Post Quantum Design in SPDM for Device Authentication and Key Establishment

Jiewen Yao[1], Krystian Matusiewicz[1], and Vincent Zimmer[1]

Intel Corporation, USA

Abstract. The Security Protocol and Data Model (SPDM) defines flows to authenticate hardware identity of a computing device. It also allows for establishing a secure session for confidential and integrity protected data communication between two devices. The present version of SPDM, namely version 1.2, relies on traditional asymmetric cryptographic algorithms that are known to be vulnerable to quantum attacks. This paper describes the means by which support for post-quantum (PQ) cryptography can be added to the SPDM protocol in order to enable SPDM for the upcoming world of quantum computing. We examine SPDM 1.2 protocol and discuss how to negotiate the use of post-quantum cryptography algorithms (PQC), how to support device identity reporting, means to authenticate the device, and how to establish a secure session when using PQC algorithms. We consider so called hybrid modes where both classical and PQC algorithms are used to achieve security properties as these modes are important during the transition period. We also share our experience with implementing PQ-SPDM and provide benchmarks for some of the winning NIST PQC algorithms.

Keywords: PQ digital signature · PQ key establishment · post quantum SPDM · device authentication · device secure session.

1 Introduction

The Security Protocol and Data Model (SPDM) 1.2 specification [16] is defined by the Distributed Management Task Force (DMTF). It is used for device identity collection, device authentication, measurement collection and device secure session establishment. The SPDM protocol is a standard for the device community. It is adopted by multiple other standard groups, including Peripheral Component Interconnect (PCI) [35], Compute Express Link (CXL) [11], Mobile Industry Processor Interface (MIPI) [27], and Trusted Computing Group (TCG) [48].

The current SPDM 1.2 specification uses traditional asymmetric cryptographic algorithms, such as RSA, ECDSA, EdDSA for digital signatures and ephemeral Diffie-Hellman over finite fields (FFDHE) or elliptic curves (ECDHE) for key establishment. Unfortunately, all of the earlier listed algorithms are not quantum safe. Attacks based on Shor's algorithm [44, 45, 37, 39] will break cryptosystems relying on the hardness of integer factorization and discrete logarithms in abelian groups. While the race to build large scale quantum computers is still at an early phase, some devices have long lifetimes and need to be supported beyond 2030. Beyond 2030 is when people expect quantum computers may be available. If a device uses SPDM and needs to be supported beyond 2030, then likely it will need to be PQC-ready and it seems prudent to consider supporting post-quantum cryptography (PQC) [29] in the SPDM protocol.

A lot of research has been dedicated to network security with PQC, such as X.509 certificates [33, 50, 32, 23, 36], Media Access Control security (MACsec) [9], Internet Protocol Security (IPSec) [17, 49], Transport Layer Security (TLS) [47, 8, 6, 25, 40, 51, 41, 10, 46, 43, 42], Secure Shell (SSH) [41, 24], WireGuard [20, 38, 26]. PQC algorithms have been implemented in devices for secure boot [21, 22].

As far as we know, this is the first study of PQC algorithm adoption for a *device* communication security protocol, such as SPDM. This aspect seems to be currently particularly relevant as the IoT market is expanding and on one hand we get more and more devices with diverse security needs (including long-term security requirements) and on the other hand, IoT devices need to be served by backend services where performance is also critical due to the scale factor.

1.1 Our Contributions

Our focus in this work is to provide a construction for supporting post-quantum cryptography algorithms in the SPDM protocol, including algorithm negotiation, device certificate transfer, digital signature signing and verification, and authenticated key exchange for secure session. We explored a prototype implementation of the SPDM PQC capability in a device with different modes:

- Traditional mode: PQC algorithms not used,
- PQC mode: only PQC algorithms are used,
- Hybrid mode: using both traditional algorithms and PQC algorithms.

The hybrid mode was studied in [5, 4] and recommended by NIST during the transition and migration phase [28].

We also consider design challenges for adopting post-quantum algorithms for devices with SPDM capability, such as backward compatibility and message size.

The paper is organized as follows. We start with a brief recap of SPDM in Section 2. Section 3 is dedicated to extending algorithm negotiation phase. Then, we follow with changes to support PQC-ready device identities in Section 4 and device authentication in Section 5. Establishment of secure session keys is treated in detail in Section 6 with security analysis. Finally, we report the results of our proof-of-concept implementation and discussion in Section 7 and Section 8.

The PQC algorithm selection has been driven by National Institute of Standards and Technology (NIST) [30]. While te protocol design is algorithm agnostic and could be used with any new PQC scheme, we report performance results for the four winners of the NIST PQC competition announced very recently.

Note that since symmetric cryptographic algorithms can be used in the quantum computing world with increased key sizes and larger hash digests, we do not study any symmetric cryptographic algorithms for SPDM, such as hash algorithms or Authenticated Encryption with Associated Data (AEAD) algorithms.

2 SPDM Background

2.1 SPDM Device Authentication

In the SPDM specification, the authentication of the identity of a device involves two steps: device identification and device authentication.

During the device manufacturing phase, each device is provisioned with a device certificate chain. This certificate chain can be treated as the device identity. The device certificate chain includes all the certificates from a trusted root certificate authority (CA) certificate that chains to a device specific leaf certificate. The device certificate contains the public key that corresponds to the device private key. The root CA endorses the device public / private key pair through the certificate chain. At runtime, an SPDM initiator (requester) may use a `GET_CERTIFICATE` message to ask an SPDM device (responder) to return its certificate chain as the identity.

The SPDM device authentication uses a challenge–response mechanism. The initiator (requester) sends a `CHALLENGE` message to the SPDM device (responder). The `CHALLENGE` message includes a nonce to prevent replay attacks. Then the device signs the challenge with its private key and returns a `CHALLENGE_AUTH` message. The authentication initiator can verify the digital signature by using the device public key taken from the certificate chain.

2.2 SPDM Secure Session

The SPDM specification allows two devices to establish a secure communication channel, similar to the network Transport Layer Security (TLS) 1.3. An SPDM requester and an SPDM responder may use an authenticated key exchange protocol to derive a set of session keys. The session keys will provide confidentiality and integrity for the communication data by using encryption and message authentication.

2.3 SPDM Algorithm Negotiation

SPDM specification defines a message `NEGOTIATE_ALGORITHMS` sent by an SPDM requester and a message `ALGORITHMS` returned by the SPDM responder that together perform the task of negotiating a common set of cryptographic algorithms. The cryptographic algorithm selection in SPDM is different from the way it is implemented in TLS. TLS defines a set of cipher suites. A TLS entity selects one cipher suite that covers all algorithms, including key exchange or agreement, authentication, block or stream ciphers, and message authentication. An SPDM requester lists all individual algorithms it can support, such as hash algorithms, responder direction asymmetric digital signature algorithms, requester direction asymmetric digital signature algorithms, key exchange algorithms and Authenticated Encryption with Associated Data (AEAD) ciphers. The SPDM responder chooses one of these options in each category as the final negotiated algorithm. As opposed to TLS cipher suites, each type of SPDM algorithm can be negotiated separately.

3 Post Quantum Design for SPDM Algorithm Negotiation

The SPDM specification defines cryptographic algorithms separately. An SPDM entity can negotiate an individual cipher, such as a hash algorithm, responder direction asymmetric digital signature algorithm, requester direction asymmetric digital signature algorithm, key exchange algorithm and Authenticated Encryption with Associated Data (AEAD) cipher suite. Table 1 shows an example.

Table 1. Current SPDM Algorithm Negotiation (Example)

Algorithm	Requester's List	Responder's Selection
Hash	`SHA256`, `SHA384`	`SHA384`
Responder Digital Signature	`RSASSA_3072`, `ECDSA_NIST_P256`, `ECDSA_NIST_P384`	`ECDSA_NIST_P384`
Requester Digital Signature	`RSASSA_3072`, `RSAPSS_3072`	`RSAPSS_3072`
Key Exchange	`FFDHE_3072`, `ECDHE_secp256r1`, `ECDHE_secp384r1`	`ECDHE_secp384r1`
AEAD	`AES_256_GCM`, `CHACHA20_POLY1305`	`AES_256_GCM`

In order to support a post quantum world, we extend three algorithm types: PQC responder digital signature algorithm, PQC requester digital signature algorithm, and PQC key exchange algorithm. The original responder digital signature algorithm, requester digital signature algorithm, and key exchange algorithms are interpreted as traditional algorithms only to support compatibility.

Table 2 shows an example of an SPDM algorithm negotiation with PQC in traditional mode, PQC mode, and hybrid mode. The Responder Digital Signature designates the SPDM asymmetric key signature algorithm for signature generation by Responder and verification by Requester. The Requester Digital Signature describes the SPDM asymmetric key signature algorithm for signature generation by Requester and verification by Responder. The PQC algorithms are separated from traditional algorithms because we expect the Responder will select one of the PQC algorithms and one of the traditional algorithms in hybrid mode.

Assuming the SPDM Requester has full capabilities, including traditional only, PQC only and hybrid mode, the Requester needs to declare all supported algorithms. The SPDM Responder needs to make the final decision. If the Responder wants to choose traditional only, it shall select one of the traditional algorithms and not select any PQC algorithm. If the Responder wants to choose PQC only, it shall select one of the PQC algorithms and not select any traditional algorithm. If the Responder wants to choose hybrid mode, it shall select one traditional algorithm and one PQC algorithm.

Table 2. SPDM Algorithm Negotiation with PQC (Example)

Algorithm	**Requester's List** (full capability)	**Responder's Selection** (traditional mode)	**Responder's Selection** (PQC mode)	**Responder's Selection** (hybrid mode)
Hash	`SHA256`, `SHA384`	`SHA384`	`SHA384`	`SHA384`
Responder Digital Signature	`RSASSA_3072`, `ECDSA_NIST_P256`, `ECDSA_NIST_P384`	`ECDSA_NIST_P384`	-	`ECDSA_NIST_P384`
Requester Digital Signature	`RSASSA_3072`, `RSAPSS_3072`	`RSAPSS_3072`	-	`RSAPSS_3072`
Key Exchange	`FFDHE_3072`, `ECDHE_secp256r1`, `ECDHE_secp384r1`	`ECDHE_secp384r1`	-	`ECDHE_secp384r1`
AEAD	`AES_256_GCM`	`AES_256_GCM`	`AES_256_GCM`	`AES_256_GCM`
PQC Responder Digital Signature	`Falcon-512`, `Falcon-1024`	-	`Falcon-512`	`Falcon-512`
PQC Requester Digital Signature	`Dilithium2`, `Dilithium5`	-	`Dilithium2`	`Dilithium2`
PQC Key Exchange	`Kyber512`, `SIDH-p434`, `SIKE-p434`	-	`SIDH-p434`	`SIDH-p434`

There is one limitation if we use this mechanism. A requester or a responder may want to say: "I can support either traditional mode or PQC mode, but I don't want to support hybrid mode." There is no way to pass this information via the `NEGOTIATE_ALGORITHMS` request and `ALGORITHMS` response message. As such, we need to use the `GET_CAPAILITIES` request and `CAPABILITIES` response message. To support this usage we have added more capabilities bits, including:

- PQC Capability: this means an entity supports PQC mode
- Hybrid Capability: this means an entity support Hybrid mode.

If both entities indicate PQC capability, then PQC mode is chosen. If both entities indicate hybrid capability, then hybrid mode is chosen. If both PQC capability and hybrid capability is selected, then hybrid mode takes precedent. We do not define a traditional algorithm capability for compatibility consideration.

The SPDM specification also defines the timing requirements. The requester and the responder need tell the peer a `CTExponent` timing parameter, which is the maximum amount of time for cryptographic processing. A PQC algorithm might have different timing requirement comparing with traditional algorithm. The timing in the hybrid mode could be the addition of timing in traditional mode and timing in PQC mode. Unfortunately, the timing parameter in `CAPABILITIES` is negotiated before the algorithm negotiation in `ALGORITHMS`. It is hard to use a single `CTExponent` to indicate the timing parameter in three different modes. Setting a maximum value is one option, but it might cause the requester wait for long time to issue a retry. The other option is to add a new `PQCTExponent` to indicate the PQC specific timing. The final interpretation of the timing parameter could be:

- Timing in Traditional Mode: $2^{\texttt{CTExponent}}$
- Timing in PQC Mode: $2^{\texttt{PQCTExponent}}$
- Timing in Hybrid Mode: $2^{\texttt{CTExponent}} + 2^{\texttt{PQCTExponent}}$

3.1 Design Considerations

Mode identification A requester and a responder may negotiate traditional algorithms and PQC algorithms separately. A device only supporting traditional mode may provide or select traditional algorithms and ignore PQC algorithms. A PQC mode only device may provide or select PQC algorithms and ignore traditional algorithms. A hybrid mode aware device may provide a list of traditional algorithms and a list of PQC algorithms or select one traditional algorithm and one PQC algorithm. An SPDM requester or an SPDM responder can know the final negotiated mode by checking if there are both traditional algorithms and PQC algorithms negotiated.

The other option is to merge the traditional algorithm and PQC algorithm together in one field. There is no impact to a requester in choosing this approach. The impact for a responder is that we need to allow the algorithm selection field to designate one or two algorithms. If only one algorithm is selected, then it is traditional mode or PQC mode. If two algorithms are selected - one that is traditional and the other that is PQC, then it is hybrid mode.

No Combinatorial Explosion Since SPDM allows both entities to negotiate algorithms individually, there is no danger of algorithm combination explosion. We use a similar technique to handle the hybrid mode. SPDM uses one bit to indicate one algorithm. In order to support N traditional mode ciphers and M PQC mode ciphers, the total required bit space is $(N + M)$.

Security Level Matching Technically, traditional algorithms and PQC algorithms are orthogonal. In the real world, a product may need to meet a NIST security level requirement. As such, an entity should choose the required traditional algorithm and a PQC algorithm with a matching security level in hybrid mode. For example, the Level 1 PQC algorithm should go with the ECC P-256 curve, the Level 3 PQC algorithm should go with the ECC P-384 curve, and the Level 5 PQC algorithm should go with the ECC P-521 curve.

Algorithm Set Matching Besides a set of NIST algorithms such as SHA, AES, RSA, DH and ECC, SPDM 1.2 also adds support for Shang-Mi (SM) algorithms, such as SM3, SM4, and SM2. In the real world, the product may choose either a NIST algorithm set, or an SM algorithm set. In the future, if NIST or SM defines its own PQC algorithm set, the PQC algorithm should go with the tradition algorithm in the same set.

Hash Algorithm used in digital signature Some traditional digital signature algorithms require fixed length input messages for signing. To reduce the combinatorial explosion problem, the hash algorithm is negotiated separately with the digital signature algorithm in SPDM. Some PQC digital signature algorithms, such as SPHINCS, have an associated hash algorithm such as SHA256 or SHAKE256. How do we indicate the hash algorithm required in PQC digital signature? Currently, we let the original hash algorithm only associate with a traditional digital signature. The PQC digital signature algorithm should indicate the hash algorithm explicitly. For example, `SPHINCS+-SHA256-128f-robust` or `SPHINCS+-SHAKE256-128f-robust` are potential options.

4 Post Quantum Design for SPDM Device Identity

The SPDM specification uses the X.509 certificate chain as the device identity. If an SPDM device supports a PQC algorithm (PQC and/or hybrid mode), the device should carry an X.509 certificate with PQC support, namely a PQC or hybrid X.509 certificate. In the hybrid mode, the SPDM requester can use the `GET_CERTIFICATE` command to retrieve one X.509 certificate chain and `SET_CERTIFICATE` command to provision one. This one certificate chain contains both a traditional algorithm and a PQC algorithm. In PQC mode or traditional mode, this hybrid certificate chain may still be used since the verifier may just skip the unneeded algorithm and certificate digital signature.

The alternative approach is to extend the `GET_CERTIFICATE` command to indicate which type of certificate chain is required, namely a traditional certificate chain or PQC certificate chain. The Certificate Type can be in `param2` and includes

- Traditional Certificate Chain (0),
- PQC Certificate Chain (1),
- Hybrid Certificate Chain (2).

In the hybrid mode, the requester needs to request the certificate chain twice: one for the traditional certificate chain and the other for the PQC certificate chain.

The X.509 certificate format is out of scope of the SPDM specification so we will not discuss the details here.

The SPDM specification may also allow a device to provision a raw public key of the peer during the manufacturing phase. In this case, the trust of the public key of the peer is established without the certificate based public key infrastructure. If PQC is required in the raw public key scenario then the manufacturer may need to provision the public keys for both a traditional algorithm and a PQC algorithm. The format of the raw public keys is out of scope of the SPDM specification. That raw key format is implementation specific so we don't discuss the details here.

The SPDM 1.2 specification defines an alias certificate mode to support the device compatible with Trusted Computing Group (TCG) Device Identity Composition Engine (DICE) specification. A DICE device may include a device ID certificate and alias certificate. The device ID certificate includes the device ID key derived from the Compound Device Identity (CDI) value computed by the DICE process. The certificate depends on the unique device secret (UDS) and measurement of DICE layer 0. The alias certificate includes a device alias key that is computed using the last CDI value in the chain of the device Trust Computing Base (TCB) component. The alias certificate is usually generated during the runtime because the CDI includes the measurement of the last layer, which includes mutual firmware. If a DICE device chooses to support a PQC alias certificate chain or Hybrid alias certificate chain, each DICE layer needs to support the PQC algorithm.

4.1 Design Considerations

No Duplication In the hybrid mode, if we require an entity to pass one hybrid certificate chain, then the message includes the identity information. In this case, the authority information just needs to be transmitted once.

If we require an entity to pass both a traditional certificate chain and a PQC certificate chain, then the identity information and authority information will be transmitted twice. Also, both certificate chains needs to be included in the transcript calculation. This is not efficient and might cause latency problems. In addition, the entity needs to check if the two certificate chains use the same identity information or authority information. If they are different, then the entity needs to record the difference and verify against the pre-defined policy, which adds lots of complexity.

Local Storage Size When an entity considers the compatibility requirements, it will consider the capability of the peer. Because the peer may support traditional mode, PQC mode and hybrid mode, the entity may need to prepare the traditional mode certificate chain, PQC mode certificate chain and hybrid mode certificate chain. A small device may only have limited storage size. Provisioning three certificate chains might not be the best choice.

Transport Message Size The SPDM specification does not define the message size limitation. However, the secured message using SPDM [15] defines a 16 bit application data length field. As such, the maximum size of an encrypted message is 2^{16} bytes. SPDM transport layer binding specifications also have a size limitation. For example, the PCI Data Object Exchange (DOE) mailbox [34] uses a 18 bit length field for the double-word (4 bytes) number, which means the maximum message size is 2^{20} bytes. The Management Component Transport Protocol (MCTP) over System Management Bus (SMBus) / Inter-Integrated Circuit (I2C) [12, 14, 13] only supports 256 bytes as a maximum in one packet.

For a large message that may potentially exceed the 256 byte limitation, SPDM 1.0 and 1.1 defines a command specific chunking mechanism. For example, the `GET_CERTIFIACTE` request message includes a 16 bit `OFFSET` and 16 bit `LENGTH` fields to indicate the requested certificate data buffer offset and length. The `CERTIFICATE` response message includes a 16 bit `PortionLength` and 16 bit

RemainderLength to indicate the transmitted data buffer length and remaining data buffer length. SPDM does not rely on low level transport layer fragmentation. A more generic chunking mechanism (CHUNK_CAP) is defined in SPDM 1.2. As such, all SPDM messages can support chunking after the chunking capability is negotiated, such as the SET_CERTIFIACTE and CSR.

The SPDM public certificate size depends upon the PQC public key size. Some PQC algorithms, such as Dilithium and Falcon, use public keys larger than 256 bytes. The public key size of Rainbow and GeMSS algorithms is larger than 2^{16} bytes. As such, the current GET_CERTIFICATE command cannot meet the requirement. We may need to extend the 16 bits offset and length field to 32 bits in a new version of GET_CERTIFICATE, or we can redefine GET_CERTIFICATE to use the generic SPDM 1.2 chunking mechanism. However, considering a recent attack [3] on Rainbow, it might not be a concern anymore.

In addition, we may define a new format in secured message using SPDM [15] by extending the 16 bit application length field to 32 bit, in order to support the device without chunking capability.

5 Post Quantum Design for SPDM Device Authentication

Some SPDM messages, such as CHALLENGE_AUTH response, MEASUREMENTS response, KEY_EXCHANGE_RSP response, and FINISH request, require a digital signature, such as RSA and ECDSA. The SPDM specification defines the binary format of the digital signature for a specific algorithm with a fixed size.

The signature format in PQC mode is exactly the same as the one in traditional mode. The signature size can be interpreted as the digital signature size of the PQC algorithm. In hybrid mode, we require that both traditional algorithm and PQC algorithm sign the same message data. The digital signature field in the SPDM message should be the concatenation of the two signatures. The first part is the traditional digital signature with fixed size, and the second part is the PQC digital signature. In order to maintain compatibility we do not add the signature size field. The format of the PQC digital signature should be defined by the PQC algorithm.

5.1 Design Considerations

Hybrid Digital Signature We require that both a traditional algorithm and a PQC algorithm sign the same message data and then concatenate signatures together.

We do not choose the option to let one algorithm sign the digital signature output from the other algorithm. This aligns with the current practice in TLS hybrid mode.

No duplication In the hybrid mode, the traditional and PQC digital signatures are combined together. There is no need to send the CHALLENGE/CHALLENGE_AUTH twice - one is for classic mode and the other is PQC mode. The transcript calculation just needs to happen one time.

Transport Message Size As we discussed in a previous section, there is a size limitation in the transport message. Unfortunately, most post quantum algorithms have a large digital signature size. The signature size of Dilithium, Falcon, SPHINCS+, and picnic algorithms are larger than 256 bytes. The signature size of `picnic_{L3,L5}_{FS,UR,full}` can be larger than 2^{16} bytes.

Currently there are multiple SPDM messages including digital signature - `CHALLENGE_AUTH` response, `MEASUREMENTS` response, `KEY_EXCHANGE_RSP` response, and `FINISH` request message. In order to support large message transportation, we can use the generic SPDM 1.2 chunking message for them.

Timing As we discussed in a previous section, a PQC algorithm may have different signature generation timing requirement. Some hash based signature (HBS) algorithms, such as SPHINCS+, are much slower than the lattices based algorithms, such as Dilithium2 or Falcon. The `PQCTExponent` should be larger than the maximum signing time for the supported PQC signature algorithms.

6 Post Quantum Design for SPDM Secure Session

SPDM defines two ways to build a secure session, based on either a pre-shared key (PSK) or an asymmetric key exchange. The PSK based key exchange involves only hash-based message authentication code (HMAC), which is still secure against quantum adversaries. The asymmetric key exchange uses ephemeral Diffie-Hellman (DH) over finite fields or elliptic curves. The SPDM specification defines the binary format of the DH public key (`ExchangeData`) for both the SPDM requester and the SPDM responder.

The SPDM 1.2 specification only supports FFDHE and ECDHE. One-way authentication is always required in an SPDM key exchange, while mutual authentication is optional.

NIST supports Key Encapsulation Mechanism (KEM) for the shared key generation. The only DH style PQC - Supersingular Isogeny Diffie-Hellman (SIDH) is broken recently [7]. We adopted KEM style algorithm to make the `ExchangeData` format in PQC mode similar to the one in the traditional mode. The requester's `ExchangeData` is the requester's public key generated with a PQC KEM key generation. The responder's `ExchangeData` is the responder's cipher text generated with the PQC KEM encapsulation. In hybrid mode, the `ExchangeData` field in the SPDM message should be the concatenation of the two key `ExchangeData`. The first part is the traditional key `ExchangeData` with a fixed size, and the second part is the PQC key `ExchangeData` with a fixed size. The format of the PQC key `ExchangeData`, such as public key and cipher text, should be determined by the selected PQC algorithm. With this approach, only key agreement part of SPDM is converted to PQC. The remaining portion, such as the identity authentication, is unchanged.

We did notice that KEM based authentication [18] was adopted by several standard proposals, such as wireguard [20] and KEMTLS [8, 43, 42]. However, that will involve a much bigger impact for the SPDM `KEY_EXCHANGE` message flow, as well as other digital signature based commands such as SPDM

`CHALLENGE` or `GET_MEASURMENT`. Our future work is to investigate the design decision on how to support SPDM KEM based Authentication and if we can get any benefit.

6.1 SPDM KEM message flow

For PQC-SPDM, we use a key encapsulation mechanism (KEM) based authenticated key exchange to replace DHE. Fig. 1 shows the high-level view of PQ-SPDM KEM-based handshake message flow with mutual authentication. The step in the FINISH message that is typeset with an asterisk is not required in one-way authentication. The KEM-related action is highlighted in blue. The KEM ephemeral secret key (esk) is highlighted in red, and the KEM ephemeral public key (epk) and cipher text are highlighted in green. *spdmvca* means the SPDM transcript for the SPDM command/response: `GET_VERSION/VERSION`, `GET_CAPABILITIES/CAPABILITIES`, and `NEGOTIATE_ALGORITHMS/ALGORITHMS`.

First Message: `KEY_EXCHANGE` (initiator to responder)

1. The initiator needs to use the KEM algorithm to generate an ephemeral private/public key pair (esk_i and epk_i).
2. To identify the session, the initiator generates a 16bit session ID (sid_i) as the first half of 32bit session ID.
3. To prevent the replay attacks, the initiator generates a 32bit random number ($rand_i$).
4. The `KEY_EXCHANGE` message ($keyex_i$) is the concatenation of the opcode ($keyexop_i$), the initiator generated session ID (sid_i), the random number ($rand_i$), the ephemeral public key (epk_i) and the initiator specific opaque data.

Second Message: `KEY_EXCHANGE_RSP` (responder to initiator)

1. After the responder receives the `KEY_EXCHANGE` message, the responder uses the KEM algorithm to encapsulate the ephemeral public key (epk_i) and derives the shared session key and the cipher text.
2. The responder generates a 16bits session ID (sid_r) as the second half of the 32bit session ID. The final session ID is the concatenation of the initiator's session ID and the responder's session ID ($sid_i||sid_r$).
3. The responder also generates a 32bit random number ($rand_r$).
4. Now the responder can prepare the `KEY_EXCHANGE_RSP` message ($keyex1_r$), which is concatenation of the opcode ($keyexop_r$), the half of the session ID (sid_r), the 32bit random number ($rand_r$), the KEM cipher text and the responder specific opaque data.
5. From the shared key, the responder uses SPDM defined key schedule algorithm to derive the ephemeral finish key (efk), initiator direction ephemeral handshake key (ehk_i) and responder direction ephemeral handshake key (ehk_r). The ephemeral finish key (efk) will be used to generate a message

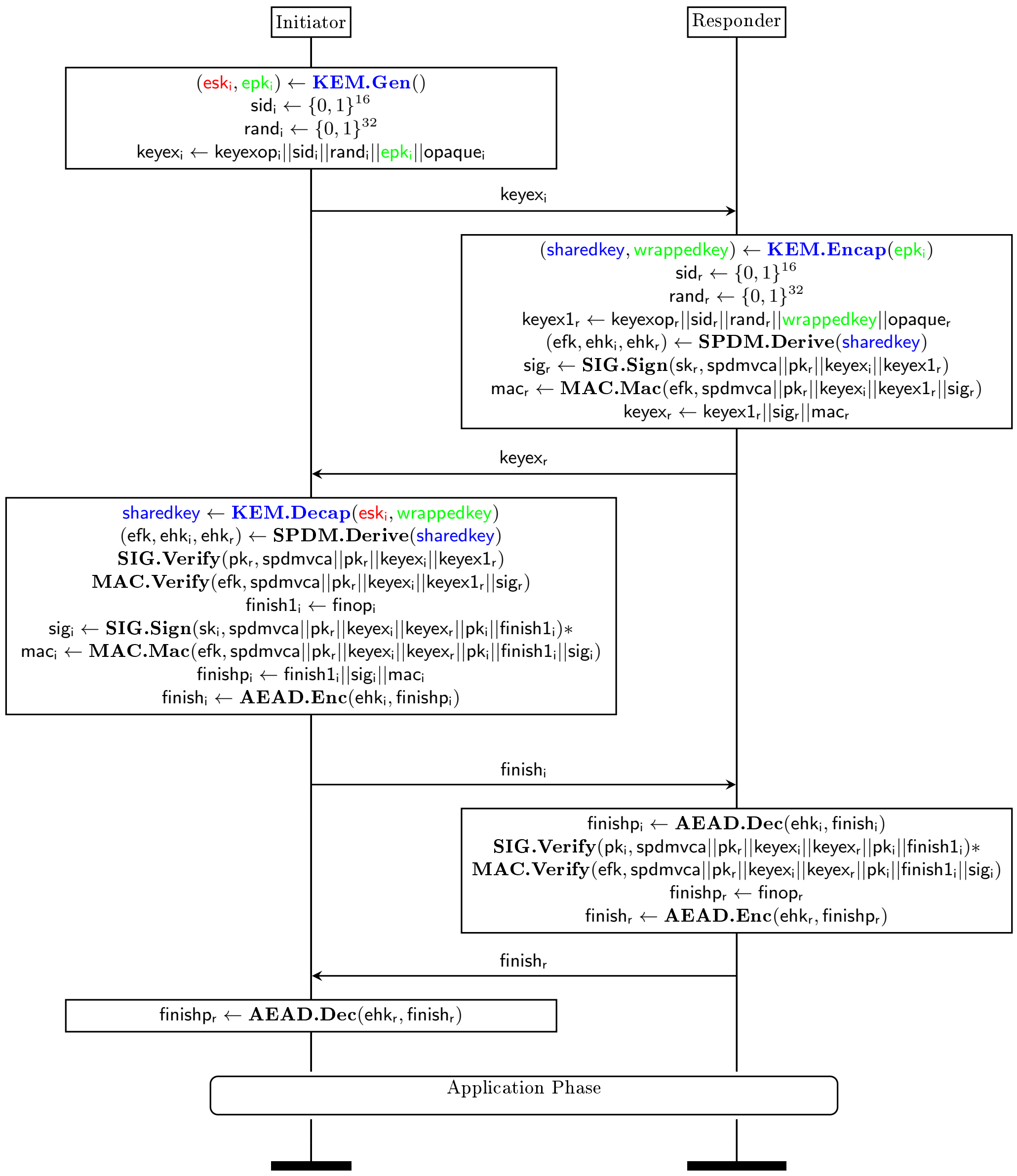
Initiator
Responder
$(\mathsf{esk_i}, \mathsf{epk_i}) \leftarrow \mathbf{KEM.Gen}()$
$\mathsf{sid_i} \leftarrow \{0,1\}^{16}$
$\mathsf{rand_i} \leftarrow \{0,1\}^{32}$
$\mathsf{keyex_i} \leftarrow \mathsf{keyexop_i}||\mathsf{sid_i}||\mathsf{rand_i}||\mathsf{epk_i}||\mathsf{opaque_i}$
$\mathsf{keyex_i}$
$(\mathsf{sharedkey}, \mathsf{wrappedkey}) \leftarrow \mathbf{KEM.Encap}(\mathsf{epk_i})$
$\mathsf{sid_r} \leftarrow \{0,1\}^{16}$
$\mathsf{rand_r} \leftarrow \{0,1\}^{32}$
$\mathsf{keyex1_r} \leftarrow \mathsf{keyexop_r}||\mathsf{sid_r}||\mathsf{rand_r}||\mathsf{wrappedkey}||\mathsf{opaque_r}$
$(\mathsf{efk}, \mathsf{ehk_i}, \mathsf{ehk_r}) \leftarrow \mathbf{SPDM.Derive}(\mathsf{sharedkey})$
$\mathsf{sig_r} \leftarrow \mathbf{SIG.Sign}(\mathsf{sk_r}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex1_r})$
$\mathsf{mac_r} \leftarrow \mathbf{MAC.Mac}(\mathsf{efk}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex1_r}||\mathsf{sig_r})$
$\mathsf{keyex_r} \leftarrow \mathsf{keyex1_r}||\mathsf{sig_r}||\mathsf{mac_r}$
$\mathsf{keyex_r}$
$\mathsf{sharedkey} \leftarrow \mathbf{KEM.Decap}(\mathsf{esk_i}, \mathsf{wrappedkey})$
$(\mathsf{efk}, \mathsf{ehk_i}, \mathsf{ehk_r}) \leftarrow \mathbf{SPDM.Derive}(\mathsf{sharedkey})$
$\mathbf{SIG.Verify}(\mathsf{pk_r}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex1_r})$
$\mathbf{MAC.Verify}(\mathsf{efk}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex1_r}||\mathsf{sig_r})$
$\mathsf{finish1_i} \leftarrow \mathsf{finop_i}$
$\mathsf{sig_i} \leftarrow \mathbf{SIG.Sign}(\mathsf{sk_i}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex_r}||\mathsf{pk_i}||\mathsf{finish1_i})*$
$\mathsf{mac_i} \leftarrow \mathbf{MAC.Mac}(\mathsf{efk}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex_r}||\mathsf{pk_i}||\mathsf{finish1_i}||\mathsf{sig_i})$
$\mathsf{finishp_i} \leftarrow \mathsf{finish1_i}||\mathsf{sig_i}||\mathsf{mac_i}$
$\mathsf{finish_i} \leftarrow \mathbf{AEAD.Enc}(\mathsf{ehk_i}, \mathsf{finishp_i})$
$\mathsf{finish_i}$
$\mathsf{finishp_i} \leftarrow \mathbf{AEAD.Dec}(\mathsf{ehk_i}, \mathsf{finish_i})$
$\mathbf{SIG.Verify}(\mathsf{pk_i}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex_r}||\mathsf{pk_i}||\mathsf{finish1_i})*$
$\mathbf{MAC.Verify}(\mathsf{efk}, \mathsf{spdmvca}||\mathsf{pk_r}||\mathsf{keyex_i}||\mathsf{keyex_r}||\mathsf{pk_i}||\mathsf{finish1_i}||\mathsf{sig_i})$
$\mathsf{finishp_r} \leftarrow \mathsf{finop_r}$
$\mathsf{finish_r} \leftarrow \mathbf{AEAD.Enc}(\mathsf{ehk_r}, \mathsf{finishp_r})$
$\mathsf{finish_r}$
$\mathsf{finishp_r} \leftarrow \mathbf{AEAD.Dec}(\mathsf{ehk_r}, \mathsf{finish_r})$
Application Phase

Fig. 1. PQ-SPDM KEM based key exchange flow

authentication code (MAC) for the transcript. The ephemeral handshake key will be used to AEAD the rest of messages in the handshake phase such as `FINISH` and `FINISH_RSP`.

6. In order to prevent man-in-the-middle-attacks, the responder shall create a transcript and sign the transcript with its permanent private key (sk_r) and generate the digital signature (sig_r). Per the SPDM specification, the transcript of `KEY_EXCHANGE_RSP` for signing is the concatenation of the negotiated SPDM protocol version, capability and algorithm ($spdmvca$), the permanent public certificate chain or public key of the responder as identity information (pk_r), the `KEY_EXCHANGE` message from initiator ($keyex_i$) and the response generated `KEY_EXCHANGER_RSP` message ($keyex1_r$).
7. The next step is to create another transcript and MAC the transcript with the ephemeral finish key (efk) and generate the MAC (mac_r). Per the SPDM specification, the transcript of `KEY_EXCHANGE_RSP` for MAC is the concatenation of the transcript of `KEY_EXCHANGE_RSP` for signing and the digital signature (sig_r).
8. The final full `KEY_EXCHANGE_RSP` message ($keyex_r$) is the concatenation of the response generated `KEY_EXCHANGER_RSP` message ($keyex1_r$), the digital signature (sig_r) and the MAC (mac_r).

Third Message: `FINISH` (initiator to responder)

1. Once the initiator receives the `KEY_EXCHANGE_RSP` message, it can use KEM algorithm decapsulate the cipher text with its ephemeral private key (esk_i) and generate the shared key.
2. The initiator can follow the SPDM defined key schedule algorithm to derive the ephemeral finish key (efk), initiator direction ephemeral handshake key (ehk_i) and responder direction ephemeral handshake key (ehk_r). These keys should be same as the one derived by the responder.
3. Then the requester shall follow the same process to construct the transcript of `KEY_EXCHANGE_RSP` for signing and verify the digital signature (sig_r) with the responder's permanent public key (pk_r). If the digital signature verification fails, then the initiator shall terminate the session handshake immediately.
4. The requester shall follow the same process to construct the transcript of `KEY_EXCHANGE_RSP` for MAC and verify the MAC (mac_r) with the ephemeral finish key (efk). If the MAC verification fails, then the initiator shall terminate the session handshake immediately.
5. At this point the initiator starts preparing the `FINISH` message ($finish1_i$) to close the handshake.
6. If mutual authentication is required, the initiator shall create the transcript of `FINISH` for signing, which is the concatenation of $spdmvca$, the permanent public certificate chain or public key of the responder as identity information (pk_r), the `KEY_EXCHANGE` message from initiator ($keyex_i$) and the response generated `KEY_EXCHANGER_RSP` message ($keyex_r$), the permanent public certificate chain or public key of the initiator as identity information (pk_i) and

the initiator generated FINISH message ($finish1_i$). The initiator needs to sign the transcript with its permanent private key (sk_i) and generate the digital signature (sig_i).

7. Then the initiator shall create the transcript of `FINISH` for MAC, which is the concatenation of the transcript of `FINISH` for signing and the digital signature (sig_i). The initiator needs to MAC the transcript with the ephemeral finish key (efk) and generate the MAC (mac_i).
8. The full `FINISH` message ($finishp_i$) is the concatenation of the requester generated `FINISH` message ($finish1_i$), the digital signature (sig_i) and the MAC (mac_i). The digital signature is absent if mutual authentication is not required.
9. The final `FINISH` message ($finish_i$) is the AEAD of the full `FINISH` message ($finishp_i$) with initiator direction handshake key (ehk_i).

Fourth Message: `FINISH_RSP` (responder to initiator)

Once the responder receives the `FINISH` message, it performs AEAD decryption and verifies the AEAD MAC.

1. If mutual authentication is required, the responder needs to verify the digital signature with the initiator's permanent public key (pk_i). If the digital signature verification fails, then the responder shall terminate the session handshake immediately.
2. The responder needs to verify the MAC with the ephemeral finish key (efk). If the MAC verification fails, then the responder shall terminate the session handshake immediately.
3. As the final step, the responder creates the `FINISH_RSP` message ($finishp_r$).
4. The final `FINISH_RSP` message ($finish_r$) is the AEAD of the responder generated `FINISH_RSP` message ($finishp_r$) with responder direction handshake key (ehk_r).

After the requester receives the `FINISH_RSP`, it performs AEAD decryption and verifies the AEAD MAC. If the AEAD MAC verification passes, then the secure session is setup between the initiator and the responder. The session application message after the handshake phase is unchanged, which uses the AEAD encryption.

In hybrid mode, there will be two shared secrets, including the traditional Diffie-Hellman ephemeral (DHE) secret and the PQC shared secret. We concatenate them together as the final SPDM key exchange shared secret and input it to the SPDM key schedule algorithm.

Note that in the SPDM specification, the SPDM key schedule algorithm names the SPDM key exchange shared secret to be "DHE Secret" because the traditional algorithms only support DHE or ECDHE. We use a new term "key exchange shared secret" to avoid any misunderstanding. In traditional mode, "key exchange shared secret" is "DHE Secret". In PQC mode, "key exchange shared secret" is "PQC shared secret". In hybrid mode, "key exchange shared secret" is "the concatenation of traditional DHE secret and PQC shared secret".

6.2 Security Analysis

KEM based key agreement In this proposal, we replaced the DHE based key agreement with a KEM based key agreement. This mechanism is similar to the key exchange model in TLS 1.3 hybrid design [47]. The KEM includes three parts.

- KEM.Gen()->(esk,epk): A probabilistic key generation algorithm. It generates an ephemeral public key (epk) and an ephemeral secret key (esk).
- KEM.Encap(epk)->(sharedkey, wrappedkey): A probabilistic encapsulation algorithm. It takes epk as input and outputs a shared secret (sharedkey) and a ciphertext pk wrapper (wrappedkey).
- KEM.Decap(esk, wrappedkey)->sharedkey: A decapsulation algorithm. It takes esk and wrappedkey as input and outputs a sharedkey.

Once the shared key is calculated, the remaining steps are the same as in the existing SPDM standard, such as key derivation with HMAC based key derivation function (HKDF), identity authentication using a digital signature, and the proof for the owner of session key via HMAC.

The required security property of a KEM scheme is indistinguishability under adaptive chosen ciphertext attack (IND-CCA2), which ensures security against an active attacker. All NIST PQC KEM finalists support IND-CCA2. However, two NIST PQC KEM alternate candidates, BIKE and SIDH, only support indistinguishability under chosen plaintext attack (IND-CPA), which means they can only guarantee security against passive attacker. The implementer should choose a proper KEM algorithm based on the security needs, we recommend ID-CCA2 schemes.

The existing SPDM only supports DHE. The DHE can be modeled as KEM, where

- KEM.Gen()->(esk,epk): To select an exponent x and calculate g^x, then esk $= x$, epk $= g^x$.
- KEM.Encap(epk)->(sharedkey, wrappedkey): To select an exponent y and calculate g^y and g^{x*y}, then sharedkey $= g^{x*y}$, wrappedkey $= g^y$.
- KEM.Decap(esk, wrappedkey)->sharedkey: To compute sharedkey $= g^{x*y}$.

The details of DH based KEM are described in RFC9180 [2].

SPDM standard recommends that the requester and responder should generate a fresh DHE key pair for each key exchange request or response. That means some key reuse might be possible. The implementer should understand the limitation of key use, such as forward secrecy, and understand the KEM algorithm specific requirement, such as number of reuses. For example, PQC SIDH is not secure when keys are reused. We recommend always using fresh, one time use key exchange values.

Shared Secret Combiner In hybrid mode, we calculate the $final\_shared\_secret$ to be the concatenation of DHE_shared_secret and $PQC\_KEM\_shared\_secret$,

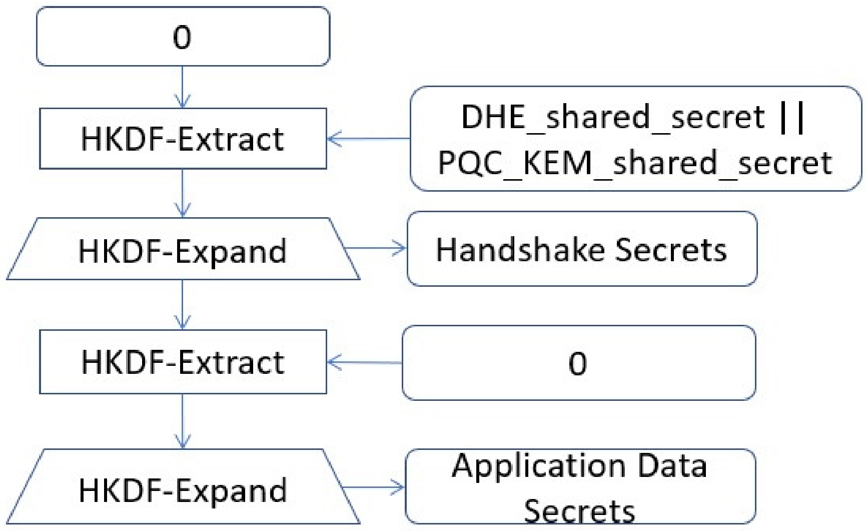

Fig. 2. SPDM Key Derivation Flow

followed by the HKDF-Extract and HKDF-Expand to derive the handshake secret and data secret. See Fig.2.

In the current SPDM and NIST PQC KEM algorithms, the length of $shared\_secret$ is fixed. As such, the length of $final\_shared\_secret$ is also fixed.

The $final\_shared\_secret$ shall have hybrid property: The secret is secure if at least one of the key exchange algorithm is secure. The analysis of KEM combiners were provided in [4, 19]. The construction follows the dual-PRF combiner which was approved to be IND-CCA secure in [4].

6.3 Design Considerations

Hybrid Key Exchange We require that a traditional algorithm and a PQC algorithm each generate key `ExchangeData` separately, then concatenate them together. We do not choose the option to define a new key exchange algorithm to combine both traditional and PQC algorithms. This aligns with the current practice in TLS hybrid mode.

No duplication In the hybrid mode, the traditional and PQC key `ExchangeData` are combined together. There is no need to send the `KEY_EXCHANGE` message twice - one is for classic mode and the other is PQC mode. The transcript calculation just needs to happen one time.

Hybrid Key Schedule We concatenate the raw traditional DHE secret and raw PQC KEM shared secret as the final key exchange shared secret.

We do not perform any other mathematical operation, such as XOR or Key Derivation Function (KDF) to process the shared secrets. This aligns with the current practice in TLS hybrid mode.

Transport Message Size As we discussed in previous section, there is a size limitation in the transport message. The public key and the cipher text of the PQC key establishment algorithm are involved in the transmission. The public key size and the cipher text size of BIKE, HQC, Kyber, NTRU-HPS, ntrulpr, Saber, FrodoKEM, SIDE/SIKE are larger than 256 bytes. The public key size of `classic-McEliece-{6688128,6960119,8192128}` exceeds 2^{16} bytes.

The current `KEY_EXCHANGE` request and response messages need to pass the key `ExchangeData`. In order to support large message transportation, we can use the generic SPDM 1.2 chunking message for them.

Timing As we discussed in a previous section, a PQC algorithm may have different KEM encapsulation timing requirements. Some isogenies of elliptic curves based algorithms such as SIKE are much slower than the lattices based or code based algorithms such as NTRU or Classic-McEliece. The `PQCTExponent` should be larger than the maximum encapsulation time for the supported KEM algorithms.

7 Results

We have developed a prototype that implements the above PQ-enabled SPDM variant using post-quantum algorithms library liboqs [31] on top of an SPDM implementation [1]. The implementation can run in both the Windows and Linux SPDM emulators. It can run in a Field Programmable Gate Array (FPGA) smart card and communicate with a host system on Intel Core CPU. We collected data for the winning PQC algorithms (Kyber as a KEM and DILITHIUM, FALCON, SPHINCS+ as digital signature primitives) described in the status report of NIST PQC 3rd round finalists [30] and compare them with the RSA and ECC in traditional mode and hybrid mode. For parameters, we choose NIST security level 1, 3 and 5 separately. The data has been collected on Intel Core(TM) i7-8665U CPU @ 1.90 GHz.

Usually, a device is an SPDM responder and a host operating system is an SPDM requester. We collect data from SPDM requester and responder separately. We considered two typical use cases: 1) device authentication which includes digital signature signing and verification, 2) one-way authenticated secure session establishment, which includes key establishment and digital signature. In the figure, we highlighted the time consuming portions - certificate verification (CERT_VERIFY) in `GET_CERTIFICATE`, digital signature signing (CHAL_SIGN) and verification (CHAL_VERIFY) in `CHALLENGE` for authentication, KEM generation (KEY_EX_KEM_GEN), encapsulation (KEY_EX_KEM_ENCAP), decapsulation (KEY_EX_KEM_DECAP) in `KEY_EXCHANGE`, and digital signature signing (KEY_EX_SIGN) and verification (KEY_EX_VERIFY) in `KEY_EXCHANGE` for authentication secure session setup.

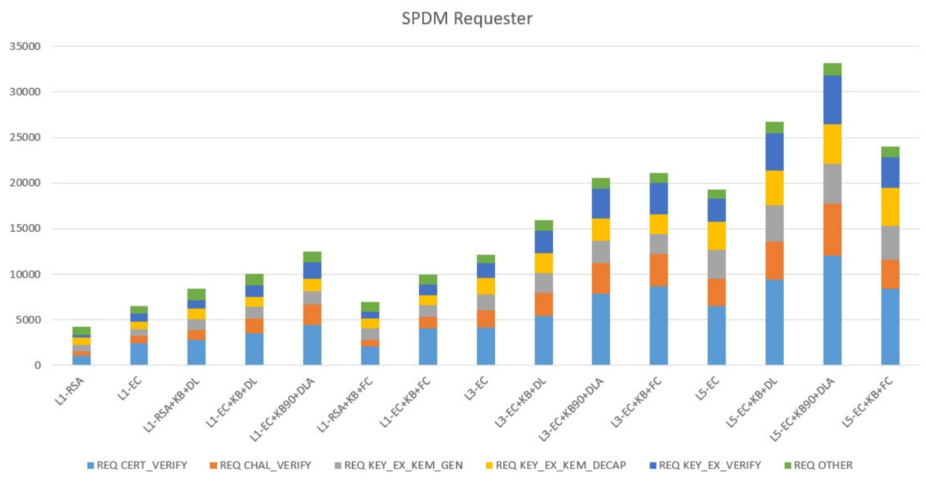

Fig. 3. SPDM Requester Performance - Kyber, DILITHIUM, FALCON (microsecond)

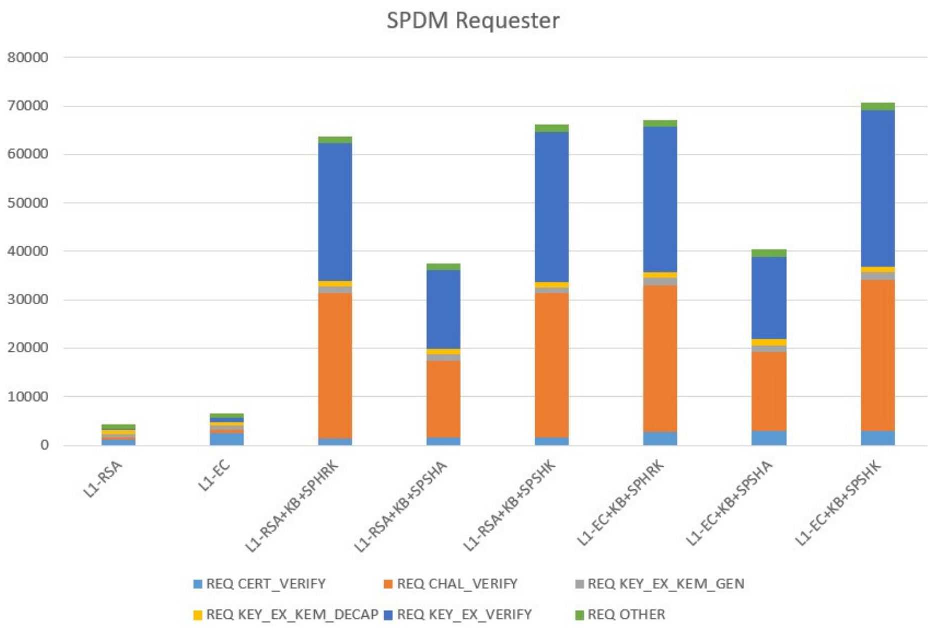

Fig. 4. SPDM Requester Performance - SPHINCS+ (microsecond)

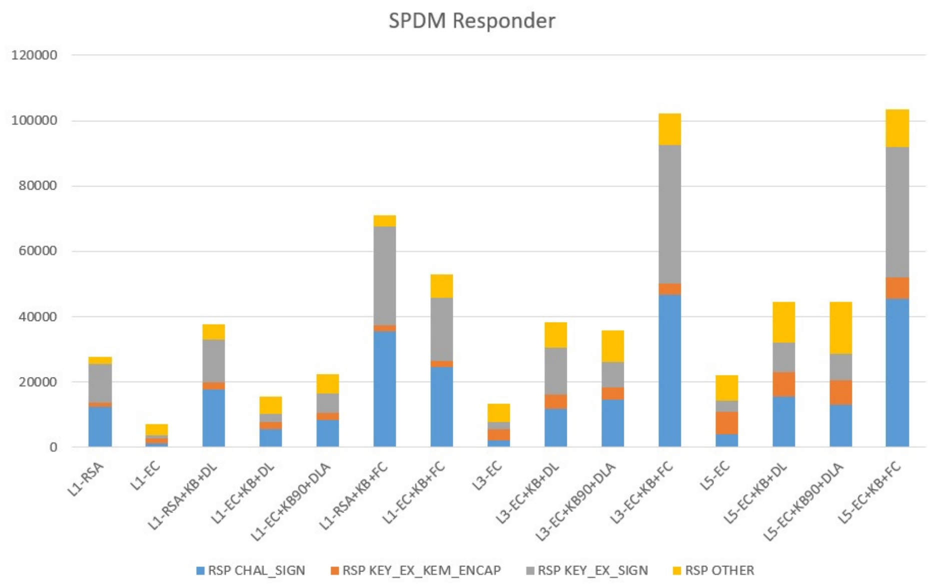

Fig. 5. SPDM Responder Performance - Kyber, DILITHIUM, FALCON (microsecond)

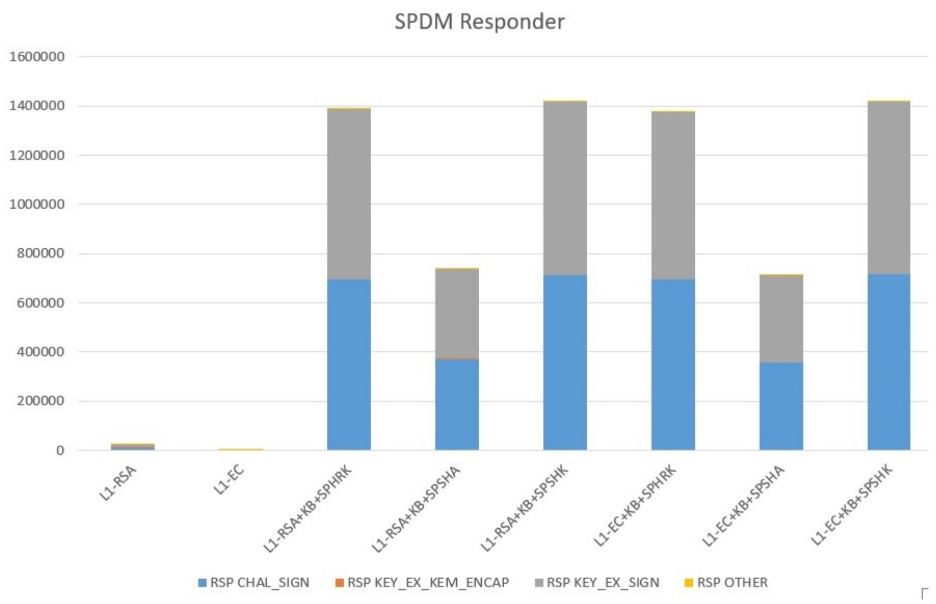

Fig. 6. SPDM Responder Performance - SPHINCS+ (microsecond)

Fig. 3 and Fig. 4 show the data for the SPDM requester. Fig. 5 and Fig. 6 show the data for the SPDM responder. L1, L3 and L5 means the minimal NIST security level for the combination. The algorithm list is below (note that for hybrid modes we have both classical and PQC algorithm suites):

- L1-RSA: `ECDHE_secp256r1` + `RSASSA_3072`
- L1-EC: `ECDHE_secp256r1` + `ECDSA_NIST_P256`
- L1-RSA+KB+DL: `ECDHE_secp256r1` & `Kyber512` + `RSASSA_3072` & `Dilithium2`
- L1-EC+KB+DL: `ECDHE_secp256r1` & `Kyber512` + `ECDSA_NIST_P256` & `Dilithium2`
- L1-EC+KB90+DLA: `ECDHE_secp256r1` & `Kyber512-90s` + `ECDSA_NIST_P256` & `Dilithium2-AES`
- L1-RSA+KB+FC: `ECDHE_secp256r1` & `Kyber512` + `RSASSA_3072` & `Falcon-512`
- L1-EC+KB+FC: `ECDHE_secp256r1` & `Kyber512` + `ECDSA_NIST_P256` & `Falcon-512`
- L1-RSA+KB+SPHRK: `ECDHE_secp256r1` & `Kyber512` + `RSASSA_3072` & `SPHINCS+-Haraka-128f-robust`
- L1-RSA+KB+SPSHA: `ECDHE_secp256r1` & `Kyber512` + `RSASSA_3072` & `SPHINCS+-SHA256-128f-robust`
- L1-RSA+KB+SPSHK: `ECDHE_secp256r1` & `Kyber512` + `RSASSA_3072` & `SPHINCS+-SHAKE256-128f-robust`
- L1-EC+KB+SPHRK: `ECDHE_secp256r1` & `Kyber512` + `ECDSA_NIST_P256` & `SPHINCS+-Haraka-128f-robust`
- L1-EC+KB+SPSHA: `ECDHE_secp256r1` & `Kyber512` + `ECDSA_NIST_P256` & `SPHINCS+-SHA256-128f-robust`
- L1-EC+KB+SPSHK: `ECDHE_secp256r1` & `Kyber512` + `ECDSA_NIST_P256` & `SPHINCS+-SHAKE256-128f-robust`
- L3-EC: `ECDHE_secp384r1` + `ECDSA_NIST_P384`
- L3-EC+KB+DL: `ECDHE_secp384r1` & `Kyber768` + `ECDSA_NIST_P384` & `Dilithium3`
- L3-EC+KB90+DLA: `ECDHE_secp384r1` & `Kyber768-90s` + `ECDSA_NIST_P384` & `Dilithium3-AES`
- L3-EC+KB+FC: `ECDHE_secp384r1` & `Kyber768` + `ECDSA_NIST_P384` & `Falcon-1024`
- L5-EC: `ECDHE_secp521r1` + `ECDSA_NIST_P521`
- L5-EC+KB+DL: `ECDHE_secp521r1` & `Kyber1024` + `ECDSA_NIST_P521` & `Dilithium5`
- L5-EC+KB90+DLA: `ECDHE_secp521r1` & `Kyber1024-90s` + `ECDSA_NIST_P521` & `Dilithium5-AES`
- L5-EC+KB+FC: `ECDHE_secp521r1` & `Kyber1024` + `ECDSA_NIST_P521` & `Falcon-1024`

8 Discussion

The data on the requester side shows that the cryptography timing caused by the hybrid mode is less than double of the traditional time. Kyber, DILITHIUM and FALCON all demonstrate good performance. The data on the responder side shows that the digital signature process contributed the majority of time. DILITHIUM is much better than FALCON. However, SPHINCS+ takes significantly longer time in both signature generation and verification.

Challenges The SPDM specification defines the digital signature data or the public key `ExchangeData` as fixed size fields. However, some PQC algorithms will generate a signature with variable size, such as FALCON. The actual size is smaller than the maximum size. The liboqs implementation uses the first four bytes to store the actual size. Care should be taken to move the variable signature data to or from a fixed size buffer to avoid buffer overflows.

If there is a transport message size limitation, we have to use chunking mechanism to transfer the large signature or public key. Chunking will increase the overhead on message disassembling and reassembling. However, even the transport layer has capability to transfer a large message, we may still consider chunking. The SPDM transport layer might not be reliable. As such, a package may be lost or broken during transmission, then a message retry maybe needed. The overhead to retry a large message is bigger than the one to retry a small chunked messaged. The implementer needs to balance the overhead of chunking and the overhead of message retry.

Backward Compatibility According to the previous discussion, the PQ-SPDM can achieve partial backward compatibility with SPDM 1.2 protocol. See Table 3.

9 Conclusions

In this paper, we proposed to a way to add PQC capability to the SPDM protocol. This addition entailed minimal modification of the existing SPDM protocol. We successfully created a prototype based upon the proposal. This prototype shows that PQC-SPDM is feasible in practice. We hope our analysis will help prepare the industry to adopt SPDM for device firmware with long-lived service requirements. This is imperative given the looming need to have more widespread support for post-quantum security capabilities. We notice that the latency caused by the digital signature signing on the responder side might become a performance concern in the future. The KEM based authentication [18] could be a way to resolve this timing problem. We will do the research on that area in the future.

References

1. post-quantum cryptography version openspdm. `https://github.com/jyao1/openspdm-pqc` (2022)
2. Barnes, R., Bhargavan, K., Lipp, B., Wood, C.A.: Hybrid Public Key Encryption. RFC 9180 (May 2022). https://doi.org/10.17487/RFC9180
3. Beullens, W.: Breaking rainbow takes a weekend on a laptop. Cryptology ePrint Archive, Report 2022/214 (2022), `https://ia.cr/2022/214`

Table 3. PQ-SPDM message compatibility

SPDM Message	**Changes for PQ-SPDM**	**Compatible?**
`GET_CAPABILITIES`, `CAPABILITIES`	1. Add extra capabilities (PQC, Hybrid mode) 2. May add extra field `PQCTExponent` as the timing required for PQC processing	YES
`NEGOTIATE_ALGORITHMS`, `ALGORITHMS`	1. Add PQC algorithms bits (PQC signature, PQC Key establishment)	YES
`GET_CERTIFICATE`, `CERTIFICATE`	1. May add certificate type (PQC, Hybrid). 2. May enlarge `PortionLength/RemainderLength` and `Offset/Length` field from 2 bytes to 4 bytes or retire the `CERTIFICATE` specific chunking.	NO
`CHALLENGE`, `CHALLENGE_AUTH`	1. Add PQC or Hybrid digital signature.	YES
`GET_MEASUREMENTS`, `MEASUREMENTS`	1. Add PQC or Hybrid digital signature.	YES
`KEY_EXCHANGE`, `KEY_EXCHANGE_RSP`	1. Add PQC or Hybrid key exchange. 2. Add PQC or Hybrid digital signature.	YES
`FINISH`, `FINISH_RSP`	1. Add PQC or Hybrid digital signature, in mutual authentication.	YES
`GET_CSR`, `CSR`	1. Add PQC or Hybrid digital signature.	YES
`SET_CERTIFICATE`, `CERTIFICATE`	1. May add certificate type (PQC, Hybrid).	YES
Secured Message Using SPDM	1. May enlarge `ApplicationDataLength` and `Length` field from 2 bytes to 4 bytes	NO

4. Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., Stebila, D.: Hybrid key encapsulation mechanisms and authenticated key exchange. In: Post-Quantum Cryptography. pp. 206–226. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-25510-7_12
5. Bindel, N., Herath, U., McKague, M., Stebila, D.: Transitioning to a quantum-resistant public key infrastructure. In: Post-Quantum Cryptography. pp. 384–405. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-59879-6_22
6. Campagna, M., Crockett, E.: Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS). Internet-Draft draft-campagna-tls-bike-sike-hybrid-07, Internet Engineering Task Force (Sep 2021), `https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid-07`, expired Draft
7. Castryck, W., Decru, T.: An efficient key recovery attack on sidh. Cryptology ePrint Archive, Report 2022/975 (2022), `https://eprint.iacr.org/2022/975`
8. Celi, S., Schwabe, P., Stebila, D., Sullivan, N., Wiggers, T.: KEM-based Authentication for TLS 1.3. Internet-Draft draft-celi-wiggers-tls-authkem-01, Internet Engineering Task Force (Mar 2022), `https://datatracker.ietf.org/doc/draft-celi-wiggers-tls-authkem`, work in Progress
9. Cho, J.Y., Sergeev, A.: Post-quantum MACsec in Ethernet Networks. J. Cyber Secur. Mobil. **10**(1), 161–176 (2021). https://doi.org/10.13052/jcsm2245-1439.1016

10. Crockett, E., Paquin, C., Stebila, D.: Prototyping post-quantum and hybrid key exchange and authentication in tls and ssh. NIST 2nd Post-Quantum Cryptography Standardization Conference, August, 2019 (2019), https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/stebila-prototyping-post-quantum.pdf
11. CXL: Compute Express Link homepage. https://computeexpresslink.org/ (2022)
12. DMTF: DSP0275 - Security Protocol and Data Model (SPDM) over MCTP Binding Specification. https://www.dmtf.org/dsp/DSP0275 (2019)
13. DMTF: DSP0237 - Management Component Transport Protocol (MCTP) SMBus/I2C Transport Binding Specification. https://www.dmtf.org/dsp/DSP0237 (2020)
14. DMTF: DSP0276 - Secured Messages using SPDM over MCTP Binding Specification. https://www.dmtf.org/dsp/DSP0276 (2020)
15. DMTF: DSP0277 - Secured Messages using SPDM specification. https://www.dmtf.org/dsp/DSP0277 (2020)
16. DMTF: Security Protocol and Data Model (SPDM) Specification, DSP0274. https://www.dmtf.org/dsp/DSP0274 (2021)
17. Fluhrer, S., Kampanakis, P., McGrew, D., Smyslov, V.: Mixing Preshared Keys in the Internet Key Exchange Protocol Version 2 (IKEv2) for Post-quantum Security. RFC 8784 (Jun 2020). https://doi.org/10.17487/RFC8784
18. Fujioka, A., Suzuki, K., Xagawa, K., Yoneyama, K.: Strongly secure authenticated key exchange from factoring, codes, and lattices. In: PKC'12: Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography. pp. 467–484. Springer International Publishing (2012). https://doi.org/10.1007/978-3-642-30057-8_28
19. Giacon, F., Heuer, F., Poettering, B.: KEM Combiners. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-76578-5_7
20. Hülsing, A., Ning, K.C., Schwabe, P., Weber, F., Zimmermann, P.R.: Post-quantum wireguard. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 304–321 (2021). https://doi.org/10.1109/SP40001.2021.00030
21. Kampanakis, P., Panburana, P., Curcio, M., Shroff, C.: Post-quantum hash-based signatures for secure boot. Cryptology ePrint Archive, Report 2020/1584 (2020), https://ia.cr/2020/1584
22. Kampanakis, P., Panburana, P., Curcio, M., Shroff, C., Alam, M.M.: Post-Quantum LMS and SPHINCS+ Hash-Based Signatures for UEFI Secure Boot. Cryptology ePrint Archive, Report 2021/041 (2021), https://ia.cr/2021/041
23. Kampanakis, P., Panburana, P., Daw, E., Geest, D.V.: The viability of post-quantum x.509 certificates. Cryptology ePrint Archive, Report 2018/063 (2018), https://ia.cr/2018/063
24. Kampanakis, P., Stebila, D., Friedl, M., Hansen, T., Sikeridis, D.: Post-quantum public key algorithms for the Secure Shell (SSH) protocol. Internet-Draft draft-kampanakis-curdle-pq-ssh-00, Internet Engineering Task Force (Oct 2020), https://datatracker.ietf.org/doc/html/draft-kampanakis-curdle-pq-ssh-00, expired Draft
25. Kiefer, F., Kwiatkowski, K.: Hybrid ECDHE-SIDH Key Exchange for TLS. Internet-Draft draft-kiefer-tls-ecdhe-sidh-00, Internet Engineering Task Force (Nov 2018), https://datatracker.ietf.org/doc/html/draft-kiefer-tls-ecdhe-sidh-00, expired Draft

26. Kniep, Q.M., Müller, W., Redlich, J.P.: Post-quantum cryptography in wireguard vpn. In: Security and Privacy in Communication Networks. pp. 261–267. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-63095-9_16
27. MIPI: MIPI home page. https://www.mipi.org/ (2022)
28. NIST: Post-Quantum Cryptography FAQs. https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs
29. NIST: Post-quantum cryptography project. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization (2016)
30. NIST: NIST IR 8413 - Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf (2022)
31. Open Quantum Safe Project: liboqs. https://github.com/open-quantum-safe/liboqs
32. Ounsworth, M.: Post-quantum Multi-Key Mechanisms for PKIX-like protocols; Problem Statement and Overview of Solution Space. Internet-Draft draft-pq-pkix-problem-statement-01, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-pq-pkix-problem-statement-01, work in Progress
33. Ounsworth, M., Pala, M.: Composite Signatures For Use In Internet PKI. Internet-Draft draft-ounsworth-pq-composite-sigs-06, Internet Engineering Task Force (Feb 2022), https://datatracker.ietf.org/doc/html/draft-ounsworth-pq-composite-sigs-06, work in Progress
34. PCI SIG: PCI Express Base Specification Revision 6.0. https://pcisig.com/specifications (2021)
35. PCI SIG: PCI-SIG homepage. https://pcisig.com/ (2022)
36. Pradel, G., Mitchell, C.J.: Post-quantum certificates for electronic travel documents. In: Computer Security: ESORICS 2020 International Workshops. pp. 56–73. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-66504-3_4
37. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. Quantum Info. Comput. **3**(4), 317–344 (jul 2003)
38. Raynal, M., Genet, A., , Romailler, Y.: PQ-WireGuard: we did it again. NIST Third PQC Standardization Conference (2021), https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/raynal-pq-wireguard-pqc2021.pdf
39. Roetteler, M., Naehrig, M., Svore, K.M., Lauter, K.: Quantum resource estimates for computing elliptic curve discrete logarithms. In: Advances in Cryptology – ASIACRYPT 2017. pp. 241–270. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-70697-9_9
40. Schanck, J.M., Stebila, D.: A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret. Internet-Draft draft-schanck-tls-additional-keyshare-00, Internet Engineering Task Force (Apr 2017), https://datatracker.ietf.org/doc/html/draft-schanck-tls-additional-keyshare-00, expired Draft
41. Schanck, J.M., Whyte, W., Zhang, Z.: Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2. Internet-Draft draft-whyte-qsh-tls12-02, Internet Engineering Task Force (Jul 2016), https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls12-02, expired Draft

42. Schwabe, P., Stebila, D., Wiggers, T.: Post-Quantum TLS Without Handshake Signatures, pp. 1461–1480. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3423350
43. Schwabe, P., Stebila, D., Wiggers, T.: More efficient post-quantum KEMTLS with pre-distributed public keys. In: Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12972, pp. 3–22. Springer (2021). https://doi.org/10.1007/978-3-030-88418-5_1
44. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134 (1994). https://doi.org/10.1109/SFCS.1994.365700
45. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (oct 1997). https://doi.org/10.1137/S0097539795293172
46. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020). https://doi.org/10.14722/ndss.2020.24203
47. Stebila, D., Fluhrer, S., Gueron, S.: Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-04, Internet Engineering Task Force (Jan 2022), `https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-04`, work in Progress
48. TCG: Tcg home page. `https://trustedcomputinggroup.org/` (2022)
49. Tjhai, C., Tomlinson, M., Bartlett, G., Fluhrer, S., Geest, D.V., Garcia-Morchon, O., Smyslov, V.: Multiple Key Exchanges in IKEv2. Internet-Draft draft-ietf-ipsecme-ikev2-multiple-ke-04, Internet Engineering Task Force (Sep 2021), `https://datatracker.ietf.org/doc/html/draft-ietf-ipsecme-ikev2-multiple-ke-04`, work in Progress
50. Truskovsky, A., Geest, D.V., Fluhrer, S., Kampanakis, P., Ounsworth, M., Mister, S.: Multiple Public-Key Algorithm X.509 Certificates. Internet-Draft draft-truskovsky-lamps-pq-hybrid-x509-01, Internet Engineering Task Force (Aug 2018), `https://datatracker.ietf.org/doc/html/draft-truskovsky-lamps-pq-hybrid-x509-01`, expired Draft
51. Whyte, W., Zhang, Z., Fluhrer, S., Garcia-Morchon, O.: Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3. Internet-Draft draft-whyte-qsh-tls13-06, Internet Engineering Task Force (Oct 2017), `https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls13-06`, expired Draft