Software Emulation of Quantum Resistant Trusted Platform Modules

Luís Fiolhais, Paulo Martins, Leonel Sousa
luis.azenhas.fiolhais@tecnico.ulisboa.pt, paulo.sergio@netcabo.pt and las@inesc-id.pt

.

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Keywords: Public-key cryptography, Symmetric-key cryptography, Post-quantum cryptography, Direct Anonymous Attestation, Trust Platform Module

Abstract: Trusted Platform Modules (TPMs) serve as the root of trust to design and implement secure systems. Conceived by the Trusted Computing Group, a computer industry consortium, components complying with the TPM 2.0 standard are stable and widely available. However, should large-scale quantum computing become a reality, the type of cryptographic primitives adopted in the current standard will no longer be secure. For this reason, this paper analyses the impact of adding three Post-Quantum (PQ) algorithms to a current non-Quantum Resistant TPM through software emulation. The experimental results give insight on the kind of implementation challenges hardware designers will face when integrating the new primitives onto the TPM, that typically features limited hardware resources and low power consumption. In particular, it is concluded that Kyber, NTTRU, and Dilithium can efficiently replace most of the functionality provided by Elliptic Curve Cryptography (ECC) and Rivest-Shamir-Adleman (RSA). In contrast, current PQ Direct Anonymous Attestation (DAA) protocols are currently not compact enough to fit into a hardware TPM.

1 INTRODUCTION

The Trusted Platform Module (TPM) is a module that builds a representation of the state of the Host machine as it boots (Arthur and Challener, 2015). Concretely, as software is loaded a hash of it is concatenated with a hash stored in a TPM Platform Configuration Register (PCR), and the result is itself hashed and then stored back in the PCR. In this manner, the TPM can be used as the root of trust for a computing platform. For instance, the hard-drive might be encrypted with a key that is only made available when the Host has booted to a trustworthy state. It also features many cryptographic functionalities and Non-Volatile (NV) storage that extend its usage to many other applications. As an example, since its NV memory is available to the Basic Input/Output System (BIOS), it might hold certificates used to ensure that only trustworthy images are used to boot the system. Furthermore, the TPM standard defines a protocol, called Direct Anonymous Attestation (DAA), that allows for a TPM to authenticate itself as a genuine module without disclosing its identity (Brickell et al., 2004).

TPMs may be implemented under several forms (Group, 2016). Hardware discrete and integrated TPMs ensure security against software attacks, with discrete TPMs offering stronger assurances against physical attacks than integrated TPMs, as they are implemented in a separate chip (Group, 2016; Microsoft, 2018). Software TPMs are typically used for development and prototyping, since they enable faster testing, while not reducing the lifetime of NV memories due to a high number of writes (Group, 2016). Beyond that, software TPMs may be used as a basis for the development of firmware TPMs, that reside on a Central Processing Unit (CPU)'s trusted execution environment providing hardware TPM-like functionality (Group, 2016; Microsoft, 2018), or virtual TPMs, which hypervisors emulate for their guests' virtual machines (Group, 2016).

The current TPM standard relies on number-theoretic cryptography. However, should quantum computers become available, this type of cryptography will no longer be secure (Shor, 1994). While there are several branches of Post-Quantum (PQ) cryptography (Bernstein et al., 2008), the focus herein is on the application of lattice-based cryptography to the TPM. Lattice-based cryptography seems to be the only PQ branch flexible enough to support all the functionalities required by the TPM (Bos et al., 2017; Kassem et al., 2019; Ducas et al., 2019; Lyubashevsky and Seiler, 2019). Moreover, by focusing on operations over lattices, one is promoting the reuse of

cryptographic accelerators, and reducing the cost of a TPM. These changes are herein applied to a software TPM, and an analysis of the impact PQ resistance would have on the design of hardware TPMs is provided. Finally, this paper presents application case studies using the Quantum Resistant (QR) TPM and identifies future research that would enable the design of QR TPM in a more practical way.

2 RELATION TO PRIOR WORK

Most proposals for PQ cryptography fall under four categories: hash, multivariate, code or lattice-based cryptography. Of these, hash and multivariate-based cryptography only support signatures (Bernstein et al., 2008). The use of hash-based signatures has been considered in initial developments towards a PQ TPM (Fuchs, 2018). Since a TPM inherently requires hashing for PCR extensions, hash-based signing might take advantage of accelerators devoted to PCR extensions. However, the resulting signatures are significantly larger than code or lattice-based signatures (Aumasson et al., 2019; Ducas et al., 2019). As the TPM also requires public-key encryption, developing primitives around these latter types of cryptography might enable the same level of system sharing, leading to more efficient signing for the same overall hardware cost. Moreover, (Fuchs, 2018) did not support PQ DAA, since it further relied on code-based cryptography. In contrast, the herein proposed construction is based on lattice-based primitives, that support signatures, public-key encryption and DAA (Ducas et al., 2019; Bos et al., 2018; Kassem et al., 2019; Lyubashevsky and Seiler, 2019).

3 TPM EMULATOR

The open-source implementations provided by IBM and Microsoft of the Software TPM (SW-TPM) (IBM et al., a) and the TPM Software Stack (TSS) (IBM et al., b) were used as a basis for the design of the proposed TPM.

Fig. 1 shows the basic architecture of the SW-TPM as provided by current open-source implementations, where the Transmission Control Protocol (TCP) interface emulates the TPM Command Transmission Interface (TCTI) layer found in the physical TPM. The base architecture is composed of: a cryptographic processor wherein a secure Random Number Generator (RNG), Rivest-Shamir-Adleman (RSA) and Elliptic-curve Cryptography (ECC) cryptographic primitives, and a hashing engine are available; a small persistent memory module (64kB) to store TPM's state; and a versatile memory to keep short-lived data. The SW-TPM can be configured with different math, hash, and symmetric backend engines. The configuration described herein uses the OpenSSL engine.

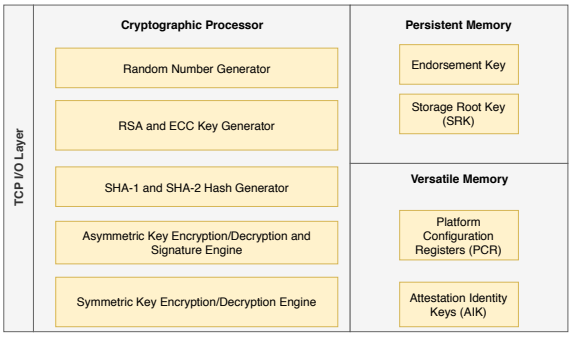

Figure 1: Original SW-TPM Architecture.

With the exception of the modules that rely on OpenSSL, the emulator makes no use of the heap, storing the versatile and persistent memory in the `bss` program segment. This is done to approximate the emulation of the TPM to its physical counterpart. This also facilitates porting the code to HW-SW co-designs, wherein controlling logic is executed on the main processor, and cryptographic operations are offloaded to Domain Specific Accelerators (DSAs).

Through the emulated TCTI layer, a client is able to interface with a SW-TPM using the commands provided in the TSS. Fig. 2 shows a callgraph of the chain of functions executed when a command is received. Once the SW-TPM receives a command it will deserialize the request and validate it, check the command's authorization if any, deserialize the data and validate its contents, and then issue the command to its relevant endpoint. After the completion of the command, the SW-TPM builds a response to the client by performing the same operations in reverse order and serializing the response data where needed.

4 QR ALGORITHMS IMPLEMENTATION IN SW-TPM

Four new QR algorithms and a new hashing algorithm were added to the base implementation described in the previous section: Kyber and NTTRU, for asymmetric key-exchange and encryption/decryption (Bos et al., 2018; Lyubashevsky and Seiler, 2019); Dilithium, for data signatures (Ducas et al., 2019); Lattice-Based Direct Anonymous Attestation (L-DAA) for anonymous attestation (Kassem

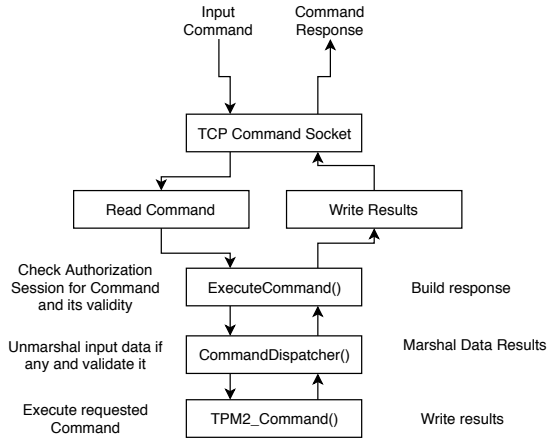

Figure 2: Function flowgraph executed by the SW-TPM when a valid command is received.

et al., 2019); and SHA3 (NIST, 2015). All implementations described in this section were designed so that they can be adopted and implemented in the current TPM's architecture. The implementation follows the guidelines reported in (FutureTPM, 2019).

Since both Kyber and Dilithium use SHA3 as Key Derivation Functions (KDFs), this new hash scheme was firstly implemented. Hashes are important across all cryptographic schemes. The most common usage of hashing schemes is in the PCR module. PCRs primarily hold data about the machine's boot state, in an hashed format. As such, they were expanded with the new SHA3 variants. Besides the standard cryptographic functions, the SHA3 standard also provides Extendable Output Functions (XOF) labeled as SHAKE; SHAKEs are cryptographic hash functions able to output an arbitrary number of pseudo-random bytes. Since the main goal is to emulate a future hardware TPM, there needs to be a limit imposed on the size of the XOF output, commensurate with hardware implementations. Kyber and Dilithium make use on average of 547B from an XOF (Gueron and Schlieker, 2016). The TPM uses a buffer equal to the maximum size of all supported hash algorithms, where the largest hash output by the standard commands is 64B from SHA512 (or SHA3-512). The standard TPM architecture supports a maximum buffer size of 2kB. This buffer size is used for sending messages to be encrypted through symmetric or asymmetric algorithms, sequence updates and performing Message Authentication Codes (MACs) to name a few. The chosen buffer size for XOFs must strike a balance between the current use case (Kyber and Dilithium) and future use cases, *e.g.*, using the TPM as a DSA for rejection sampling when building pseudo-random secret data, or when performing a TPM backed mutually authenticated Key Exchange (KEX). Therefore, in order to reuse the maximum buffer struct already available while attempting forward-compatibility for the use-cases previously referred, the upper bound of an XOF was set to 1kB.

The integration of Kyber (Bos et al., 2018), NTTRU (Lyubashevsky and Seiler, 2019), and Dilithium (Ducas et al., 2019) implementations reuse the reference implementation provided by their authors. Even though there are optimized implementations available using vector instructions, these have not been used because current physical TPMs do not have access to those resources (Bos et al., 2018; Ducas et al., 2019).

For the most part Kyber and Dilithium follow the TSS's command specification, having been included in the TPM 2.0 standardized functions for key creation, signature creation and verification, and encryption and decryption functions. In addition, Kyber and NTTRU possess two other methods for key-exchange (encapsulation and decapsulation) which are not standardized in the TPM specification. Therefore, two new commands were added for the sole purpose of performing the Kyber and NTTRU encapsulation (`CC_KYBER_Enc` and `CC_NTTRU_Enc`) and decapsulation (`CC_KYBER_Dec` and `CC_NTTRU_Dec`). Further, to be on par with RSA and ECC support, two other commands were added for Kyber encryption (`CC_KYBER_Encrypt`) and decryption (`CC_KYBER_Decrypt`).

For the L-DAA (Kassem et al., 2019) implementation, different challenges arise in comparison to the previous algorithms. Even though the TPM specification offers standard commands for anonymous attestation, the L-DAA algorithm generates data responses three orders of magnitude larger than the default attestation algorithm. This limitation prevents the new L-DAA algorithm from using the default commands. Hence, and to avoid transferring MBs of data (or GBs using the highest considered security parameter set) in a single command transaction, the TSS was extended with new commands, namely: `CC_LDAA_Join` to perform the join operation; `CC_LDAA_SignProceed` to authorize the TPM to proceed with the signature; `CC_LDAA_CommitTokenLink` to calculate the token link, the error polynomial, and the basename polynomial to be used during the commitment procedures; `CC_LDAA_SignCommit{1, 2, 3}` to perform commitments; and `CC_LDAA_SignProof` to reply to the challenges sent by the host.

Supporting this L-DAA implementation and its associated data sizes required extensive modifications to the SW-TPM in order to reduce transfers of data to the order of MBs. Furthermore, there are two scenarios where the L-DAA implementation pushes

the memory boundaries of the current TPM architecture. The first scenario occurs when storing the L-DAA state in persistent memory so that an L-DAA session can be recovered across reboots. Each session state requires storing an additional 32MB in the persistent memory. In order to avoid further aggravating the memory constraints, the current implementation only supports one L-DAA session at a time. The second scenario occurs when processing the second and third commitments of the L-DAA signature. During this computation, a polynomial matrix must be shared between the TPM and the host, requiring GBs of memory, which the TPM can not sustain. Thus, the shared polynomial matrix is dynamically generated inside the TPM from a seed determined by the host.

Even though it is possible to change the persistent memory to an off-chip denser memory, and thus allow the TPM to withstand more than one L-DAA session at a time, the shared matrix poses greater architectural challenges. The matrix shared between the host and the TPM is known and constant at the beginning of the first commitment stage. The regeneration of the shared matrix imposes a high performance and power penalty, but so would its possible caching lead to a large area and memory overhead. Thus, matrix regeneration remains the only solution that the current TPM architecture can handle with the aforementioned memory increase.

The new proposed architecture with the addition of PQ algorithms is featured in Fig. 3.

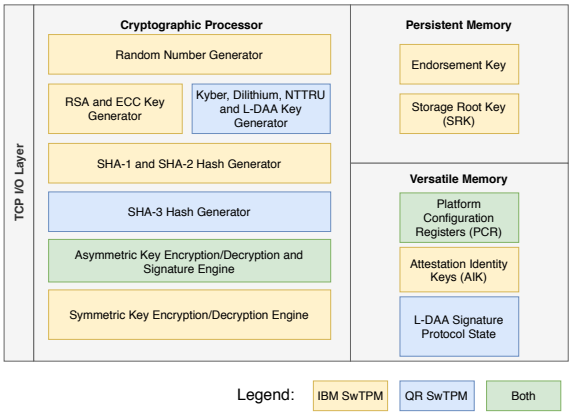

Figure 3: Proposed QR SW-TPM Architecture.

5 EXPERIMENTAL RESULTS

The baseline configuration used in performing the data size tests, in Tab. 1, and the execution time tests, in Tab. 2, is as follows. The ASCII string "My super secret. Please don't share.\n" is used for encryption and signature; signed messages use the SHA3-256 hash; and all keys are created as non-primary with the fixed TPM and parent properties. All the measured times result from taking the median over one hundred runs running on an Intel i5-5257U. The used TSS commands are: `CC_Create` for key creation; `CC_Sign` for data signature; `CC_VerifySignature` for signature verification; `CC_{KYBER, RSA}_Encrypt` and `CC_{KYBER, RSA}_Decrypt` for Kyber and RSA encryption and decryption; and `CC_{KYBER, NTTRU}_Enc` and `CC_{KYBER, NTTRU}_Dec` for Kyber and NTTRU encapsulation and decapsulation. All test were executed on the authors' fork of the SW-TPM (Fiolhais et al., 2020a) and the corresponding TSS (Fiolhais et al., 2020b).

Kyber (Bos et al., 2018), NTTRU (Lyubashevsky and Seiler, 2019), and Dilithium (Ducas et al., 2019) have been successfully implemented, replacing most RSA and ECC functionality. Comparing the key creation execution time, the QR algorithms show a speedup over RSA of 1.23x and a small slowdown of 0.95x in relation to ECC, specifically Kyber and Dilithium. Regarding signature and encryption/decryption execution times, the PQ schemes show commensurate execution times in comparison to their traditional counterparts. The largest discrepancy occurs in the decapsulation command using an ECC key. Since the ECC decapsulation requires more than one TSS command to perform, there is an accrued overhead in the total timing measurements from command issue and reception. Using the same Application Programming Interface (API) across all algorithms (same number of commands) would most likely result in commensurate results. However, if the application constraints allow it, the addition of a vector unit would provide speedups of 3x to the lattice-based algorithms in the QR TPM (Bos et al., 2017). Even though both schemes use larger public and private keys, they still fit into the current architecture with no modifications.

The L-DAA algorithm implementation is the outlier among the four implemented PQ algorithms. While the memory limitation issues exposed by L-DAA have been addressed, the persistent memory had to be increased from 64kB to 35MB, the versatile memory had to be increased from 154kB to 512MB, and the TCP I/O buffers had to be increased from 1MB to 128MB.

In order to set a baseline comparison between the L-DAA and the EC-DAA schemes, instead of comparing the final hashed signature, the comparison is solely set on the commit generation by the TPM. The

Table 1: Key, Signature, Commit and Encryption sizes (in kB) comparison between the non-PQ and PQ algorithms. The selected security mode for each PQ scheme follows the recommendations in the original publications (Bos et al., 2018; Ducas et al., 2019; Kassem et al., 2019; Lyubashevsky and Seiler, 2019)

	RSA (2048 bits)	ECC (nistp256)	EC-DAA	Kyber768	Dilithium (III)	L-DAA	NTTRU
Public Key	0.29	0.14	0.14	1.10	1.50	25.00	1.25
Private Key	0.27	0.10	0.10	2.50	3.60	24.00	2.50
Signature / Commit	0.26	0.08	0.25	—	2.70	624×10^3	—
Encryption	0.26	—	—	1.20	—	—	—
Encapsulation	—	—	—	1.09	—	—	1.25

Table 2: Execution Time (in ms) comparison between the non-PQ and PQ algorithms. The selected security mode for each PQ scheme follows the recommendations in the original publications (Bos et al., 2018; Ducas et al., 2019; Kassem et al., 2019; Lyubashevsky and Seiler, 2019)

	RSA (2048 bits)	ECC (nistp256)	EC-DAA	Kyber768	Dilithium (III)	L-DAA	NTTRU
Key Creation	226	166	166	169	170	374	166
Signature / Commit	176	179	168	—	185	7.2×10^6	—
Verify Signature	173	176	—	—	175	—	—
Encryption	173	—	—	173	—	—	—
Decryption	172	—	—	172	—	—	—
Encapsulation	—	166	—	169	—	—	170
Decapsulation	—	337	—	165	—	—	169

generation of each commit used the TSS commands `ldaa_signcommit[1|2|3]` for the L-DAA scheme and the `commit` command for the EC-DAA scheme. The comparison between the L-DAA scheme and the EC-DAA scheme is not favorable. The size of the commitment generated by the L-DAA is six orders of magnitude larger (Tab. 1) than the EC-DAA scheme. And the L-DAA is 43,000 times slower than the EC-DAA scheme (Tab. 2).

6 PROTOCOL CASE STUDY: FAST ID ONLINE

Fast ID Online (FIDO) is a set of technology-agnostic security specifications for strong authentication. The FIDO specifications support the Universal Authentication Framework (UAF) and Universal 2nd Factor (U2F) frameworks. While the former strives to achieve strong authentication without password, the latter strengthens password use by tying authentication to physical tokens. U2F has been growing in popularity since it is an open authentication standard that enables internet users to securely access a large number of online services with one single security key instantly and with no drivers or client software needed. In this section, the inclusion of the TPM in a U2F solution is considered. The user's FIDO-enabled device creates a new key pair, and the public key is shared with the online service and associated with the user's account. The service can then authenticate the user by requesting that the registered device signs a challenge with the private key.

The use-case scenario analyzes the enrolment of a device in a secure setting. This might correspond to the setting-up of the device by the IT department of a company. The key used to authenticate the device against other company services is generated and sealed on the TPM to the current machine state. This ensures that, after an employee is given the TPM-backed device, should it ever be compromised, the device will no longer be able to connect to other company services. In practice, authentication will be tied to the employee's password and the correct functioning of the device, significantly reducing the possibility of any attack within the company's network.

The TPM sealing process is a multi-step operation. Firstly, an audit session is initiated, which builds a digest of a series of TPM commands. As part of this session, the PCRs are read, such that the final audit

Table 3: Token sealing unsealing timing results (in ms) using Dilithium (k=3), RSA (2048 bits) and ECC (nistp256). Note that the timings take into account the execution of some commands more than once

Operation	Dilithium	RSA	ECC
CC_StartAuthSession	168	170	167
CC_PCR_Read	169	169	169
CC_PolicyPCR	168	168	168
CC_Unseal	167	171	168
CC_EncryptDecrypt	500	20	20

digest is representative of the computer state. A key is then created, passing the aforementioned digest as a policy that must be satisfied for the key to be used. The key creation operation takes as input the blob one wants to seal. The blob is encrypted by the created key whilst outside the TPM. Since the TPM implementation limits the size of the sealed blobs to 128B, it was decided to seal a randomly generated Advanced Encryption Standard (AES) key that is used to encrypt signing key material used for authentication in the context of FIDO. The sealed block can later be accessed by creating a new audit session, reading the PCRs values, and performing a call to the unseal operation associated with this session. As long as the machine is in a trustworthy state, the PCRs will contain the same values, and the unsealing will be successful.

The execution timings of the TPM main operations for accessing the FIDO credentials, for authentication processes based on Dilithium, RSA and ECC, have been experimentally evaluated and are reported in Tab. 3. The main difference in the execution timings pertains the CC_EncryptDecrypt command. This results from the fact that the signing keys are encrypted under an AES key that is sealed to the computer state. The TPM implementation encrypts at most 1kB of data at a time. While for both RSA and ECC a small amount of data has to be decrypted, Dilithium's key material is large, and three calls to CC_EncryptDecrypt are required to completely decrypt it. Nevertheless, the execution time necessary to access the Dilithium key is still small enough, and may, for instance, be executed at the same time as the user is typing their password. This gives strong incentives towards the integration of the FutureTPM into FIDO, as it provides for quantum resistance with no noticeable degradation in performance.

7 APPLICATION CASE STUDY: TPM-BACKED QR-TLS

To further test the Kyber and Dilithium implementations in a real world scenario, a light Transport Layer Security (TLS) 1.3 test application was created backed by the proposed QR-TPM. The TLS protocol is widely used to set up an encrypted communication channel between a server and a client wherein both endpoints can freely share sensitive data. From a high-level perspective, a TLS 1.3 connection setup starts by performing a Key Encapsulation Mechanism (KEM), so that both parties generate the same symmetric key to be used the in the secure tunnel. Once the shared secret has been calculated, the client and the server start communicating solely through secret-key cryptography. Then the client checks the validity of the server's certificate. Upon a successful confirmation both endpoints can start communicating in the secure channel.

Providing a high-level interface to an application requires updating OpenSSL with support for new Kyber and Dilithium bindings. In addition, the requests for the QR algorithms need to be forwarded to the TPM, which OpenSSL allows by changing the underlying cryptographic engine. Once the engine is swapped, there needs to be a middleware between OpenSSL and the TPM to interpret and route requests. In this instance, Intel's TSS implementation was used (Intel and Contributors, 2020). Note that, due to limitations in Intel's TSS, the ephemeral key generation and KEM are performed in OpenSSL, whereas signature verification and generation are performed in the QR TPM. The final API layers are put together as in Fig. 4.

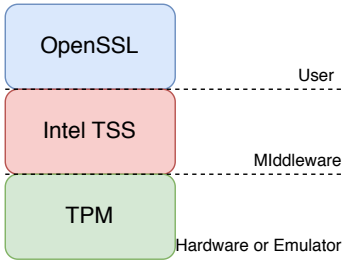

Figure 4: API stack used to build a QR TLS.

To test this application two scenarios were devised, one using standard state-of-the-art encryption and another using QR encryption, both supported the proposed SW-TPM as a cryptographic engine. The state-of-the-art scenario uses an RSA (2048 bits) certificate and an ephemeral Elliptic curve Diffie-Hellman (ECDH) (nistp256) key. This configuration meets the TLS 1.3 requirements and is widely used in servers and browsers. The QR configuration uses a Dilithium (III) certificate and an ephemeral Kyber768 key.

All tests were executed using the OpenSSL s_server and s_client commands to build a server

and a client, where the client would request the server for its `index.html`. Using the state-of-the-art configuration a session setup takes 430 ms, whereas the QR configuration takes 290 ms. These results are the median of one thousand runs for each configuration.

8 DISCUSSION

Experimental results suggest that the benefits of adding Kyber, NTTRU, and Dilithium to an hardware TPM far outweigh their cost from a security and resources perspective. In addition to their execution times and memory requirements having similar costs to traditional cryptography, the implementation of a single vector unit brings performance improvements to both schemes. In contrast, the current L-DAA proposal (Kassem et al., 2019) can not be feasibly implemented or used in a physical TPM.

The implementation of QR algorithms highlight how insufficiently equipped the current TPM architecture is when using lattice-based algorithms. The addition of a vector unit would net a 3x speedup to most lattice-based primitives. Alternatively, the addition of DSAs for polynomial arithmetic would greatly benefit the TPM. There is rich literature in this field which could be leveraged (Sinha Roy et al., 2019; Riazi et al., 2020).

However, the addition of QR schemes to the TPM should not be regarded as an all or nothing scenario. Rather, the proposed schemes should be carefully chosen and slowly rolled into the TPM architecture. Three great candidates are: Kyber, Dilithium, and NTTRU. Since they all share the same underlying arithmetic, the addition of one streamlines the addition of the remainder or future lattice-based proposals. Further, from the protocol and application case studies, it can be concluded that Kyber and Dilithium can be deployed with marginal impact to the user.

The high memory usage of the L-DAA can be interpreted as the immaturity of the algorithm and should not be considered as a motive to disregard lattice-based attestation. More generally, this should not be seen as a deterrent to the usage of other QR cryptographic schemes. One may, for instance, use QR encryption along with traditional DAA signatures for authentication. In the event that a quantum computer is developed in future years, the cryptograms exchanged with Kyber now will remain secure, while attestation keys can easily be revoked.

9 CONCLUSION AND FUTURE WORK

This first paper exploring the integration of QR primitives in the TPM shows that one can do so while maintaining much of the same infrastructure of TPM 2.0, for instance regarding the generation of keys and the generation and verification of digital signatures. However, when algorithms significantly exceed the sizes of pre-allocated buffers and message sizes of current architectures, their implementation may be too expensive for physical platforms. From the presented analysis, both in synthetic testing, and protocol and application use cases, it is ascertained that the Kyber, NTTRU, and Dilithium implementations can replace most of the functionality provided by the non-QR ECC and RSA algorithms, while offering higher security at the cost of a small increase in memory footprint. In contrast, the addition of the considered L-DAA scheme drastically increases the resources necessary for the functioning of the TPM, limiting its application to virtual or firmware TPMs of high-performance computing architectures. Since one of the major obstacles in future-proofing the TPM is related to efficiently achieving QR DAA, next versions of the SW library will consider other recent developments in this domain, such as (Chen et al., 2019).

ACKNOWLEDGEMENTS

This research was supported by European Unions Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 and FCT Grant No. SFRH/BD/145477/2019.

The authors would like to thank Athanasios Giannetsos and Sofianna Menesidou for their insights and for providing the TPM protocol code on Sec.6.

REFERENCES

Arthur, W. and Challener, D. (2015). *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, Berkely, CA, USA, 1st edition.

Aumasson, J.-P., Bernstein, D. J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.-L., Hulsing, A., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M. M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., and Schwabe, P. (2019). Sphincs+ – submission to the 2nd round of the nist post-

quantum project. https://sphincs.org/data/sphincs+-round2-specification.pdf.

Bernstein, D. J., Buchmann, J., and Dahmen, E. (2008). *Post Quantum Cryptography*. Springer Publishing Company, Incorporated, 1st edition.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehle, D. (2018). CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 353–367.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehl, D. (2017). Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634. https://eprint.iacr.org/2017/634.

Brickell, E., Camenisch, J., and Chen, L. (2004). Direct anonymous attestation. Cryptology ePrint Archive, Report 2004/205. https://eprint.iacr.org/2004/205.

Chen, L., El Kassem, N., Lehmann, A., and Lyubashevsky, V. (2019). A Framework for Efficient Lattice-Based DAA. In *Proceedings of the 1st Workshop on Cyber-Security Arms Race (CYSARM)*.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and Stehl, D. (2019). Dilithium - Submission to the NIST post-quantum project. https://pq-crystals.org/dilithium/data/dilithium-specification-round2.pdf. [Online; accessed 5-September-2019].

Fiolhais, L., Martins, P., and Sousa, L. (2020a). Software TPM. https://github.com/FutureTPM/sw-tpm.

Fiolhais, L., Martins, P., and Sousa, L. (2020b). Software TSS. https://github.com/FutureTPM/tss.

Fuchs, A. (2018). PQC TSS and PQC TPM - a prototype. 1st FutureTPM Workshop on Quantum-Resistant Crypto Algorithms. https://futuretpm.eu/1st-futuretpm-workshop.

FutureTPM (2019). D5.1 first version of implementation. https://futuretpm.eu/downloads/FutureTPM-D5.1-FutureTPM-1st_version_of_implementation-PU-M18.pdf.

Group, T. C. (2016). Trusted Platform Module (TPM) 2.0: A BRIEF INTRODUCTION. Technical report, Trusted Computing Group. https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf.

Gueron, S. and Schlieker, F. (2016). Speeding up r-lwe post-quantum key exchange. Cryptology ePrint Archive, Report 2016/467. https://eprint.iacr.org/2016/467.

IBM, Microsoft, and Contributors. IBM's Software TPM 2.0. https://sourceforge.net/projects/ibmswtpm2/. [Online; accessed 16-March-2020].

IBM, Microsoft, and Contributors. IBM's TPM 2.0 TSS. https://sourceforge.net/projects/ibmtpm20tss/. [Online; accessed 16-March-2020].

Intel and Contributors (2020). OSS implementation of the TCG TPM2 Software Stack (TSS2). https://github.com/tpm2-software/tpm2-tss. [Online; accessed 16-March-2020].

Kassem, N. E., Chen, L., Bansarkhani, R. E., Kaafarani, A. E., Camenisch, J., Hough, P., Martins, P., and Sousa, L. (2019). More efficient, provably-secure direct anonymous attestation from lattices. *Future Generation Computer Systems*, 99:425 – 458.

Lyubashevsky, V. and Seiler, G. (2019). Nttru: Truly fast ntru using ntt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019, Issue 3:180–201.

Microsoft (2018). TPM recommendations. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/tpm-recommendations.

NIST (2015). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. [Online; accessed 5-September-2019].

Riazi, M., Laine, K., Pelton, B., and Dai, W. (2020). Heax: An architecture for computing on encrypted data. In *ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134.

Sinha Roy, S., Turan, F., Jarvinen, K., Vercauteren, F., and Verbauwhede, I. (2019). Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398.