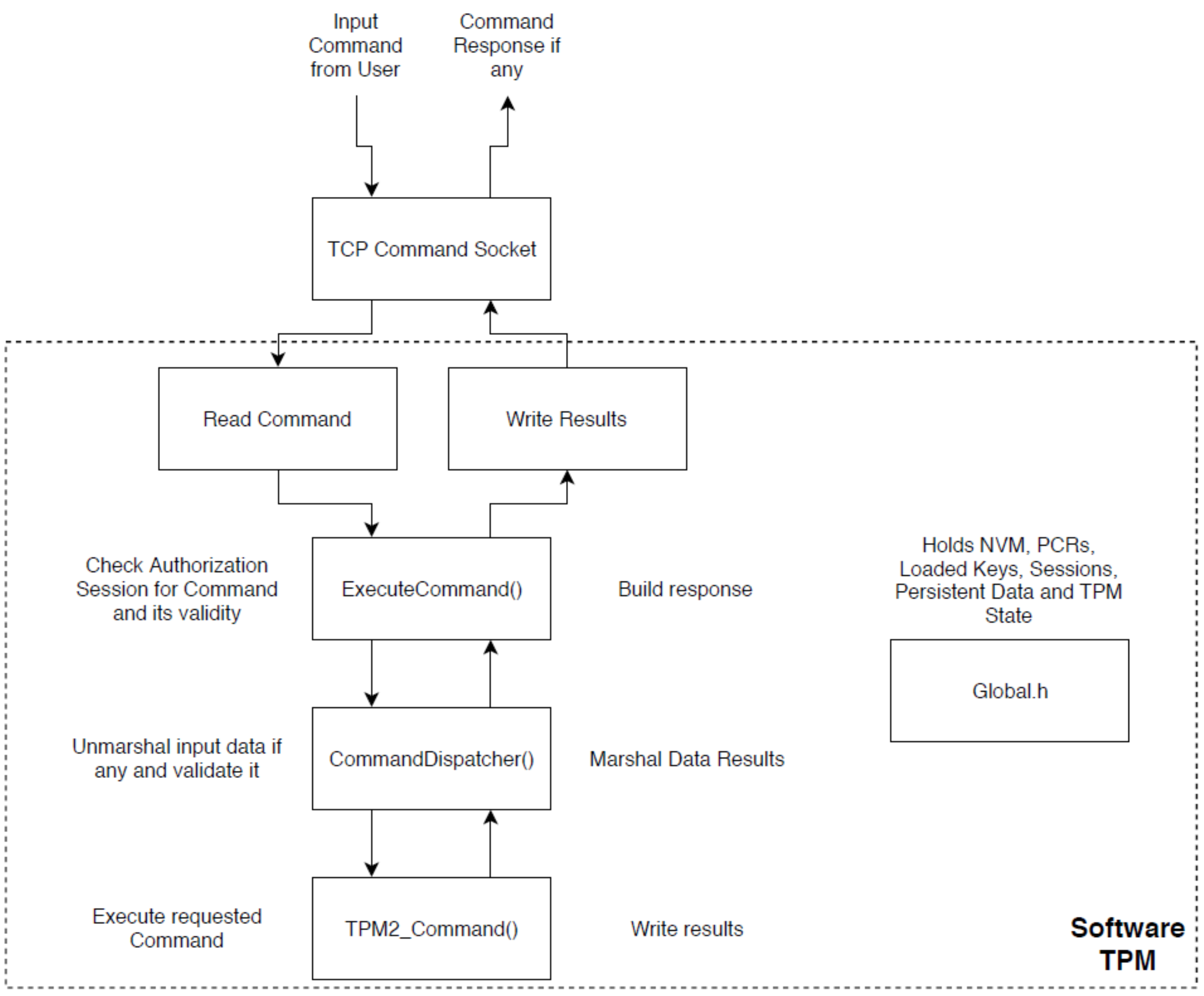


Figure 4: Software TPM Workflow of Commands

3.5 Open Issues

The following issues were identified during the development of the Software TPM and the TSS:

- When running the complete regression test suite using a Kyber key (of any mode) as a primary key the suite fails in the Salt tests with an integrity error on the Software TPM side. However, when running just the Salt tests with the same Kyber primary, no errors are reported.
- The addition of the LDAA algorithm required extensive modifications throughout both the Software TPM and the TSS. In doing so, the Non-Volatile Memory was increased from 64kB to 20MB. The regression test suite does not reflect this update.
- The LDAA only supports one session per TPM.
- Current Kyber implementation corresponds to the unmodified submission to the NIST PQC challenge (i.e., to round 1).
- When performing a hash using any of the SHAKE algorithms its output is limited to 1024B. Furthermore, even though there is a mechanism which allows a hash state to be continuously update from the client there is no endpoint to continue reading a hash state once it is finalized. Thus, neither of the SHAKEs possess the expected Application

Programming Interface (API) to interface with. This limitation is due to the Software TPM cryptography backend, OpenSSL, which does not provide such an interface.

Since the aforementioned minor issues do not present any significant impediment for the usage of the TPM in practice, and since these cryptographic primitives might change in the future, for instance to improve the performance of post-quantum DAA and to add a new endpoint which provides proper support for extendable output functions (XOF), we believe it would be more efficient to resolve them on a later, more consolidated, deliverable, *e.g.*, D5.3 “Final version of implementation”.

Chapter 4 Conclusions

In this deliverable we have provided an overview of the design and implementation of the Quantum-Resistant Software Trusted Platform Module. We have described the key performance indicators on which to measures the three different implementations of the TPM (hardware, software and virtual), and we have described in detail the current implementation of the software TPM, in particular the basic operations and how to add new algorithms. In addition, we have described how the TPM Software Stack (TSS) has been updated to support the new QR cryptographic primitives, and we have provided detailed indication on how to extend both the software TPM and TSS with further algorithms. The next steps involve the implementation of the QR virtual TPM (Task 5.2) and hardware TPM (task 5.3), which will build upon the current implementation of the QR software TPM.

List of Abbreviations

Abbreviation	Translation
API	Application Programming Interface
DAA	Direct Anonymous Attestation
LDAA	Lattice-based Direct Anonymous Attestation
QR	Quantum Resistant
SW-TPM	Software TPM
TPM	Trusted Platform Module
TSS	TPM Software Stack
V-TPM	Virtual TPM
WP	Work Package
XOF	Extendable Output Functions

Bibliography

Alkim, E., Ducas, L., Poppelmann, T., & Schwabe, P. (2016). Post-quantum Key Exchange - A New Hope. *USENIX Security Symposium.* Vancouver, Canada.

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

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., M. Schanck, J., . . . Stehlé, D. (2018). CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. *2018 IEEE European Symposium on Security and Privacy, EuroS&P.* London, UK.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., & Stehlé, D. (2019, 03 30). *Dilithium - Submission to the NIST post-quantum project.* Retrieved from NIST: https://pq-crystals.org/dilithium/data/dilithium-specification-round2.pdf

El Kassem, N., Chen, L., El Bansarkhani, R., El Kaafarani, A., Camenisch, J., Hough, P., . . . Sousa, L. (2019). More efficient, provably-secure direct anonymous attestation from lattices. *Future Generation Computer Systems, Volume 99*, 425-258.

NIST. (2015, 4 8). *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* . Retrieved from NIST: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

TCG. (2018, 2 7). *TCG Algorithm Registry*. Retrieved from TCG: https://trustedcomputinggroup.org/wp-content/uploads/TCG-_Algorithm_Registry_Rev_1.27_FinalPublication.pdf