PQUIP
Internet-Draft
Intended status: Informational
Expires: 28 September 2026

T. Reddy
Nokia
D. Wing
Citrix
B. Salter
UK National Cyber Security Centre
K. Kwiatkowski
PQShield

27 March 2026

Adapting Constrained Devices for Post-Quantum Cryptography
draft-ietf-pquip-pqc-hsm-constrained-04

Abstract

This document provides guidance on integrating Post-Quantum Cryptography (PQC) into resource-constrained devices, such as IoT nodes and lightweight Hardware Security Modules (HSMs). These systems often operate with strict limitations on processing power, RAM, and flash memory, and may even be battery-powered. The document emphasizes the role of hardware security as the basis for secure operations, supporting features such as seed-based key generation to minimize persistent storage, efficient handling of ephemeral keys, and the offloading of cryptographic tasks in low-resource environments. It also explores the implications of PQC on firmware update mechanisms in such constrained systems.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-pquip-pqc-hsm-constrained/.

Discussion of this document takes place on the pquip Working Group mailing list (mailto:pqc@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/pqc/. Subscribe at https://www.ietf.org/mailman/listinfo/pqc/.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Copyright Notice

Table of Contents

1. Introduction

The transition to post-quantum cryptography (PQC) poses significant challenges for resource-constrained devices, such as Internet of Things (IoT) devices, which are often equipped with Trusted Execution Environments (TEEs), secure elements, or other forms of hardware security modules (HSMs). These devices typically operate under strict limitations on processing power, RAM, and flash memory, and in some cases are battery-powered. Adopting PQC algorithms in such environments is difficult due to their substantially larger key sizes and, in some cases, higher computational demands. Consequently, the migration to PQC requires careful planning to ensure secure and efficient key management within constrained platforms.

Constrained devices are often deployed as clients initiating outbound connections, but some also act in server roles or enforce local authentication policies. As a result, designers may need to consider PQ solutions to address confidentiality, both outbound and inbound authentication, and signature verification used in secure boot, firmware updates, and device attestation.

This document provides guidance and best practices for integrating PQC algorithms into constrained devices. It reviews strategies for key storage, ephemeral key management, and performance optimization tailored to low-resource environments. The document also examines ephemeral key generation in protocols such as TLS, along with techniques to optimize PQC signature operations to improve performance within constrained cryptographic modules.

This document focuses on PQC algorithms standardized by NIST or specified by the IRTF CFRG, and that have corresponding IETF protocol specifications, either published as RFCs or progressing through the IETF standards process. Specifically, it covers the following algorithms:

* Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) [FIPS203].

* Module-Lattice-Based Digital Signature Algorithm (ML-DSA) [FIPS204].

* Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) [FIPS205].

* Hierarchical Signature System/Leighton-Micali Signature (HSS/LMS) [RFC8554], and the related eXtended Merkle Signature Scheme (XMSS) [RFC8391].

Additional post-quantum algorithms are expected to be standardised in future, which may also prove suitable for use in constrained devices. Since algorithms may change prior to standardisation (or may end up unstandardised), no concrete guidance is provided on these here, but future specifications may provide guidance on the following algorithms:

* The Falcon signature scheme [Falcon] has shorter keys and signatures than ML-DSA, though its use of floating point arithmetic may make it challenging to implement on some devices.

* The HQC KEM [HQC] is a code-based KEM, so offers algorithmic diversity to complement lattice-based KEMs, though it is less performant than ML-KEM.

* Smaller SLH-DSA parameter sets [Smaller-SPHINCS] may be standardised in future, which may make use of SLH-DSA more palatable on constrained devices.

This document focuses on device-level adaptations and considerations necessary to implement PQC efficiently on constrained devices. Actual protocol behaviour is defined in other documents.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Key Management in Constrained Devices for PQC

The embedded cryptographic components used in constrained devices are designed to securely manage cryptographic keys, often under strict limitations in RAM, flash memory, and computational resources. These limitations are further exhausted by the increased key sizes and computational demands of PQC algorithms.

One mitigation of storage limitations is to store only the seed rather than the full expanded private key, as the seed is far smaller and can derive the expanded private key as necessary. [FIPS204] Section 3.6.3 specifies that the seed 両 generated during ML-DSA.KeyGen can be stored for later use with ML-DSA.KeyGen_internal. To reduce storage requirements on constrained devices, private keys for Initial Device Identifiers (IDevIDs), Locally Significant Device Identifiers (LDevIDs), and the optional attestation private key can be stored as seeds instead of expanded key material.

3.1. Seed Management

To comply with [FIPS203], [FIPS204], [FIPS205] and [REC-KEM] guidelines:

3.1.1. Seed Storage

Several post-quantum algorithms use a seed to generate their private keys (e.g., ML-KEM and ML-DSA). Those seeds are smaller than private keys, hence some implementations may choose to retain the seed rather than the full private key to save on storage space. The private key can then be derived from the seed when needed or retained in a cache within the security module.

The seed is a Critical Security Parameter (CSP) as defined in [ISO19790], from which the private key can be derived, hence it must be safeguarded with the same level of protection as a private key. Seeds should be securely stored within a cryptographic module of the device whether hardware or software-based to protect against unauthorized access.

The choice between storing a seed or an expanded private key involves trade-offs between storage efficiency and performance. Some constrained cryptographic modules may store only the seed and derive the expanded private key on demand, whereas others may prefer storing the full expanded key to reduce computational overhead during key usage.

The choice between storing the seed or the expanded private key has direct implications on performance, as key derivation incurs additional computation. The impact of this overhead varies depending on the algorithm. For instance, ML-DSA key generation, which primarily involves polynomial operations using the Number Theoretic Transform (NTT) and hashing, is computationally efficient compared to other post-quantum schemes. In contrast, SLH-DSA key generation requires constructing a Merkle tree and multiple Winternitz One-Time Signature (WOTS+) key generations, making it significantly more computationally intensive. In many embedded deployments, SLH-DSA is expected to be used primarily for firmware verification, in which case key generation is performed offline and does not impact device performance. However, in scenarios where the device generates its own SLH-DSA key pairs, the higher key generation cost may influence seed-storage design decisions and depend on performance considerations or standards compliance (e.g., PKCS#11).

While vulnerabilities like the "Unbindable Kemmy Schmidt" misbinding attack [BIND] demonstrate the risks of manipulating expanded private keys in environments lacking hardware-backed protections, these attacks generally assume an adversary has some level of control over the expanded key format. However, in a hardware-backed protected environment, where private keys are typically protected from such manipulation, the primary motivation for storing the seed rather than the expanded key is not directly tied to mitigating such misbinding attacks.

The expanded private key is derived from the seed using a one-way cryptographic function. As a result, if the seed is not retained at key generation time, it cannot be reconstructed from the expanded key (as the reverse operation is computationally infeasible). Implementations should account for this non-recoverability when designing seed management.

A challenge arises when importing an existing private key into a system designed to store only seeds. When a user attempts to import an already expanded private key, there is a mismatch between the key format used internally (seed-based) and the expanded private key. This issue arises because the internal format is designed for efficient key storage by deriving the private key from the seed, while the expanded private key is already fully computed. As NIST has not defined a single private key format for PQC algorithms, this creates a potential gap in interoperability.

3.1.2. Efficient Key Derivation

When storing only the seed in a constrained cryptographic module, it is crucial that the device is capable of deriving the private key efficiently whenever required. However, repeatedly re-deriving the private key for every cryptographic operation may introduce significant performance overhead. In scenarios where performance is a critical consideration, it may be more efficient to store the expanded private key directly (in addition to the seed). Implementations may choose to retain (cache) several recently-used or frequently-used private keys to avoid the computational overhead and delay of deriving private keys from their seeds with each request.

The key derivation process, such as ML-KEM.KeyGen_internal for ML-KEM or similar functions for other PQC algorithms, must be implemented in a way that can securely operate within the resource constraints of the device. If using the seed-only model, the derived private key should only be temporarily held in memory during the cryptographic operation and discarded immediately after use. However, storing the expanded private key may be a more practical solution in time-sensitive applications or for devices that frequently perform cryptographic operations.

3.1.3. Exporting Seeds and Private Keys

Given the potential for hardware failures or the end-of-life of devices containing keys, it is essential to plan for backup and recovery of cryptographic seeds and private keys. Constrained devices should support secure seed- or key-backup mechanisms, leveraging protections such as encrypted storage and ensuring that security measures are in place so that the backup data is protected from unauthorized access.

There are two distinct approaches to exporting private keys or seeds from a constrained device:

3.1.3.1. Direct Transfer Over TLS

In scenarios where the constrained device supports mutually authenticated TLS with a peer, the device can securely transfer encrypted private key material directly to another cryptographic module over a mutually authenticated TLS connection.

3.1.3.2. Export of Encrypted Seeds and Private Keys

In more common constrained device scenarios for secure exporting of seeds and private keys, a strong symmetric encryption algorithm, such as AES in key-wrap mode ([RFC3394]), should be used to encrypt the seed or private key before export. This ensures that the key remains protected even if the export process is vulnerable to quantum attacks.

Operationally, the exported data and the symmetric key used for encryption must both be protected against unauthorized access or modification.

3.1.3.3. Security Requirements for Export Operations

The encryption and decryption of seeds and private keys must occur entirely within the cryptographic modules to reduce the risk of exposure and ensure compliance to established security standards.

3.2. Ephemeral Key Management

Given the increased size of PQC key material, ephemeral key management will have to be optimized for both security and performance.

For PQC KEMs, ephemeral key pairs are generated from an ephemeral seed, that is used immediately during key generation and then discarded. Furthermore, once the shared secret is derived, the ephemeral private key will have to be deleted. Since the private key resides in the constrained cryptographic module, removing it optimizes memory usage, reducing the footprint of PQC key material in the cryptographic module. This also ensures that that no unnecessary secrets persist beyond their intended use.

Additionally, ephemeral keys, whether from traditional ECDH or PQC KEM algorithms, are intended to be unique for each key exchange instance and kept separate across connections (e.g., TLS). Deleting ephemeral keying material after use helps ensure that key material cannot be reused across connections, which would otherwise introduce security and privacy issues.

Constrained devices implementing PQC ephemeral key management will have to:

* Generate ephemeral key pairs on-demand from an ephemeral seed stored temporarily within the cryptographic module.

* Enforce immediate seed erasure after the key pair is generated and the cryptographic operation is completed.

* Delete the private key after the shared secret is derived.

* Prevent key reuse across different algorithm suites or sessions.

4. Optimizing Memory Footprint in Post-Quantum Signature Schemes

A key consideration when deploying post-quantum cryptography in cryptographic modules is the amount and type of memory available. In constrained devices, it is important to distinguish between volatile memory (RAM), used for intermediate computations during cryptographic operations, and non-volatile storage (e.g., flash), used for storing keys, firmware, and configuration data. For instance, ML-DSA, unlike traditional signature schemes such as RSA or ECDSA, requires significant RAM during signing due to multiple Number Theoretic Transform (NTT) operations, matrix expansions, and rejection sampling loops. These steps involve storing large polynomial vectors and intermediate values, making ML-DSA more memory-intensive.

Some constrained systems, particularly battery-operated devices, may have limited RAM available for cryptographic operations, even if sufficient non-volatile storage is available. In such cases, straightforward implementations of PQ schemes may exceed available RAM, making them infeasible without optimization.

Several post-quantum schemes can be optimized to reduce the memory footprint of the algorithm. For instance, SLH-DSA has two flavours: the "f" variants which are parameterized to run as fast as possible, and the "s" variants which produce shorter signatures. Developers wishing to use SLH-DSA may wish to utilize the "s" variants on devices with insufficient RAM to use the "f" variants. Further optimizations may be possible by running the signature algorithm in a "streaming manner" such that constrained device does not need to hold the entire signature in memory at once, as discussed in [Stream-SPHINCS].

Implementations may trade off resource usage across CPU, RAM, and non-volatile storage. For example, techniques such as lazy expansion reduce RAM usage at the cost of increased computation, while storing expanded key in non-volatile storage can reduce runtime overhead. Designers should balance these trade-offs based on the target platform.

Both the ML-KEM and ML-DSA algorithms were selected for general use. Two optimization techniques that can be applied to make ML-DSA more feasible in constrained cryptographic modules are discussed in Section 4.1.1 and Section 4.2.

4.1. Memory requirements of Lattice-Based Schemes

The dominant source of memory usage in ML-DSA comes from holding the expanded matrix A and the associated polynomial vectors needed to compute the noisy affine transformation t = A*s1 + s2, where A is a large public matrix derived from a seed, and t, s1, s2 are polynomial vectors involved in the signing process. The elements of those matrices and vectors are polynomials with integer coefficients modulo Q. ML-DSA uses a 23-bit long modulus Q, where in case of ML-KEM it is 12 bits, regardless of security level. Conversely, the sizes of those matrices depend on the security level.

To compute memory requirements, we need to consider the dimensions of the public matrix A and the size of the polynomial vectors. Using ML-KEM-768 as an example, the public matrix A has dimensions 5x5, with each polynomial having 256 coefficients. Each coefficient is stored on 2 bytes (uint16), leading to a size of 5 _5_ 256 _2 = 12,800 bytes (approximately 12.5 KB) for the matrix A alone. The polynomial vectors t, s1, and s2 also contribute significantly to memory usage, with each vector requiring 5_ 256 _2 = 2,560 bytes (approximately 2.5 KB) each. Hence, for straightforward implementation, the minimal amount of memory required for these vectors is 12,800 + 3_ 2,560 = 20,480 bytes (approximately 20 KB). Similar computation can be easily done for other security levels as well as ML-DSA. The ML-DSA has much higher memory requirements due to larger matrix and polynomial sizes (i.e. ML-DSA-87 requires approximately 79 KB of RAM during signing operations).

It is worth noting that different cryptographic operations may have different memory requirements. For example, during ML-DSA verification, the memory usage is lower since the private key components are not needed.

4.1.1. Lazy Expansion as a Memory Optimization Technique

The lazy expansion technique is an optimization that significantly reduces memory usage by avoiding the need to store the entire expanded matrix A in memory at once. Instead of pre-computing and storing the full matrix, lazy expansion generates parts of it on-the-fly as needed for the process. This approach leverages the fact that not all elements of the matrix are required simultaneously, allowing for a more efficient use of memory.

As an example, we can look at the computation of matrix-vector multiplication t=A*s1. The matrix A is generated from a seed using a PRF, meaning that any element of A can be computed independently when needed. Similarly, the vector s1 is expanded from random seed and a nonce using a PRF.

The lazy expansion would first generate first element of a vector s1 (s1(0)) and then iterate over each row of matrix A in a first column. This approach generates partial result, that is a vector t. To finalize the computation of a vector t, the next element of s1 (s1(1)) is generated, and the process is repeated for each column of A until all elements of s1 have been processed. This method requires significantly less memory, in case of ML-KEM-768, size of element s1 (512 bytes) and a vector t (2560 bytes) is 256 _2 = 512 bytes, meaning that only 512 bytes + one row of matrix A (5_ 256 _2 = 2560 bytes) + one element of t (5_ 2 = 10 bytes) need to be stored in memory at any time, leading to a total of approximately 3 KB of memory usage, compared to the approximately 20 KB required for a straightforward implementation. The savings are even more pronounced for higher security levels, such as ML-DSA-87, where lazy expansion can reduce memory usage from approximately 79 KB to around 12 KB.

With lazy expansion, the implementation differs slightly from the straightforward version. Also, in some cases, lazy expansion may introduce additional computational overhead. Notably, applying it to ML-DSA signing operation may require to recompute vector y ([FIPS204], Algorithm 7, line 11) twice. In this case implementers need to weigh the trade-off between memory savings and additional computation.

This memory optimization was initially described in [Bot19]. Other optimizations can be found in [Gre20] and [BosRS22].

4.2. Pre-hashing as a Memory Optimization Technique

To address the memory consumption challenge, algorithms like ML-DSA offer a form of pre-hash using the 亮 (message representative) value described in Section 6.2 of [FIPS204]. The 亮 value provides an abstraction for pre-hashing by allowing the hash or message representative to be computed outside the cryptographic module. This feature offers additional flexibility by enabling the use of different cryptographic modules for the pre-hashing step, reducing memory consumption within the cryptographic module. The pre-computed 亮 value is then supplied to the cryptographic module, eliminating the need to transmit the entire message for signing. [RFC9881] discusses leveraging External亮-ML-DSA, where the pre-hashing step (External亮-ML-DSA.Prehash) is performed in a software cryptographic module, and only the pre-hashed message (亮) is sent to the hardware cryptographic module for signing (External亮-ML-DSA.Sign). By implementing External亮-ML-DSA.Prehash in software and External亮-ML-DSA.Sign in an hardware cryptographic module, the cryptographic workload is efficiently distributed, making it practical for high-volume signing operations even in memory-constrained cryptographic modules.

The main advantage of this method is that, unlike HashML-DSA, the External亮-ML-DSA approach is interoperable with the standard version of ML-DSA that does not use pre-hashing. This means a message can be signed using ML-DSA.Sign, and the verifier can independently compute 亮 and use External亮-ML-DSA.Verify for verification -- or vice versa. In both cases, the verifier does not need to know whether the signer used internal or external pre-hashing, as the resulting signature and verification process remain the same.

5. Optimizing Performance in PQC Signature Schemes

When implementing PQC signature algorithms in constrained cryptographic modules, performance optimization becomes a critical consideration. Transmitting the entire message to the cryptographic module for signing can lead to significant overhead, especially for large payloads. To address this, implementers can leverage techniques that reduce the data transmitted to the cryptographic module, thereby improving efficiency and scalability.

One effective approach involves sending only a message digest to the cryptographic module for signing. By signing the digest of the content rather than the entire content, the communication between the application and the cryptographic module is minimized, enabling better performance. This method is applicable for any PQC signature algorithm, whether it is ML-DSA, SLH-DSA, or any future signature scheme. For such algorithms, a mechanism is

often provided to pre-hash or process the message in a way that avoids sending the entire raw message for signing. In particular, algorithms like SLH-DSA present challenges due to

their construction, which requires multiple passes over the message digest during the signing process. The signer does not retain the entire message or its full digest in memory at once. Instead, different parts of the message digest are processed sequentially during the signing procedure. This differs from traditional algorithms like RSA or ECDSA, which allow for more efficient processing of the message, without requiring multiple passes or intermediate processing of the digest.

5.1. Impact of rejection sampling in ML-DSA Signing on performance

In constrained and battery-powered IoT devices that perform ML-DSA signing, the rejection-sampling loop introduces variability in signing latency and energy consumption due to the probabilistic nature of the signing process. While this results in a variable number of iterations in the signing algorithm, the expected number of retries for the standardized ML-DSA parameter sets is quantified below.

The analysis in this section follows the algorithmic structure and assumptions defined in [FIPS204]. Accordingly, the numerical results are analytically derived and characterize the expected behavior of ML-DSA.

The ML-DSA signature scheme uses the Fiat宴鈇hamir with Aborts construction [Lyu09]. As a result, the signature generation algorithm is built around a rejection-sampling loop. This section examines the rejection-sampling behavior of ML-DSA, as rejection sampling is not commonly used as a core mechanism in traditional digital signature schemes.

Rejection sampling is used to ensure that intermediate and output values follow the distributions required by the security proof. In particular, after computing candidate signature components, the signer checks whether certain norm bounds are satisfied. If any of these bounds are violated, the entire signing attempt is discarded and restarted with fresh randomness.

The purpose of rejection sampling is twofold. First, it prevents leakage of information about the secret key through out-of-range values that could otherwise bias the distribution of signatures. Second, it ensures that the distribution of valid signatures is statistically close to the ideal distribution assumed in the security reduction.

The number of rejections during signature generation depends on four factors:

* the message (i.e., the value of ホシ)

* the secret key material

* when hedged signing is used (see [FIPS204], Section 3.4), the random seed

* the context string (see [FIPS204], Section 5.2)

As a result, some message-key combinations may lead to a higher number of rejection iterations than others.

Using Equation (5) from [Li32] and assuming an RBG as specified in [FIPS204] (Section 3.6.1), the rejection probability during ML-DSA signing can be computed. These probabilities depend on the ML-DSA parameter set and are summarized below.

ML-DSA Variant	Acceptance Probability
ML-DSA-44	0.2350
ML-DSA-65	0.1963
ML-DSA-87	0.2596

Table 1: Acceptance probability - per-attempt probability of successful signing for the given ML-DSA variant.

Each signing attempt can be modeled as an independent Bernoulli trial: an attempt either succeeds or is rejected, with a fixed per-attempt acceptance probability. Under this assumption, the expected number of iterations until a successful signature is generated is the reciprocal of the acceptance probability. Hence, if r denotes the per-iteration rejection probability and p = 1 - r the acceptance probability, then the expected number of signing iterations is 1/p. Using this model, the expected number of signing attempts for each ML-DSA variant is shown below.

ML-DSA Variant	Expected Number of Attempts
ML-DSA-44	4.255
ML-DSA-65	5.094
ML-DSA-87	3.852

Table 2: Expected Number of Attempts for the given ML-DSA variant.

This model also allows computing the probability that the rejection-sampling loop completes within a given number of iterations. Specifically, the minimum number of iterations n required to achieve a desired completion probability can be computed as: n >= ln(1 - desired_probability) / ln(1 - p), where p is the per-iteration acceptance probability. For example, achieving a 99% probability of completing the signing process for ML-DSA-65 requires at most 21 iterations of the rejection-sampling loop.

Finally, based on these results, the cumulative distribution function (CDF) can be derived for each ML-DSA variant. The CDF expresses the probability that the signing process completes within at most a given number of iterations.

Iterations	ML-DSA-44	ML-DSA-65	ML-DSA-87
1	0.2350	0.1963	0.2596
2	0.4148	0.3541	0.4518
3	0.5523	0.4809	0.5941
4	0.6575	0.5828	0.6995
5	0.7380	0.6647	0.7775
6	0.7996	0.7305	0.8353
7	0.8467	0.7834	0.8780
8	0.8827	0.8259	0.9097
9	0.9103	0.8601	0.9331
10	0.9314	0.8876	0.9505
11	0.9475	0.9096	0.9634

Table 3: CDF values denote the probability of completing the signing process within the given number of iterations, for each ML-DSA variant.

The table Table 3 shows that while acceptance rate is relatively high for ML-DSA, the probability quickly grows with increasing number of iterations. After 11 iterations, each ML-DSA variant achieves over 90% probability of completing the signing process.

5.1.1. Practical Implications for Constrained Cryptographic Modules

As shown above, the rejection-sampling loop in ML-DSA signing leads to a probabilistic runtime with a geometrically distributed number of iterations. While the expected execution time is small, the tail of the distribution implies that, with low probability, a signing operation may require significantly more iterations than average. This unfavorable tail behavior represents a practical concern for ML-DSA deployments on constrained devices with limited execution capability and may require additional consideration.

As discussed in Section 3.1, in many deployment scenarios, constrained devices primarily perform signature verification, while signature generation is performed on more capable systems (e.g., firmware signing infrastructure). Therefore, the impact of rejection sampling is primarily relevant for devices that perform ML-DSA signing.

Devices that only verify signatures are not affected, as those operations do not involve rejection sampling and have deterministic execution times.

In firmware update and secure boot scenarios, signature verification is typically performed during early boot stages, where the bootloader has exclusive access to system resources. In such environments, the practical impact of resource constraints on signature verification is reduced compared to general runtime environments.

5.1.2. Suggestions for benchmarking ML-DSA Signing Performance

When benchmarking ML-DSA signing performance in constrained cryptographic modules, it is important to account for the probabilistic nature of the rejection-sampling loop. Reporting only a single timing measurement or a best-case execution time may lead to misleading conclusions about practical performance.

To provide a more comprehensive assessment of ML-DSA signing performance, benchmarks should report the following two metrics:

1. Single-iteration signing time: The signing time for a signature operation that completes within a single iteration of the rejection-sampling loop. This metric reflects the best-case performance of the signing algorithm and provides insight into the efficiency of the core signing operation without the overhead of repeated iterations.

2. Average signing time: The average signing time measured over a sufficiently large number of signing operations, using independent messages and, where applicable, independent randomness. Alternatively, an implementation MAY report the signing time corresponding to the expected number of iterations (see Table 2). This approach requires identifying a message, key, and randomness combination that results in the expected iteration count.

Libraries implementing ML-DSA should provide a mechanism to report the number of rejection-sampling iterations used during the most recent signing operation. This enables benchmarking tools to accurately compute average signing times across multiple signing operations.

6. Additional Considerations for PQC Use in Constrained Devices

6.1. Key Rotation and Renewal

In constrained devices, managing the lifecycle of cryptographic keys including periodic key rotation and renewal is critical for maintaining long-term security and supporting cryptographic agility. While constrained devices may rely on integrated secure elements or lightweight HSMs for secure key storage and operations, the responsibility for orchestrating key rotation typically resides in the application layer or external device management infrastructure.

Although the underlying cryptographic module may offer primitives to securely generate new key pairs, store fresh seeds, or delete obsolete keys, these capabilities must be integrated into the device’s broader key management framework. This process is especially important in the context of PQC, where evolving research may lead to changes in recommended algorithms, parameters, and key management practices.

The security of PQC schemes continues to evolve, with potential risks arising from advances in post-quantum algorithms, cryptanalytic or implementation vulnerabilities. As a result, constrained devices should be designed to support flexible and updatable key management policies. This includes the ability to:

* Rotate keys periodically to provide forward-secrecy,

* Update algorithm choices or key sizes based on emerging security guidance,

* Reconfigure cryptographic profile of the device via firmware updates.

6.2. Cryptographic Artifact Sizes for Post-Quantum Algorithms

The sizes of keys, ciphertexts, and signatures of post-quantum algorithms are generally larger than those of traditional cryptographic algorithms. This increase in size is a significant consideration for constrained devices, which often have limited memory and storage capacity. For example, the key sizes for ML-DSA and ML-KEM are larger than those of RSA or ECDSA, which can lead to

increased memory usage and slower performance in constrained environments.

The following table lists the sizes of cryptographic artifacts for representative instantiations of SLH-DSA and ML-KEM at NIST Security Level 1, as defined in [NISTSecurityLevels], ML-DSA at NIST Security Level 2, and HSS/LMS and XMSS at NIST Security Level 3; these are the lowest defined security levels for the respective schemes.

Algorithm	Type	Size (bytes)
ML-DSA-44	Public Key	1312
	Private Key	2560
	Signature	2420
SLH-DSA-SHA2-128s	Public Key	32

Table 4

	Private Key	64
	Signature	7856
SLH-DSA-SHA2-128f	Public Key	32

Table 5

	Private Key	64
	Signature	17088
LMS_SHA256_M24_H15_W4	Public Key	48
	Private Key	44
	Signature	2004
XMSS-SHA2_10_192	Public Key	48
	Private Key	104
	Signature	1492
ML-KEM-512	Public Key	800
	Private Key	1632
	Ciphertext	768
	Shared Secret	32
X25519	Public Key	32
	Private Key	32
	Shared Secret	32
Ed25519	Public Key	32
	Private Key	32
	Signature	64

Table 6

Corresponding sizes for higher security levels will typically be larger - see [FIPS203], [FIPS204], [FIPS205], [SP800-208] for sizes for all parameter sets.

7. Post-quantum Firmware Upgrades for Constrained Devices

Constrained devices deployed in the field require periodic firmware upgrades to patch security vulnerabilities, introduce new cryptographic algorithms, and improve overall functionality. However, the firmware upgrade process itself can become a critical attack vector if not designed to be post-quantum. If an adversary compromises the update mechanism, they could introduce malicious firmware, undermining all other security properties of the cryptographic modules. Therefore, ensuring a post-quantum firmware upgrade process is critical for the security of deployed constrained devices.

CRQCs pose an additional risk by breaking traditional digital signatures (e.g., RSA, ECDSA) used to authenticate firmware updates. If firmware verification relies on traditional signature algorithms, attackers could generate forged signatures in the future and distribute malicious updates.

7.1. Post-Quantum Firmware Authentication

To ensure the integrity and authenticity of firmware updates, constrained devices will have to adopt PQC digital signature schemes for code signing. These algorithms must provide long-term security, operate efficiently in low-resource environments, and be compatible with secure update mechanisms, such as the firmware update architecture for IoT described in [RFC9019].

[I-D.ietf-suit-mti] defines mandatory-to-implement cryptographic algorithms for IoT devices, and recommends use of HSS/LMS [RFC8554] to secure software devices. The SUIT working group may consider adding post-quantum algorithms, such as SLH-DSA and ML-DSA, in future specifications.

Stateful hash-based signature schemes, such as HSS/LMS or the similar XMSS [RFC8391], are good candidates for signing firmware updates. Those schemes offer efficient verification times, making them more practical choices for constrained environments where performance and memory usage are key concerns. Their security is based on the security of the underlying hash function, which is well-understood. A major downside of stateful hash-based signatures is the requirement to keep track of which One-Time Signature (OTS) keys have been reused, since reuse of a single OTS key allows for signature forgeries. However, in the case of firmware updates, the OTS keys will be signing versioned updates, which may make state management easier. [I-D.ietf-pquip-hbs-state] discusses various strategies for a correct state and backup management for stateful hash-based signatures.

Other post-quantum signature algorithms may also be viable for firmware signing:

* SLH-DSA, a stateless hash-based signature specified in [FIPS205], also has well-understood security based on the security of its underlying hash function, and additionally doesn't have the complexities associated with state management that HSS and XMSS have.

However, signature generation and verification are comparatively slow, and signature sizes are generally larger than other post-quantum algorithms. SLH-DSA's suitability as a firmware signing algorithm will depend on the capabilities of the underlying hardware.

* ML-DSA is a lattice-based signature algorithm specified in [FIPS204]. It is more performant than SLH-DSA, with significantly faster signing and verification times, as well as shorter signatures.

This will make it possible to implement on a wider range of constrained devices. The mathematical problem underpinning ML-DSA, Module Learning With Errors (M-LWE), is believed to be a hard problem by the cryptographic community, and hence ML-DSA is believed to be secure. Cryptographers are more confident still in the security of hash-based signatures than M-LWE, so developers may wish to factor that in when choosing a firmware signing algorithm.

7.2. Hybrid Signature Approaches

To enable secure migration from traditional to post-quantum security, PQ/T hybrid digital signature methods can be used for firmware authentication, combining a traditional and a post-quantum algorithm using either non-composite or composite constructions as defined in [RFC9794].

A non-composite approach, where both signatures are generated and carried separately, is simple to implement, requires minimal changes to existing signing, and aligns well with current secure boot and update architectures.

Composite constructions, which combine multiple algorithms into a single signature, require changes to cryptographic processing. In such constructions, the additional cost of including a traditional algorithm is typically small compared to the post-quantum component, but overall resource usage remains dominated by the post-quantum algorithm, particularly in terms of key size, signature size, code size, and verification cost.

Implementations should ensure that verification enforces the intended hybrid authentication property, namely that authentication remains secure as long as at least one component algorithm remains secure.

8. Security Considerations

The security considerations for key management in constrained devices for PQC focus on the secure storage and handling of cryptographic seeds, which are used to derive private keys. Seeds must be protected with the same security measures as private keys, and key derivation should be efficient and secure within resource-constrained cryptographic module. Secure export and backup mechanisms for seeds are essential to ensure recovery in case of hardware failure, but these processes must be encrypted and protected from unauthorized access.

8.1. Side Channel Protection

Side-channel attacks exploit physical leaks during cryptographic operations, such as timing information, power consumption, electromagnetic emissions, or other physical characteristics, to extract sensitive data like private keys or seeds. Given the sensitivity of the seed and private key in PQC key generation, it is critical to consider side-channel protection in cryptographic module design. While side-channel attacks remain an active research topic, their significance in secure hardware design cannot be understated. Cryptographic modules must incorporate strong countermeasures against side-channel vulnerabilities to prevent attackers from gaining insights into secret data during cryptographic operations.

9. Acknowledgments

Thanks to Jean-Pierre Fiset, Richard Kettlewell, Mike Ounsworth, Keegan Dasilva Barbosa, Hannes Tschofenig and Aritra Banerjee for the detailed review.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <https://www.rfc-editor.org/rfc/rfc3394>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9019] Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", RFC 9019, DOI 10.17487/RFC9019, April 2021, <https://www.rfc-editor.org/rfc/rfc9019>.

10.2. Informative References

[BIND] Schmieg, S., "Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK", April 2024, <https://eprint.iacr.org/2024/523.pdf>.

[BosRS22] Bos, J., Renes, J., and A. Sprenkels, "Dilithium for Memory Constrained Devices", December 2022, <https://eprint.iacr.org/2022/323.pdf>.

[Bot19] Botros, L., Kannwischer, M. J., and P. Schwabe, "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4", May 2019, <https://eprint.iacr.org/2019/489.pdf>.

[Falcon] Fouque, P.-A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., and Z. Zhang, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU", October 2020, <https://falcon-sign.info/falcon.pdf>.

[FIPS203] "Module-lattice-based key-encapsulation mechanism standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.203, August 2024, <https://doi.org/10.6028/nist.fips.203>.

[FIPS204] "Module-lattice-based digital signature standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.204, August 2024, <https://doi.org/10.6028/nist.fips.204>.

[FIPS205] "Stateless hash-based digital signature standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.205, August 2024, <https://doi.org/10.6028/nist.fips.205>.

[Gre20] Greconici, D., Kannwischer, M., and D. Sprenkels, "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4", Universitatsbibliothek der Ruhr-Universitat Bochum, IACR Transactions on Cryptographic Hardware and Embedded

Systems pp. 1-24, DOI 10.46586/tches.v2021.i1.1-24, December 2020, <https://doi.org/10.46586/tches.v2021.i1.1-24>.

[HQC] Gaborit et al, "Hamming Quasi-Cyclic (HQC)", August 2025, <https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf>.

[I-D.ietf-pquip-hbs-state] Wiggers, T., Bashiri, K., Kテカlbl, S., Goodman, J., and S. Kousidis, "Hash-based Signatures: State and Backup Management", Work in Progress, Internet-Draft, draft-ietf-pquip-hbs-state-04, 27 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-pquip-hbs-state-04>.

[I-D.ietf-suit-mti] Moran, B., Rテクnningstad, O., and A. Tsukamoto, "Cryptographic Algorithms for Internet of Things (IoT) Devices", Work in Progress, Internet-Draft, draft-ietf-suit-mti-23, 22 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-suit-mti-23>.

[ISO19790] ISO, "Information security, cybersecurity, and privacy protection 寰Security requirements for cryptographic modules", February 2025, <https://www.iso.org/standard/82423.html>.

[Li32] Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and D. Stehl辿, "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1)", February 2021, <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>.

[Lyu09] Lyubashevsky, V., "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures", Springer Berlin Heidelberg, Lecture Notes in Computer Science pp. 598-616, DOI 10.1007/978-3-642-10366-7_35, ISBN ["9783642103650", "9783642103667"], 2009, <https://doi.org/10.1007/978-3-642-10366-7_35>.

[NISTSecurityLevels]
NIST, "Post-Quantum Cryptography: Security (Evaluation Criteria)", January 2017, <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)>.

[REC-KEM] Alagic, G., Barker, E., Chen, L., Moody, D., Robinson, A., Silberg, H., and N. Waller, "Recommendations for key-encapsulation mechanisms", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.sp.800-227, September 2025, <https://doi.org/10.6028/nist.sp.800-227>.

[RFC8391] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/rfc/rfc8391>.

[RFC8554] McGrew, D., Curcio, M., and S. Fluhrer, "Leighton-Micali Hash-Based Signatures", RFC 8554, DOI 10.17487/RFC8554, April 2019, <https://www.rfc-editor.org/rfc/rfc8554>.

[RFC9794] Driscoll, F., Parsons, M., and B. Hale, "Terminology for Post-Quantum Traditional Hybrid Schemes", RFC 9794, DOI 10.17487/RFC9794, June 2025, <https://www.rfc-editor.org/rfc/rfc9794>.

[RFC9881] Massimo, J., Kampanakis, P., Turner, S., and B. E. Westerbaan, "Internet X.509 Public Key Infrastructure -- Algorithm Identifiers for the Module-Lattice-Based Digital Signature Algorithm (ML-DSA)", RFC 9881, DOI 10.17487/RFC9881, October 2025, <https://www.rfc-editor.org/rfc/rfc9881>.

[Smaller-SPHINCS]
Fluhrer, S. and Q. Dang, "Smaller Sphincs+ or, Honey, I Shrunk the Signatures", January 2024, <https://eprint.iacr.org/2024/018.pdf>.

[SP800-208]
Cooper, D., Apon, D., Dang, Q., Davidson, M., Dworkin, M., and C. Miller, "Recommendation for Stateful Hash-Based Signature Schemes", National Institute of Standards and Technology, DOI 10.6028/nist.sp.800-208, October 2020, <https://doi.org/10.6028/nist.sp.800-208>.

[Stream-SPHINCS]
Niederhagen, R., Roth, J., and J. W辰lde, "Streaming SPHINCS+ for Embedded Devices using the Example of TPMs", August 2021, <https://eprint.iacr.org/2021/1072.pdf>.

Authors' Addresses

Tirumaleswar Reddy
Nokia
Bangalore
Karnataka
India
Email: k.tirumaleswar_reddy@nokia.com

Dan Wing
Citrix
United States of America
Email: danwing@gmail.com

Ben Salter
UK National Cyber Security Centre
Email: ben.s3@ncsc.gov.uk

Kris Kwiatkowski
PQShield
Email: kris@amongbytes.com