



Post-Quantum Migration Playbook

Dr. Nadim Kobeissi
Symbolic Software



Foreword

Symbolic Software has been auditing cryptographic systems since 2017. Over more than 250 engagements with organisations including Mozilla, 1Password, Coinbase, Zoom, and the Linux Foundation, our work has converged on a single conviction: security begins with architecture, not patches. AI systems can now find and exploit implementation bugs at scale; the work that still requires human cryptographers is upstream of that — protocol design, cryptographic primitive selection, threat modelling, and formal verification.

The post-quantum migration is the largest design-level decision the industry has faced in a generation. The standards are finalised. The libraries are catching up. The regulatory clocks are running. This playbook collects, in one place, the decisions our clients keep asking us to help them make: which post-quantum primitive to choose, when to hybridise and when to stop, how to migrate Transport Layer Security (TLS) without breaking production, what the library landscape looks like today, how to test conformance, and which bug classes we keep finding in audits.

The recommendations here are opinionated where we think the right answer is clear, and explicit about uncertainty where it is not. They are calibrated against the asymmetry of consequences: the cost of migrating too early is engineering overhead; the cost of migrating too late is unrecoverable.

The post-quantum landscape changes quickly enough that any printed document is a snapshot. Always check for a more recent edition at <https://pq-migration.symbolic.software>, and [get in touch](#) if you would like our help applying any of this to your specific system.

Contents

1 How to use this playbook	1
1.1 Who this is for	1
1.2 Who this is not for	2
1.3 How the playbook is structured	2
1.4 How to read the callouts	3
1.5 The frame: risk asymmetry	4
1.6 What this playbook is not	4
2 The migration clock.....	5
2.1 Why migration is not optional	5
2.2 Don't wait for the factoring record	6
2.3 The risk asymmetry	7
2.4 The regulatory landscape	7
2.4.1 NIST: the standards.....	7
2.4.2 NSA: CNSA 2.0.....	8
2.4.3 Germany: BSI TR-02102	8
2.4.4 France: ANSSI	9
2.4.5 Other regimes	9
2.5 The timeline you actually care about	9
3 Choosing primitives	11
3.1 The shape of the choice	11
3.2 Choosing a KEM parameter set	12
3.3 Choosing a signature scheme	12
3.3.1 ML-DSA	12
3.3.2 SLH-DSA	13

3.3.3	When to use which	13
3.4	Hashes, AEAD, and symmetric primitives	14
4	Hybrid constructions	15
4.1	What a hybrid is, and what it buys you	15
4.2	Hybrid KEMs are not the same problem as hybrid signatures	15
4.3	The KEM combiner: X-Wing and the TLS group	17
4.4	Lattice confidence is earned, not assumed	17
4.5	The real risk is complexity	18
4.6	Pitfalls of hybrid construction	18
5	Migrating TLS and PKI	20
5.1	Where TLS migration is easy	20
5.2	Where TLS migration is hard	21
5.2.1	Certificate signatures	21
5.2.2	Certificate size	22
5.2.3	Certificate transparency	22
5.3	Downgrade protection	23
5.4	QUIC, DTLS, and other transports	23
5.5	What to do this quarter	24
6	Migrating secure messaging	25
6.1	Why messaging is different	25
6.2	One-to-one: PQXDH-style initial keys	25
6.2.1	Continuous ratchet PQ	26
6.3	Group messaging: MLS	26
6.4	Stored ciphertexts and key escrow	27
6.5	Pitfalls in messaging migration	27
7	The library landscape	29
7.1	What “ready” means	29
7.2	The Crucible-clean shortlist	30
7.3	Selected libraries by ecosystem	30
7.3.1	C / C++ ecosystem	30
7.3.2	Rust ecosystem	31

7.3.3	Go ecosystem	31
7.3.4	Java / .NET ecosystems	31
7.4	Choosing a library	32
8	Conformance testing	34
8.1	What Crucible tests	34
8.2	What we found	35
8.2.1	The implementations that passed cleanly	35
8.2.2	Conformance Finding 1: Missing encapsulation key modulus check ..	35
8.2.3	Conformance Finding 2: Bouncy Castle accepts wrong-length de- capsulation keys	36
8.3	How to wire up your implementation	36
8.4	Limits of conformance testing	37
8.5	Beyond Crucible	37
8.6	Get Crucible	38
9	Rollout strategy	39
9.1	Phases of a TLS hybrid rollout	39
9.2	What to monitor during rollout	40
9.2.1	Negotiation rate	40
9.2.2	Handshake latency	40
9.2.3	Handshake failures	40
9.2.4	Bytes on the wire	40
9.2.5	CPU and memory	40
9.2.6	Library- and platform-specific symptoms.....	40
9.3	Rollback	41
9.4	Operational pitfalls we have seen	41
9.5	Communication	42
9.6	Procurement leverage	42
10	Bug-class gallery	43
10.1	Primitive-level bug classes	43
10.2	Protocol-integration bug classes	45
10.3	Operational and key-management bug classes	45
10.4	Library and dependency bug classes	46
10.5	What to do with this chapter	47

11 About Symbolic Software	48
Bibliography.....	49
Appendix A Reference tables	51
A.1 NIST post-quantum standards (2024)	51
A.2 TLS named groups for hybrid PQ	51
A.3 Approximate primitive sizes	52
A.4 Useful pointers	52

List of Tables

3.1	ML-KEM parameter sets and their stated security category.	12
3.2	ML-DSA parameter sets, with approximate sizes.	13
3.3	Selected SLH-DSA parameter sets, illustrating the size range.	13
A.1	Finalised National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) standards and their status as of mid-2026.	51
A.2	Selected TLS 1.3 named groups for hybrid post-quantum key exchange. ...	51
A.3	Public-key, ciphertext / signature, and shared-secret sizes for selected primitives. “Approx.” values come from the standard documents; check the latest revision.	52

List of Acronyms

ACSC	Australian Cyber Security Centre	9
ANSSI	Agence nationale de la sécurité des systèmes d'information . . .	2
BSI	Bundesamt für Sicherheit in der Informationstechnik	2
CA	Certificate Authority	16
CNSA	Commercial National Security Algorithm Suite	2
CRQC	Cryptographically Relevant Quantum Computer	5
CT	Certificate Transparency	22
ECDH	Elliptic Curve Diffie-Hellman	6
FIPS	Federal Information Processing Standard	7
HNDL	Harvest Now, Decrypt Later	5
IETF	Internet Engineering Task Force	20
KAT	Known Answer Test	32
KDF	Key Derivation Function	15
KEM	Key Encapsulation Mechanism	9
KEX	Key Exchange	41
MLS	Messaging Layer Security	2
NCSC	National Cyber Security Centre	9
NIST	National Institute of Standards and Technology	vii
NSA	National Security Agency	2
PQC	Post-Quantum Cryptography	vii
PQ	Post-Quantum	9
TLS	Transport Layer Security	ii

How to use this playbook

TL;DR: Read this first

This playbook is organised around *decisions*, not theory. If you are migrating a real system, you can skip the chapters that are not in your scope. Each chapter ends with a short **Decision** block that summarises what we recommend, and a **From the audit floor** section with the specific bug classes we have seen in production.

1.1 | Who this is for

This playbook is written for the engineer or architect who has been told that their system needs to be “post-quantum ready” and now has to decide what that actually means. It assumes the reader is comfortable with TLS, modern AEAD, elliptic-curve Diffie-Hellman, and the difference between a key encapsulation mechanism and a signature scheme. It does not assume any prior background in lattice cryptography, and we will not be deriving any ring-LWE security reductions.

If you fit one or more of the following, this guide is for you:

- You maintain a TLS-terminating service and need to enable hybrid post-quantum key exchange.
- You operate a public-key infrastructure — a certificate authority, a software signing pipeline, a hardware attestation chain — and need a migration plan for signatures.
- You build secure messaging or end-to-end encrypted storage and need to understand what “post-quantum” means for ratcheting protocols and group messaging.
- You ship a cryptographic library or SDK and need to know which primitives, parameter sets, and hybrid constructions to expose to your users.

- You are responsible for compliance against National Security Agency (NSA) Commercial National Security Algorithm Suite (CNSA) 2.0, Bundesamt für Sicherheit in der Informationstechnik (BSI) TR-02102, Agence nationale de la sécurité des systèmes d'information (ANSSI) guidance, or any equivalent regulatory regime that mandates post-quantum migration on a published timeline.

1.2 | Who this is not for

We do not cover the underlying mathematics of lattice cryptography, the design of new schemes, or the choice between competing post-quantum candidates that did not make it through the NIST standardisation process. Those are good questions; they are not the questions production teams are asking us in 2026.

1.3 | How the playbook is structured

The chapters are intentionally short and scoped to a single migration concern. They are designed to be read out of order: jump to the chapter that maps to your immediate decision.

- Chapter 2 sets the regulatory clock — the deadlines from the NSA, BSI, ANSSI, and NIST that determine *when* you have to be done.
- Chapter 3 answers *which* primitive to choose — ML-KEM, ML-DSA, or SLH-DSA — and at which parameter set.
- Chapter 4 answers *whether to hybridise*, with a focus on the X25519 + ML-KEM-768 construction (X-Wing) that has emerged as the de facto industry default.
- Chapter 5 addresses TLS 1.3 specifically — handshake key shares, downgrade protection, and rollout strategy.
- Chapter 6 addresses secure messaging — ratcheting, group messaging (Messaging Layer Security (MLS)), and the new asymmetric primitives this exposes.
- Chapter 7 maps the current library landscape: BoringSSL, AWS-LC, OpenSSL, rustls, Go crypto/, libsodium, and more.
- Chapter 8 explains how to test a post-quantum implementation for conformance using *Crucible*, our open-source testing framework.
- Chapter 9 covers the operational playbook — canary, monitoring, and roll-back.

- Chapter 10 is a gallery of real bug classes we have found in production audits.

1.4 | How to read the callouts

Throughout this playbook, you will see four types of callout box. They are designed to be skimmable; if you only have ten minutes, read these.

Decision 1.1: An example decision

A green **Decision** block summarises what we recommend in a specific situation, and why. If our recommendation is contingent on something — a regulatory regime, a specific library version, a threat model assumption — the contingency will be stated explicitly here.

Pitfall 1.1: An example pitfall

A yellow **Pitfall** highlights a common mistake we see in audits. Pitfalls are not catastrophic on their own, but they accumulate, and they are the kind of thing that does not show up in a unit test.

From the audit floor 1.1: An example bug class

A red **From the audit floor** block describes a real bug class we have found in production code. Specific clients are not named; the bug class is the part that generalises.

TL;DR

A **TL;DR** block summarises a chapter or section in two or three sentences. Read this first to decide whether the surrounding text is relevant to you.

1.5 | The frame: risk asymmetry

The recommendations in this playbook do not rest on certainty about quantum timelines. They rest on the asymmetry of consequences. If migration turns out to be premature, the cost is engineering overhead — larger keys, larger signatures, more bandwidth, more complex protocol negotiations. These costs are real but recoverable. If migration is late, the consequences are categorically different: private keys become extractable, signatures become forgeable, and the encrypted traffic harvested today is decrypted retroactively. The first failure mode is engineering debt; the second is unrecoverable.

That asymmetry is the basis for the recommendation throughout this guide. Where the answer is clear, we say so directly. Where the trade-off is genuinely contested, we name what is being traded against what.

1.6 | What this playbook is not

This document is not a substitute for a security review. It is a map, not a guarantee. Specifically:

- It does not replace cryptographic protocol verification. If your protocol is non-standard, you should still model it formally; tooling like Verifpal, ProVerif, or Tamarin exists for exactly that purpose.
- It does not constitute a conformance certification. Even with this guide and our *Crucible* test framework, you should still have your final implementation independently audited before relying on it for production traffic.
- It does not cover every algorithm in the NIST PQC portfolio. We deliberately focus on the standards that have been finalised and on the constructions that have meaningful production deployment in 2026.

If you are unsure whether the choices in this playbook are right for your specific system — and especially if your system has a non-standard protocol, or a regulatory environment we do not cover — get in touch. The whole reason we wrote this is that the right answers depend heavily on context, and that context is the part you cannot get from a guide.

The migration clock

TL;DR

You have less time than you think, and the deadline you actually care about is probably set by your *regulator* or your *customer's compliance team*, not by NIST. Map your system to the right regulatory regime first; everything else follows from that. And do not wait for a public quantum factoring record to act — by the time you see one, you are already late.

2.1 | Why migration is not optional

Two forces are driving post-quantum migration. The first is the Harvest Now, Decrypt Later (HNDL) threat model: an adversary with the ability to record encrypted traffic today, and decrypt it later when a Cryptographically Relevant Quantum Computer (CRQC) becomes available. For traffic with long confidentiality requirements — legal communications, healthcare, intelligence, intellectual property, journalists' source material — this is already a deployed threat, regardless of whether the CRQC arrives in 2029, 2035, or never.

The second is regulation. Multiple national security agencies and standards bodies have published binding migration timelines, and major customers — particularly governments, financial institutions, and large enterprises — now require post-quantum readiness in their procurement and supplier-attestation processes. The migration is happening because procurement says it is happening.

2.2 | Don't wait for the factoring record

A common reasoning error in migration planning is to wait for a public demonstration that a CRQC has arrived. The argument runs: “no one has factored anything larger than 15; we have plenty of time.” This is precisely backwards.

Bas Westerbaan [1] and Sam Jaques, among others, have laid out the resource trajectory for Shor's algorithm against RSA-2048 and Elliptic Curve Diffie-Hellman (ECDH) on standard curves. The pattern they identify is consistent: by the time a quantum computer can factor a small but cryptographically embarrassing number — the first 32-bit factoring, say — the engineering distance to factoring 2048-bit RSA is no longer enormous. The same applies to elliptic-curve discrete logarithm. The visible milestone is a lagging indicator, not a leading one.

Scott Aaronson [2] has noted that this is similar to expecting an announcement before a small nuclear detonation. By the time the announcement arrives, the relevant decision was made years earlier. For migration planning, this means the threshold to start migration is not “a CRQC exists”; it is “a CRQC is plausible within the operational lifetime of systems being designed today.”

That threshold has now been crossed for many threat models. In late March 2026, a team led by Google Quantum AI [3] published updated quantum resource estimates for solving ECDLP! (ECDLP!) on the secp256k1 curve — the curve underpinning most cryptocurrency and a large share of TLS. The estimates compile to fewer than 1,500 logical qubits and under 90 million Toffoli gates, projected to execute on under 500,000 physical qubits in minutes under reasonable hardware assumptions. These numbers are roughly a 20x improvement over earlier published estimates. They do not prove that a CRQC will exist by any specific year, but they shift “within the operational lifetime of systems being designed today” from a worst-case to a median-case projection.

Decision 2.1: The threshold for action

You should be migrating now if any of the following apply:

- Your data has a confidentiality requirement that extends past 2030 (legal, medical, intelligence, intellectual property, source-material journalism).
- Your system signs artefacts that must remain valid for more than a decade (firmware, certificate authorities, software supply chain roots).
- Your customers, regulators, or procurement processes are starting to require post-quantum attestations.
- Your system has a multi-year deployment lifecycle and you would be unable to roll out a cryptographic migration in less than 12 months.

If *none* of these apply, you have more runway — but not as much as the absence of a factoring record might suggest. Plan accordingly.

2.3 | The risk asymmetry

The case for moving now does not rest on certainty about quantum timelines. It rests on the asymmetry of consequences.

If migration turns out to be premature — the quantum timeline is significantly longer than current estimates suggest — the cost is engineering overhead. Larger keys, larger signatures, more bandwidth, more complex protocol negotiations, and the implementation risks inherent in deploying newer primitives. These costs are real and non-trivial, but they are recoverable. A system designed with ML-KEM and ML-DSA that turns out not to have needed them for another fifteen years is a system that over-invested in security. There are worse failure modes.

If migration is late — the estimates are roughly correct and classical asymmetric cryptography becomes vulnerable within the operational lifetime of systems being designed today — the consequences are categorically different. Private keys are extractable. Signatures are forgeable. HNDL attacks become retroactively devastating. Data that has been exfiltrated cannot be un-exfiltrated.

That asymmetry is the basis for the recommendation in this playbook.

2.4 | The regulatory landscape

2.4.1 | NIST: the standards

The NIST PQC standardisation process produced three finalised Federal Information Processing Standard (FIPS) standards in August 2024:

- **FIPS 203 ML-KEM**: a key encapsulation mechanism based on module-lattice problems (originally CRYSTALS-Kyber). Standardised at three security categories: ML-KEM-512, ML-KEM-768, and ML-KEM-1024.
- **FIPS 204 ML-DSA**: a digital signature scheme based on module-lattice problems (originally CRYSTALS-Dilithium). Standardised at ML-DSA-44, ML-DSA-65, and ML-DSA-87.
- **FIPS 205 SLH-DSA**: a stateless hash-based signature scheme (originally SPHINCS+). Standardised in multiple parameter sets, with both SHA2 and SHAKE variants.

A fourth signature scheme, FN-DSA (originally Falcon), has been announced but is not yet finalised at the time of writing. NIST is also running a separate “signature on-ramp” competition for additional schemes; do not assume those are production-ready until they are finalised, peer-reviewed, and supported in the libraries you actually depend on.

NIST also publishes [SP 800-131A](#), the transition guidance document specifying when classical algorithms are deprecated and disallowed for federal use. This is the document that gives you the deprecation dates for RSA-2048, ECDSA-P-256, and so on. Check the latest revision; it changes.

2.4.2 | NSA: CNSA 2.0

The NSA’s CNSA 2.0 suite specifies the post-quantum algorithms required for U.S. national security systems. The high bits to know:

- CNSA 2.0 specifies ML-KEM-1024 for key establishment, not ML-KEM-768.
- CNSA 2.0 specifies ML-DSA-87 for general-purpose signatures, with LMS or XMSS for software/firmware signing where statefulness is acceptable.
- Hash functions: SHA-384 or SHA-512.
- Symmetric: AES-256.
- The transition timeline is staggered by use case (browsers and software signing first; networking and operating systems later). Treat these as deadlines, not aspirations.

If your system serves a U.S. government customer, CNSA 2.0 is binding, and the parameter sets are not negotiable. Notably, CNSA 2.0 *does not currently mandate hybrid* constructions — it specifies the post-quantum algorithm directly. This is a meaningful divergence from civilian best practice (see Chapter 4).

2.4.3 | Germany: BSI TR-02102

The German BSI publishes annual technical guidelines (TR-02102) listing the cryptographic mechanisms it considers fit for use. The BSI has historically been ahead of the curve on hybrid constructions and has explicitly recommended hybridised post-quantum key exchange for years. The guideline is updated annually; check the current revision.

2.4.4 | France: ANSSI

ANSSI has published positioning papers explicitly recommending *hybrid* post-quantum key exchange for a multi-year transition window. ANSSI's position on Key Encapsulation Mechanism (KEM)s is that pure post-quantum constructions should not yet be deployed without a classical fallback, on the grounds that the post-quantum schemes are too new to have accumulated the level of cryptanalytic confidence that warrants relying on them alone. This is the most cautious of the major regulatory positions. For KEMs during the transition window we agree with it; see Chapter 4 for why the same logic applies less forcefully to signatures.

2.4.5 | Other regimes

Other regulators (UK National Cyber Security Centre (NCSC), Australian Australian Cyber Security Centre (ACSC), EU regulatory frameworks, sectoral regulators in finance and healthcare) have either published their own guidance or aligned with one of the above. If you are subject to multiple regimes, build to the strictest superset.

2.5 | The timeline you actually care about

A well-known feature of regulatory timelines is that they tell you when you have to be *done*, not when you have to *start*. Working backwards from the strictest deadline you face:

- **Deployment** typically requires at least 6–12 months of canary, observation, and gradual rollout (Chapter 9).
- **Implementation and integration** of a Post-Quantum (PQ) migration in a non-trivial system typically takes 9–18 months once design is settled.
- **Design** — choosing primitives, parameter sets, hybrid constructions, and protocol changes — typically takes 3–6 months for a system of moderate complexity, longer if your protocol is non-standard.
- **Audit and conformance testing** (Chapter 8) adds another 2–4 months.

This adds up to roughly 18–36 months end-to-end. If you have a 2030 deadline and are starting in 2026, you are not early.

Pitfall 2.1: Treating “standardised” as “ready to deploy”

NIST finalising FIPS 203/204/205 in August 2024 is not the same as your library, your hardware, your protocol stack, and your operational tooling being

ready for production deployment. Library support has lagged finalisation by 6–18 months for most ecosystems. Plan around the readiness of your actual stack, not the standard’s publication date.

From the audit floor 2.1: Compliance theatre

We have audited multiple “post-quantum ready” products where the marketing claim was true in the narrowest sense — the codebase contained an ML-KEM implementation — but the actual TLS handshake, certificate chain, signing pipeline, or session-key derivation still used classical primitives end-to-end. Adding a primitive to a library is not the same as migrating a system. Verify the data path, not just the dependency graph.

Choosing primitives

TL;DR

For most civilian production systems: **ML-KEM-768** for key encapsulation and **ML-DSA-65** for signatures. Hybridise the KEM during the transition window where backward compatibility requires it (Chapter 4); use pure ML-DSA for signatures unless your specific use case has a long-lived signature-validity requirement. For NSA CNSA 2.0 environments: **ML-KEM-1024** and **ML-DSA-87**, without the hybrid (per current CNSA guidance). Reach for SLH-DSA only when you have a specific reason.

3.1 | The shape of the choice

NIST has standardised three post-quantum algorithms:

- One KEM (ML-KEM, [FIPS 203](#)).
- Two signature schemes (ML-DSA, [FIPS 204](#); SLH-DSA, [FIPS 205](#)).

For KEM, the only question is the parameter set. For signatures, you must additionally choose between a lattice-based scheme (ML-DSA) and a hash-based scheme (SLH-DSA), which have very different size and performance trade-offs.

3.2 | Choosing a KEM parameter set

ML-KEM is parameterised by a security category. The three options are:

Table 3.1: ML-KEM parameter sets and their stated security category.

Parameter set	NIST cat.	Public key	Ciphertext	Shared secret
ML-KEM-512	1	800 B	768 B	32 B
ML-KEM-768	3	1184 B	1088 B	32 B
ML-KEM-1024	5	1568 B	1568 B	32 B

The security categories correspond, roughly, to the cost of the best known attack matching the cost of brute-forcing AES-128 (cat. 1), AES-192 (cat. 3), and AES-256 (cat. 5).

Decision 3.1: ML-KEM parameter selection

For most civilian production systems where you can afford the wire bytes, choose **ML-KEM-768**. It matches the AES-192 security level, has industry consensus, and is what major deployments (Cloudflare, Google, AWS) have settled on.

For CNSA 2.0 environments, choose **ML-KEM-1024** — it is the only parameter set CNSA 2.0 currently approves.

Avoid **ML-KEM-512** unless you have an extreme size or compute constraint. The bytes saved are not worth the smaller security margin against future crypt-analytic improvements.

3.3 | Choosing a signature scheme

The signature decision is harder than the KEM decision because there is a real choice between schemes, not just between parameter sets.

3.3.1 | ML-DSA

ML-DSA is the lattice-based signature scheme. Its parameter sets are:

ML-DSA is fast — both signing and verification are competitive with classical signatures — but the key and signature sizes are much larger than Ed25519 (32 B / 64 B) or ECDSA-P256 (33 B compressed or 65 B uncompressed; 64–72 B DER signature). For protocols where signature size matters at scale (X.509 certificates, certificate transparency logs, signed software updates), this is the dominant cost.

Table 3.2: ML-DSA parameter sets, with approximate sizes.

Parameter set	NIST cat.	Public key	Signature
ML-DSA-44	2	1312 B	2420 B
ML-DSA-65	3	1952 B	3309 B
ML-DSA-87	5	2592 B	4627 B

3.3.2 | SLH-DSA

SLH-DSA is the hash-based signature scheme. Its security relies only on the security of an underlying hash function (SHA2 or SHAKE), which makes it the most cryptographically conservative choice in the NIST portfolio. It is also the slowest and produces the largest signatures.

Table 3.3: Selected SLH-DSA parameter sets, illustrating the size range.

Parameter set	NIST cat.	Public key	Signature (approx.)
SLH-DSA-128s	1	32 B	7856 B
SLH-DSA-128f	1	32 B	17088 B
SLH-DSA-192s	3	48 B	16224 B
SLH-DSA-256s	5	64 B	29792 B

The *s* (small) variants minimise signature size at the cost of much slower signing; the *f* (fast) variants are the reverse. Verification is reasonably fast in either case.

3.3.3 | When to use which

Decision 3.2: Signature scheme selection

Default to ML-DSA-65 (or ML-DSA-87 for CNSA 2.0). It is the right choice for almost every general-purpose signature use case — TLS certificates, JWT-style tokens, attestations, code signing for general software, signed messages.

Choose SLH-DSA when you specifically need conservative cryptographic assumptions: long-lived root signing keys, firmware signing for hardware that may be deployed for decades, or any context where you are willing to pay 5–10x the signature size to depend only on hash-function security.

Wait on FN-DSA (Falcon) until it is finalised, library support stabilises, and the implementation pitfalls — particularly the floating-point requirements — are well understood.

3.4 | Hashes, AEAD, and symmetric primitives

Post-quantum migration is mostly about asymmetric cryptography. Symmetric primitives are largely unaffected — Grover’s algorithm gives a quadratic speedup against symmetric search, which is addressed by doubling the key size.

- **AES-256** is post-quantum acceptable. AES-128 is generally considered acceptable too, but if you have the option, AES-256 gives you margin.
- **SHA-256, SHA-384, SHA-512** are post-quantum acceptable. SHA-3 likewise.
- **ChaCha20-Poly1305** is post-quantum acceptable. The 256-bit key is already at the right level.
- **HKDF, HMAC** with SHA-256 or larger are post-quantum acceptable.

Pitfall 3.1: Forgetting to upgrade adjacent primitives

CNSA 2.0 specifies SHA-384/512 and AES-256 alongside ML-KEM-1024 / ML-DSA-87. We have audited systems where the post-quantum primitives were correctly chosen but the symmetric primitives were left at AES-128 or SHA-256. If you are migrating for compliance, migrate the whole suite.

From the audit floor 3.1: Mixing parameter sets across the protocol

We have seen implementations use ML-KEM-768 for the handshake but derive session keys with HKDF-SHA-256 truncated to 128 bits, undoing the security category lift. The KEM security category is only as good as the weakest link in the key derivation chain. Match your KDF output and AEAD key length to your stated security category.

Hybrid constructions

TL;DR

Design new systems as **post-quantum native**: ML-KEM for key encapsulation, ML-DSA for signatures. Use hybrid *KEMs* (X-Wing, X25519MLKEM768) where backward compatibility with classical peers is required during transition — TLS is the canonical example. Hybrid *signatures* are not a sound default; reach for them only in specific use cases such as long-lived code signing or root keys whose validity must span decades.

4.1 | What a hybrid is, and what it buys you

A hybrid construction combines a classical asymmetric primitive (e.g. X25519) with a post-quantum primitive (e.g. ML-KEM-768) such that the resulting construction is at least as secure as the stronger of the two. Concretely: an attacker who breaks only X25519 cannot compromise the hybrid, and an attacker who breaks only ML-KEM cannot compromise the hybrid. Both must be broken for the hybrid to fall.

The mechanism is straightforward. For KEM hybrids, both KEMs produce shared secrets, and the secrets are combined — typically via a Key Derivation Function (KDF) bound to a transcript — to derive the final session key. For signature hybrids, two signatures are produced and both must verify.

4.2 | Hybrid KEMs are not the same problem as hybrid signatures

The most consequential design distinction in this chapter is one the industry has often blurred: hybrid KEMs and hybrid signatures hedge against *different* risks, and the urgency of the hedge is not the same in the two cases.

Hybrid KEMs defend against the HNDL threat. An adversary who records ciphertexts today can store them indefinitely and decrypt them once a CRQC is available.

Data that is confidential in 2026 must remain confidential in 2036. A hybrid KEM ensures that an attacker must break *both* the lattice problem and the elliptic-curve discrete logarithm to recover plaintext — if ML-KEM is later found flawed, X25519 is still there; if X25519 falls to a future quantum adversary, ML-KEM is still there. The cost is modest: a slightly larger key exchange and a second KDF call. The insurance is real.

Hybrid signatures hedge against a different threat. A signature is verified at the time it is received; an adversary who arrives in 2036 cannot retroactively forge a signature that was verified and acted upon in 2026. There is no stockpile of signatures waiting to be broken. The threat model that justifies hybrid KEMs — store now, exploit later — does not apply in the same way to signatures.

Hybrid signatures are not without merit. Long-lived signatures on firmware, software updates, or root certificates could in principle be forged by a future quantum adversary, and there is a reasonable belt-and-suspenders argument for hedging in the authentication context for these specific cases. But the case is less urgent than for KEMs, and the integration cost is the same.

Decision 4.1: Hybrid KEMs vs. hybrid signatures

For KEMs, hybridise during the transition where backward compatibility with classical peers is required. The X25519 + ML-KEM-768 hybrid (X-Wing, or the X25519MLKEM768 TLS named group) is the production default and is appropriate for almost every internet-facing system through the migration window. For signatures, do not hybridise by default. Use ML-DSA. Reach for a hybrid signature only in specific cases:

- Code signing for hardware whose deployed lifetime exceeds the migration window (firmware, automotive, industrial).
- Root Certificate Authority (CA) keys whose signatures must remain valid for decades.
- Specific regulatory regimes that mandate it.

For everything else — TLS server certificates, JWTs, attestations, ordinary code signing — pure ML-DSA is the right answer. The complexity of dual-signature schemes is real, and the threat it hedges against is meaningfully smaller than the HNDL threat that justifies hybrid KEMs.

4.3 | The KEM combiner: X-Wing and the TLS group

The dominant hybrid KEM construction in production today combines X25519 (an elliptic-curve Diffie-Hellman key exchange) with ML-KEM-768. There are two specifications worth knowing:

- **X-Wing**: a clean, IETF-track combiner construction [4] that takes the X25519 shared secret, the ML-KEM-768 shared secret, and a transcript binding, and feeds them through a KDF to produce the final secret. X-Wing is designed to be IND-CCA secure as a hybrid KEM and is appropriate for non-TLS protocols that need a generic hybrid.
- **TLS X25519MLKEM768**: the TLS-specific named group, which combines the two secrets within the TLS 1.3 key schedule rather than as a generic KEM. This is what is actually deployed on the public internet today.

Decision 4.2: Which combiner

For TLS 1.3, use the standard X25519MLKEM768 named group. Do not invent your own combiner.

For non-TLS protocols where you need a generic hybrid KEM, use **X-Wing** where library support exists. If you must roll your own combiner, follow a published construction, include both ciphertexts and both public keys in the binding string fed to the KDF, and have it reviewed before shipping.

4.4 | Lattice confidence is earned, not assumed

A recurring objection to pure post-quantum deployment is that lattice-based cryptography is “too new” to trust without a classical fallback. The numbers do not bear this out. NTRU was patented in 1997. The learning-with-errors problem was introduced by Regev in 2005, a year before Curve25519 was published in 2006. ML-KEM and ML-DSA survived nearly a decade of the most intensive public cryptanalysis effort in the field’s history through the NIST PQC process. SIKE, the supersingular-isogeny candidate, was broken during the competition by Castryck and Decru in 2022; the lattice candidates were not. That outcome is evidence of scrutiny that produced a result, not absence of scrutiny.

Symbolic Software’s own conformance testing [5] of fifteen ML-KEM and ML-DSA implementations across five languages found two minor conformance gaps and zero security vulnerabilities (Chapter 8). The implementations shipping inside AWS-LC, Cloudflare CIRCL, the Go standard library, and wolfSSL’s FIPS module all passed

every test. The post-quantum ecosystem is not speculative anymore; it is production-grade software that has been tested, audited, and deployed at scale.

This is the basis of the recommendation [6, 7] to design new systems as post-quantum native. The hybrid KEM is a sensible transitional safety blanket, particularly where backward compatibility forces it. It is not a permanent end state.

4.5 | The real risk is complexity

Our audit practice keeps producing the same lesson: the most common source of cryptographic failure is not a broken primitive. It is the complexity surrounding the primitive — the state machine that manages keys, the serialisation layer that encodes messages, the fallback logic that negotiates which algorithm to use.

Every hybrid construction doubles the number of moving parts in exactly these layers. A hybrid KEM requires two key generations, two encapsulations, two decapsulations, and a combiner that must correctly preserve the security properties of both components. A hybrid signature requires two signing operations, two verification operations, and a composition that must not introduce verification-oracle attacks.

For KEMs, the HNDL threat justifies this cost. For signatures, the same complexity is added without the same threat to hedge against. That is the asymmetry to keep in mind whenever someone proposes a four-algorithm dual-signature scheme as the new normal.

4.6 | Pitfalls of hybrid construction

Pitfall 4.1: Concatenate-without-bind

$\text{KDF}(X25519_ss \parallel MLKEM_ss)$ without binding the public keys and ciphertexts of both KEMs into the KDF input has known weaknesses against malicious peers. Always bind the transcript.

Pitfall 4.2: Truncating shared secrets

Both X25519 and ML-KEM produce 32-byte shared secrets. A common implementation mistake is to truncate one of them on the assumption it is “the same length” as the classical one. Use the KDF to derive the session key length you actually need.

Pitfall 4.3: “Hybrid” that XORs the two secrets

We have seen $session_key = X25519_ss \oplus MLKEM_ss$. This is not a hybrid combiner. It is at most as strong as the weaker secret. Use a KDF.

Pitfall 4.4: Treating hybrid as a permanent design point

The point of a hybrid KEM is to bridge a transition. It is not a long-term security argument that doubles your trust budget. Plan the deprecation path to pure post-quantum at the start, not at the end. A hybrid that becomes load-bearing forever is a system that will carry classical-side complexity long after it has stopped earning its keep.

From the audit floor 4.1: Selective failure on the post-quantum half

We have audited implementations where a malformed ML-KEM ciphertext caused decapsulation to fail in a way that was distinguishable from a successful decapsulation, and the implementation fell back silently to the classical Diffie-Hellman half. The result was an attacker-triggered downgrade of the hybrid to its classical component. The fix: fail closed on KEM decapsulation failure, and verify that the protocol layer does not have a separate fallback path that re-introduces the downgrade.

From the audit floor 4.2: Negotiation-time hybrid stripping

A separate class of bug: the protocol negotiation allows a man-in-the-middle to strip the post-quantum option from the supported-groups list, and the client falls back to classical. The fix is downgrade protection in the transcript hash — which TLS 1.3 already provides, if you do not work around it. Some custom protocols we have audited had no such protection.

Migrating TLS and PKI

TL;DR

For TLS 1.3 key exchange, enable the X25519MLKEM768 named group. This is the easiest single change with the highest HNDL-mitigation impact you can make today, and it is already in production at Cloudflare, Google, AWS, Microsoft, and Apple — over 65% of human traffic to Cloudflare is post-quantum encrypted as of writing. For certificates, plan a multi-year dual-algorithm transition rather than a flag-day cutover. Authentication is the harder half of the migration.

5.1 | Where TLS migration is easy

The TLS 1.3 handshake's key exchange has been the easiest part of the post-quantum transition. The protocol already supports negotiating named groups, and the Internet Engineering Task Force (IETF) has assigned codepoint 0x11EC (4588) to X25519MLKEM768. The deployment trajectory is steep: Cloudflare's network shipped X25519MLKEM768 in October 2024 (succeeding the earlier draft X25519Kyber768Draft00), Chrome 124 enabled it by default for outgoing connections in April 2024, and the share of post-quantum-encrypted traffic to Cloudflare has gone from roughly 18% of human traffic in 2024 to over 57% in early 2026 to over 65% by mid-2026 [8]. AWS, Microsoft, Meta, and Apple have all shipped support in production.

Decision 5.1: TLS 1.3 key exchange

Enable X25519MLKEM768 alongside X25519 on both clients and servers. Order it first in the supported-groups list so it is preferred when both sides support it. Continue to support X25519 for backward compatibility with peers that do not yet implement the hybrid.

This is, in operational terms, the easiest single change with the highest HNDL impact you can make. If you do nothing else this year, do this.

5.2 | Where TLS migration is hard

The handshake key exchange is one of three places TLS uses asymmetric cryptography. The other two — certificate signatures and certificate chain validation — are operationally harder. Cloudflare's stated 2029 target for full post-quantum security including authentication [8] is a useful reference point: even an organisation with the operational maturity to roll out hybrid KEMs globally is treating the authentication migration as a multi-year project.

5.2.1 | Certificate signatures

A migrated TLS deployment eventually needs:

- Server certificates signed with a post-quantum signature algorithm (typically ML-DSA-65; ML-DSA-87 for CNSA 2.0).
- Intermediate CAs issuing those certificates.
- Root CAs included in client trust stores.
- Updated software at every layer that knows how to parse, validate, and present these signatures.

This is a multi-year programme, and not one that any single deployment can drive unilaterally. The CA/Browser Forum, public CAs, browser vendors, and IETF working groups are coordinating the transition. Practically:

- The first phase (current) is post-quantum *key exchange*, with classical certificate signatures. HNDL threat is addressed; the certificate chain remains classical.
- The second phase will introduce post-quantum or hybrid certificate signatures, likely as dual-algorithm certificates that contain both a classical and a post-quantum signature for a transition window.
- The third phase will deprecate classical signatures.

Decision 5.2: Certificate signature migration

Do not try to issue or accept post-quantum certificate chains until your trust store, your TLS library, and your peers' implementations all support them.

Track the IETF LAMPS and TLS working groups; track CA/Browser Forum baseline requirements; align with the major root programmes (Mozilla, Apple, Microsoft, Chrome).

For server certificates that you control end-to-end (internal services, mTLS within your own fleet), you can move faster — pure ML-DSA-65 is fine in a closed ecosystem. There is no HNDL threat to certificate signatures, so a hybrid signature scheme on these is not the right cost/benefit (Chapter 4).

5.2.2 | Certificate size

ML-DSA-65 signatures are roughly 50x larger than Ed25519 signatures. ML-DSA-87 are larger still. A naive transition to post-quantum certificates inflates TLS handshake sizes from kilobytes to tens of kilobytes per handshake. This has measured real-world performance impact, particularly for:

- Mobile networks with high latency and lossy channels (a single TCP retransmission of an inflated certificate can dwarf the handshake cost).
- Connections with strict CPU or memory budgets (embedded devices, IoT).
- Initial-load performance for web applications, where the handshake is on the critical path.

Pitfall 5.1: Underestimating handshake size

A TLS handshake with a full post-quantum chain (server cert + intermediate + OCSP staple) can exceed the typical Initial Window for TCP. Test under realistic loss conditions. KEMTLS, intermediate certificate suppression, and chain-shortening are all active areas of work; track them.

5.2.3 | Certificate transparency

Certificate Transparency (CT) logs already operate at large scale; every certificate issued by a public CA is logged. With post-quantum certificate signatures, both the certificate and the SCTs become larger. CT log operators will need to scale; relying parties will need to validate larger SCT lists. Plan for this if you are involved in the CT ecosystem.

5.3 | Downgrade protection

TLS 1.3's transcript-hashing approach to downgrade protection generally extends correctly to hybrid named groups, but two specific failure modes have come up in audits.

From the audit floor 5.1: Cleartext supported-groups stripping

A man-in-the-middle modifies the ClientHello's supported-groups extension, removing X25519MLKEM768, before forwarding to the server. The server selects X25519; both sides complete the handshake and detect no anomaly because the transcript hash they compute matches what the (modified) traffic carries. The TLS 1.3 transcript hash protects against tampering with the negotiated parameters, but only after negotiation has occurred — a peer that has been steered into a weaker negotiation outcome cannot detect it from the transcript alone.

This is fundamentally a problem of *policy*: the client must *require* the hybrid when its policy says it should be available, not merely *prefer* it. Configure clients with hard-fail policies for connections to peers known to support the hybrid (e.g. via DNS hints, HSTS-style assertions, or operator policy).

From the audit floor 5.2: Hybrid named group with classical fallback path

A custom TLS-like protocol we audited supported X25519MLKEM768 but had a separate "compatibility" message used when the hybrid handshake failed mid-flight, which fell back to classical without re-deriving any session state. An attacker who could inject a single malformed packet during the post-quantum portion of the handshake could induce the fallback. Standard TLS 1.3 does not have this problem; non-standard protocols sometimes do.

5.4 | QUIC, DTLS, and other transports

QUIC reuses TLS 1.3's handshake; the same migration applies, with the additional consideration that QUIC's initial packets have stricter size constraints (1,200 bytes) and a post-quantum handshake forces additional packets in the initial flight. This is generally fine but worth measuring on lossy paths.

DTLS 1.3 likewise reuses TLS 1.3's handshake. The fragmentation behaviour for large DTLS records becomes more relevant when handshakes grow.

For SSH, OpenSSH 9.0+ supports `sntrup761x25519-sha512@openssh.com` (a different hybrid KEM); migration there is a configuration change. Eventually expect ML-KEM-based hybrids in OpenSSH; until then, the existing hybrid is acceptable.

5.5 | What to do this quarter

Decision 5.3: A 90-day TLS hardening plan

1. Inventory all TLS-terminating services and clients you operate.
2. Audit which TLS library and version each one uses.
3. For each library that supports X25519MLKEM768, enable it on the server side and observe the negotiation rate.
4. For clients you control (mobile apps, agents, daemons), upgrade their TLS library to one that offers the hybrid as a client.
5. Add monitoring: how many of your handshakes are negotiating the hybrid? You want this number to climb monotonically.
6. Defer post-quantum *certificate* migration to a separate, longer programme.

Migrating secure messaging

TL;DR

Secure messaging post-quantum migration is harder than TLS, because the threat model is harsher (HNDL on stored ciphertexts) and the protocol surface is more complex (ratcheting, group messaging, asynchronous initial keys). Where you have a choice: use MLS for new group messaging deployments and follow Signal's PQXDH-style hybrid initial-key construction for one-to-one channels.

6.1 | Why messaging is different

A TLS handshake produces a session key that is used for the lifetime of one connection, typically minutes to hours. A messaging session lasts months or years; the key material is derived once at session establishment and ratcheted forward, with each ratcheted key used to encrypt a stream of messages whose ciphertexts may be stored on infrastructure outside the user's control.

This makes the HNDL threat especially acute: an adversary who records ciphertexts and obtains the post-compromise root keys at any point in the future can decrypt the entire session history. The classical defence — forward secrecy via Diffie-Hellman ratcheting — relies on the Diffie-Hellman exchanges remaining secure forever. In a post-quantum world, that assumption fails.

6.2 | One-to-one: PQXDH-style initial keys

Signal's protocol stack has evolved to add post-quantum protection at session establishment via PQXDH (post-quantum extended triple Diffie-Hellman), which adds a one-shot KEM (using ML-KEM) into the initial key derivation alongside the classical X3DH inputs. Each ongoing message ratchet remains classical, but the initial root key is hybrid, which mitigates the HNDL threat for the bulk of a session's traffic.

Decision 6.1: Messaging initial-key migration

For new one-to-one secure-messaging protocol designs, follow the PQXDH pattern: combine your classical X3DH-style derivation with an ML-KEM encapsulation against a long-lived post-quantum prekey, and feed both into the root KDF.

The post-quantum prekey requires the same prekey-rotation infrastructure as classical signed prekeys, plus storage budget proportional to ML-KEM-768's larger public key (1184 B vs. 32 B for X25519).

6.2.1 | Continuous ratchet PQ

Adding PQ protection to the *continuous ratchet* — not just the initial key derivation — is no longer purely research. Apple's PQ3, deployed in iMessage from iOS 17.4 in February 2024, is the production example: it uses ML-KEM not only at session establishment but on a periodic post-quantum rekey schedule (every 50 messages or every 7 days, whichever comes first), giving the session self-healing properties against post-compromise attackers. The cost is real — ML-KEM rekey adds wire bytes and CPU to a fraction of message rounds — but the design space is now demonstrated rather than hypothetical.

For new designs, the question is no longer “does continuous-ratchet PQ work in production” (yes, at iMessage scale) but “what rekey cadence and what hybrid combiner do you use.” Track Apple's PQ3 specification, the Signal protocol team's public work, and the IETF's MLS working group rather than designing from scratch.

6.3 | Group messaging: MLS

[RFC 9420](#) MLS is the IETF-standardised group messaging protocol. MLS was designed with cryptographic agility in mind: its tree-based key derivation uses a parameterised KEM, and the working group has been actively considering post-quantum and hybrid KEM variants.

Decision 6.2: Group messaging

For new group messaging deployments, build on MLS, not on a custom protocol. MLS has had cryptographic review at a level no proprietary protocol matches, and its post-quantum migration path is being designed by the IETF rather than by you.

6.4 | Stored ciphertexts and key escrow

Many messaging systems store encrypted message archives, encrypted backups, or encrypted multi-device sync state. These ciphertexts are the primary target of the HNDL threat: they are stored, often for years, and the protection is asymmetric encryption against a long-lived keypair.

Decision 6.3: Encrypted backup and archive keys

Migrate the asymmetric encryption used for backups, archives, and multi-device sync to a hybrid KEM construction with explicit forward-secrecy properties. Treat these keypairs as having the longest exposure window in your system.

6.5 | Pitfalls in messaging migration

Pitfall 6.1: Treating “post-quantum” as a single switch

A messaging app may have ten places where asymmetric primitives are used: identity keys, signed prekeys, one-time prekeys, the initial root key derivation, the message ratchet, group key derivation, backup encryption, multi-device sync, attestation, and so on. “Adding post-quantum” means addressing each of them, with appropriate parameter choices, in an order that matches the threat model. There is no single switch.

Pitfall 6.2: Ignoring the prekey distribution channel

ML-KEM prekeys are 1184 bytes each; if you previously distributed 100 prekeys per user, you now distribute 118400 bytes. This affects bandwidth budgets at the server, prekey-pool sizing, and battery on mobile clients that fetch and verify them. Plan the distribution channel.

From the audit floor 6.1: Replay-attackable post-quantum prekeys

We audited a system where post-quantum prekeys were issued one-shot but the server had no mechanism to retire them after use. An attacker with access to the server (or to a compromised prekey response) could replay the same ML-KEM encapsulation, defeating forward secrecy. The fix is the same as for classical one-time prekeys: the server must atomically remove a prekey when it is consumed.

The library landscape

TL;DR

Several post-quantum libraries are now production-ready. The list of implementations that have passed Symbolic Software's *Crucible* conformance battery cleanly is the right shortlist for most teams: AWS-LC, Cloudflare CIRCL, the Go standard library, libcrux, mlkem-native, liboqs, wolfCrypt, Trail of Bits ml-dsa, and a handful of others (see Chapter 8). Pick a library whose maintenance, audit posture, and language ecosystem fit your deployment, not one with the longest feature list.

7.1 | What “ready” means

For a library to be *production-ready* for post-quantum cryptography, in our view, it must:

1. Implement the finalised NIST standards ([FIPS 203](#), [FIPS 204](#), optionally [FIPS 205](#)) — not earlier IND-CPA-only round-3 candidates.
2. Have constant-time guarantees for the operations that need them, with evidence: explicit constant-time engineering, testing under tools like `ctgrind` / `TIMECOP` / `dudect`, and ideally formal verification of the relevant code paths.
3. Have current and ongoing maintenance — post-quantum cryptography moves quickly, and a library that has been “done” for two years is probably out of date.
4. Be auditable: open-source, with a clear source of truth, in a language that allows independent review.
5. Be supported in your deployment environment (operating system, language ecosystem, FFI).
6. Pass an independent conformance battery — ideally *Crucible*, ideally as part of CI.

7.2 | The Crucible-clean shortlist

The fastest filter is to start with the implementations that passed Symbolic Software’s full Crucible battery cleanly. As of the most recent published run (Chapter 8):

- **Kyber-K2SO** (Symbolic Software, Go) — clean Go reference.
- **mlkem-native** (C) — ships inside AWS-LC and liboqs.
- **Go standard library** `crypto/mlkem` — shipped in Go 1.24 (Feb 2025), enabled by default in TLS 1.3.
- **AWS-LC** (Amazon, C) — AWS KMS, S3, CloudFront.
- **CIRCL** (Cloudflare, Go) — Cloudflare TLS deployment.
- **liboqs** (Open Quantum Safe, C).
- **wolfCrypt** (C) — FIPS 140-3 validated, embedded/IoT.
- **itzmeanjan/ml-kem** (C++20) — header-only, fully `constexpr`.
- **Trail of Bits ml-dsa** (Go) — side-channel resistant ML-DSA.

These are not the only acceptable libraries. They are the ones with both significant deployment exposure and a clean run against an independent conformance battery as of writing. Library landscapes change quickly; verify the current Crucible run before relying on this list.

7.3 | Selected libraries by ecosystem

7.3.1 | C/C++ ecosystem

- **BoringSSL** (Google): production-ready ML-KEM-768 in TLS, used in Chrome and other Google services. Not a stable API library; intended for Google use.
- **AWS-LC** (Amazon): forked from BoringSSL, production-ready ML-KEM and ML-DSA, used in AWS services. Stable API. Crucible-clean.
- **OpenSSL 3.x**: ML-KEM and ML-DSA support is shipping in 3.5+; check the specific version for current status.
- **liboqs** (Open Quantum Safe): a research/experimentation library bundling many PQ schemes, including post-NIST candidates. Useful for compatibility testing. Crucible-clean for the standardised primitives.

- **libsodium**: post-quantum support is in progress; check the maintainer roadmap.
- **wolfCrypt**: FIPS 140-3 validated; strong fit for embedded and IoT. Crucible-clean.

7.3.2 | Rust ecosystem

- **rustls + ring/aws-lc-rs**: X25519MLKEM768 support via the underlying provider. Production usage growing.
- **pqcrypto** crate family: bindings to PQClean reference implementations. Useful for prototyping. Note: PQClean and the `pqcrypto/rustpq` bindings showed conformance gaps in our Crucible run — specifically the missing encapsulation key modulus check. Verify the specific algorithm crate’s audit status before production use.

7.3.3 | Go ecosystem

- **Go crypto/mlkem**: ML-KEM shipped in the Go standard library as of Go 1.24 (February 2025), with X25519MLKEM768 enabled by default in `crypto/tls`. Use the standard library when available. Crucible-clean.
- **Symbolic Software Kyber-K2SO**: clean Go reference implementation of ML-KEM (FIPS 203). Crucible-clean. Use where the standard library is not yet sufficient or where independent implementation diversity is wanted.
- **CIRCL** (Cloudflare): broad post-quantum primitives. Powers Cloudflare’s TLS deployment. Crucible-clean.
- **Trail of Bits ml-dsa**: side-channel resistant ML-DSA. Crucible-clean.

7.3.4 | Java / .NET ecosystems

- **Bouncy Castle**: long-standing support for many PQ schemes, including ML-KEM and ML-DSA. The 1.80 release showed an input-validation conformance gap (wrong-length decapsulation keys silently accepted; see Chapter 8). Verify the current version.
- **.NET Cryptography**: ML-KEM and ML-DSA support is rolling in via the standard cryptographic providers; check the specific .NET version.

7.4 | Choosing a library

Decision 7.1: Library selection

Choose the library that meets all of:

1. Implements the finalised FIPS standards.
2. Has been independently audited, ideally by more than one party.
3. Is actively maintained, with releases in the last 6 months.
4. Is supported in your language and runtime.
5. Passes a conformance battery (Crucible or equivalent), with the run reproducible in your CI.
6. Has a clear position on side-channel resistance for your deployment context (server, embedded, HSM).

For most teams, this points to the platform-default — the standard library of your language, or the crypto library shipped by your platform vendor (Apple CryptoKit, AWS-LC, Cloudflare CIRCL) — not to a third-party crate or package with unclear maintenance.

Pitfall 7.1: Mixing reference and optimised implementations

NIST reference implementations are designed for clarity, not for performance or constant-time behaviour. Several pitfalls we have seen:

- Production code linking against the reference implementation, expecting AVX2-class throughput. Performance falls off a cliff.
- Production code linking against an unaudited optimised implementation that diverges from the reference in subtle ways (e.g. different rejection-sampling order). Test against Known Answer Test (KAT)s and Crucible.
- Code that links the reference implementation in tests and the optimised one in production. Differential testing is good; differential *environments* are bad.

From the audit floor 7.1: Variable-time decapsulation

We have audited Go and C implementations of ML-KEM where the decapsulation path branched on the result of an internal comparison in a way that was distinguishable via timing. ML-KEM's design tolerates implicit-rejection, and the standard *requires* constant-time decapsulation precisely because the rejection-vs-accept distinction would otherwise be observable. The KyberSlash class of timing attacks specifically exploited gaps of this kind in pre-FIPS implementations. Crucible's decapsulation-robustness category targets this bug class.

From the audit floor 7.2: Weak randomness in keygen

A surprisingly common bug: an ML-KEM keygen function that draws randomness from a non-cryptographic source, or from a source that has not been seeded. The post-quantum primitives have very specific entropy requirements; do not assume that a working implementation exercises the random number generator path.

Conformance testing with Crucible

TL;DR

Even a correct-looking post-quantum implementation can fail in specific bug-class patterns we have seen repeatedly in audits. Symbolic Software publishes *Crucible*, an open-source conformance testing framework that encodes those bug classes as targeted, reusable test batteries. We tested 15 of the most-deployed ML-KEM and ML-DSA implementations across five languages with it: 1,296 test executions, two minor conformance gaps, zero security vulnerabilities. The implementations that matter — AWS-LC, Cloudflare CIRCL, Go's standard library, wolfSSL's FIPS module — all passed.

8.1 | What Crucible tests

Crucible is not a unit test suite for a specific library. It is a *conformance harness* that exercises an implementation against a curated set of test categories, each targeting a specific class of bug we have observed in real-world post-quantum cryptography audits or in the public literature (KyberSlash, various timing advisories). The current public release covers [FIPS 203](#) (ML-KEM) and [FIPS 204](#) (ML-DSA):

- The ML-KEM battery has **78 tests across 6 categories**: compression arithmetic, NTT correctness, coefficient bounds enforcement, decapsulation robustness, serialisation, and rejection sampling.
- The ML-DSA battery has **51 tests across 6 categories**: norm checks, arithmetic, signing internals, verification edge cases, serialisation, and constant-time behaviour.
- Every test is tagged with the bug class it targets, the FIPS spec section it verifies, and — where applicable — the real-world audit finding that motivated it.

8.2 | What we found

We built harnesses for the fifteen most widely deployed ML-KEM and ML-DSA implementations and ran the full battery against each. Across 1,296 test executions, we found two minor conformance gaps and **zero security vulnerabilities**.

8.2.1 | The implementations that passed cleanly

Eleven of the fifteen implementations passed every applicable test:

- **Kyber-K2SO** (Symbolic Software, Go) — our own clean reference.
- **mlkem-native** (C) — ships inside AWS-LC and liboqs.
- **Go standard library** `crypto/mlkem` — used by every Go 1.24+ binary that uses TLS 1.3.
- **AWS-LC** (Amazon, C) — powers AWS KMS, S3, CloudFront.
- **CIRCL** (Cloudflare, Go) — production TLS deployment.
- **liboqs** (Open Quantum Safe, C).
- **wolfCrypt** (C) — FIPS 140-3 validated, embedded/IoT.
- **itzmeanjan/ml-kem** (C++20) — header-only, fully constexpr.
- **Trail of Bits ml-dsa** (Go) — side-channel resistant.

These are the implementations that ship inside AWS, Cloudflare, the Go standard library, and wolfSSL's FIPS module. The post-quantum primitives, where they are deployed at scale, are being implemented correctly.

8.2.2 | Conformance Finding 1: Missing encapsulation key modulus check

- **Affected:** `pq-crystals/kyber` reference, PQClean, `rustpq/pqcrypto`.
- **Severity:** Conformance gap (not a security vulnerability).
- **Spec reference:** FIPS 203 §7.2, Equation 7.1.

FIPS 203 requires that before encapsulation, the encapsulation key must pass a modulus check ensuring that no 12-bit coefficient encodes a value $\geq q = 3329$. All three affected implementations accept a key with a coefficient value of 3329 without error and provide no separate key-validation function. This is a spec conformance

gap rather than a vulnerability: the practical impact is interoperability (two implementations may handle an out-of-range coefficient differently — one reduces mod q , another wraps — producing different shared secrets from the same key). The pq-crystals codebase predates FIPS 203, and the modulus check was added during standardisation.

8.2.3 | Conformance Finding 2: Bouncy Castle accepts wrong-length decapsulation keys

- **Affected:** Bouncy Castle 1.80 (Java).
- **Severity:** Input validation gap.
- **Spec reference:** FIPS 203 §7.3.

Bouncy Castle's `MLKEMPrivateKeyParameters` constructor accepts byte arrays of any length without validating against the expected size for the parameter set. A decapsulation key one byte short of the expected length (2399 bytes vs. 2400 for ML-KEM-768) is accepted and decapsulation proceeds. A truncated dk means the implicit-rejection secret z is partially missing, which could weaken the Fujisaki–Okamoto transform's security guarantees. In practice, the dk is private and an attacker would not normally control it — but a corrupted key file should fail loudly rather than silently produce wrong results.

8.3 | How to wire up your implementation

Crucible communicates with the system under test over a simple stdin/stdout JSON line protocol. Wiring up an implementation typically takes well under a day. The minimal harness is:

1. Accept JSON requests on stdin specifying the operation (`keygen`, `encaps`, `decaps`, `sign`, `verify`, plus parameter set), input bytes, and any test-specific metadata.
2. Invoke the implementation under test.
3. Return the result as a JSON line on stdout.

This shape lets the same Crucible test battery run against implementations in any language — the harness is small and the cross-language friction is contained to a single shim. Crucible ships with example harnesses for Rust, Go, and C.

Decision 8.1: When to run Crucible

Run Crucible at three points in your migration:

1. Once when you first integrate a new PQ library, to establish a baseline.
2. In CI on every commit that touches the cryptographic dependency or the harness.
3. Once before each release, against the release artefact rather than a development build.

The first run is where the surprises happen.

8.4 | What conformance testing does and does not give you

- Conformance testing detects *known* bug classes. It cannot find an unknown one.
- Conformance testing of a library does not certify the system that uses it. A correct ML-KEM implementation can still be misused at the protocol layer.
- Conformance testing complements, but does not replace, an audit. Audits look at design and integration; conformance tests look at primitive correctness.

8.5 | Beyond Crucible

Other testing approaches that complement conformance testing:

- **NIST KATs:** NIST publishes Known Answer Tests for each finalised standard. Run them. They catch encoding bugs and gross numerical bugs but exercise correct-input round-trips only — they do not exercise edge cases or malformed input.
- **Fuzz testing:** especially valuable on the deserialisation paths, which handle attacker-controlled input.
- **Constant-time analysis:** tools like `ctgrind`, `TIMECOP`, `dudect`, and Valgrind's uninitialised-memory checks find timing-distinguishable code paths.
- **Formal verification:** such efforts can sometimes remove entire bug classes from the implementation, at the cost of substantial verification effort.

From the audit floor 8.1: Implementations that pass KATs but fail Crucible

NIST KATs exercise correct-input round-trips. Crucible exercises malformed inputs, out-of-bounds inputs, and edge cases. We have seen multiple ML-KEM implementations that pass every NIST KAT but fail in production because they accept malformed ciphertexts that a stricter implementation would reject. Conformance to KATs is necessary but not sufficient.

From the audit floor 8.2: Constant-time-by-the-compiler

A surprisingly persistent bug class: source code that looks constant-time but is compiled into branchy code by the optimiser. Verify the constant-time property at the binary level (with `ctgrind` or equivalent), not at the source level. The KyberSlash class of timing attacks specifically exploits this gap.

8.6 | Get Crucible

Crucible is open source and available at <https://github.com/symbolicsoft/crucible>. It ships with harness templates for Rust, Go, and C. If you maintain an ML-KEM or ML-DSA implementation, wire it up and run the battery. If you find a bug we missed, it becomes a new test — the framework grows sharper over time.

Rollout strategy

TL;DR

Treat the post-quantum migration as a regular production rollout: dark launch, shadow traffic, gradual percentage ramp, and an explicit rollback path. Cloudflare's public reporting of over 65% post-quantum-encrypted human traffic — with full authentication migration targeted for 2029 — is a useful reference point: even a sophisticated, well-instrumented operator is treating this as a multi-year programme.

9.1 | Phases of a TLS hybrid rollout

1. **Library upgrade:** ship the new TLS library or provider into production behind a feature flag. The hybrid is built-in but not yet enabled. Verify that the upgrade itself does not regress.
2. **Dark launch (1–5% of traffic):** enable the hybrid for a small fraction of clients and observe.
3. **Ramp (5% → 25% → 50% → 100%):** increase the fraction over weeks. At each step, watch the indicators in Section 9.2.
4. **Default-on:** the hybrid is the default for all traffic. Classical fallback remains for non-supporting peers.
5. **Hardening:** tighten policy — for traffic to peers known to support the hybrid, fail-closed if the hybrid is not negotiated.

9.2 | What to monitor during rollout

9.2.1 | Negotiation rate

The fraction of TLS handshakes that successfully negotiate X25519MLKEM768. This should climb monotonically as both ends of more connections are upgraded. A flat or falling negotiation rate is a signal that something is degrading the upgrade path. Cloudflare’s published 65% figure for human traffic is a useful upper-bound benchmark: that is roughly the ceiling you can reach today with a global edge that proactively offers the hybrid.

9.2.2 | Handshake latency

Both p50 and p99. The hybrid handshake has slightly more compute and meaningfully more bytes than X25519. On well-provisioned servers and modern clients you should see a small but measurable latency cost. A large or growing cost is a signal of CPU saturation or unexpected fragmentation.

9.2.3 | Handshake failures

By failure type. Failures classified as “unsupported group” from peers should be small and decreasing; failures classified as “protocol error” or “decode error” are more concerning and warrant investigation.

9.2.4 | Bytes on the wire

Per-handshake byte counts. A regression here is the precursor to retransmission-driven latency problems on lossy paths.

9.2.5 | CPU and memory

ML-KEM keygen and encapsulation are not free. Watch CPU on the servers handling the most handshakes per second.

9.2.6 | Library- and platform-specific symptoms

Some libraries surface specific telemetry (handshake group counters, fallback counters). Use them.

9.3 | Rollback

Have a rollback path before you start. The dark launch and ramp are exactly so that you have a safe point to roll back to. For the hybrid KEM case, rollback is simply re-disabling the hybrid in the supported-groups list — the classical Key Exchange (KEX) continues to work. Practise this. A rollback you have never executed is not a rollback.

Decision 9.1: Rollback policy

At each phase of the rollout, define explicit thresholds at which you will roll back. For example: “If hybrid handshake failure rate exceeds 0.5% over a 30-minute window, automatically reduce the hybrid percentage by half.” Ad-hoc decisions during an incident are how you end up rolling forward when you should have rolled back.

9.4 | Operational pitfalls we have seen

Pitfall 9.1: Asymmetric upgrades

Server-side hybrid support is upgraded; client-side is not. The negotiation rate stays at zero, and the team concludes “the hybrid does not work”. The hybrid works fine; the clients are old. Plan client and server upgrades together, and instrument both sides.

Pitfall 9.2: Hidden middleboxes

Corporate proxies, load balancers, deep packet inspection devices, and TLS-terminating CDNs in the request path can all silently strip a hybrid named group, or reject larger ClientHellos as malformed. We have seen rollouts blocked for weeks because of a single middlebox. Test through every layer of your actual production path.

Pitfall 9.3: OCSP and certificate-side surprises

A larger handshake combined with OCSP stapling can push the response over a buffer or fragmentation threshold somewhere in the stack. The TLS-side rollout tests fine in isolation; the failure shows up only with real cert chains and stapled OCSP.

From the audit floor 9.1: Ratchet roll-forward in messaging

A messaging-system rollout we audited intended to migrate the prekey pool to post-quantum prekeys, but the client logic for “do I have a post-quantum prekey?” was racy with respect to the prekey-pool refresh, and a fraction of new sessions were established with classical-only initial keys despite both ends supporting the hybrid. Operational rollouts of cryptographic upgrades have data-race surfaces that pure protocol designs do not. Test under load.

9.5 | Communication

A migration that ships well but is not communicated well will be reported as a security regression by the first journalist who notices the larger handshake. Have an external-communications plan:

- Announce the rollout publicly, with rationale (HNNDL threat, regulatory deadlines).
- Document what changed at the protocol level so security researchers can follow along.
- If your peers can opt in to “require hybrid” policy, tell them how.

9.6 | Procurement leverage

One of the most useful things a large organisation can do is make post-quantum support a contractual requirement for vendors. Cloudflare have explicitly recommended this [8]: “make post-quantum support a requirement for any procurement.” For organisations that buy software, SaaS, or cloud services, this is the single biggest lever for moving the broader ecosystem. Vendors do not migrate on principle; they migrate on contract.

From the audit floor: a bug-class gallery

TL;DR

This chapter collects, in one place, the bug classes that have shown up in our audits of post-quantum implementations and integrations. None of these are new findings published here for the first time; we describe them in the abstract, decoupled from any specific client. They are listed so you can search for them in your own code.

10.1 | Primitive-level bug classes

From the audit floor 10.1: Variable-time secret-dependent code

The most common class of finding. Even when the source code looks constant-time, the compiler may emit branches; the CPU may execute conditional moves with data-dependent timing; the memory access pattern may leak through cache state. Verified at the binary level by tools like `ctgrind`, `dudect`, or `TIMECOP`. The KyberSlash family of timing attacks exploited exactly this gap in pre-FIPS Kyber implementations.

From the audit floor 10.2: Missing or wrong bounds checks

ML-KEM and ML-DSA both require coefficient bounds checking on parsed inputs. We have audited implementations that omitted the check entirely (accepting malformed inputs), checked the wrong bound (off-by-one), or short-circuited the check on a faster path (correct on the slow path, wrong on the fast).

The Crucible run published in Chapter 8 found a specific instance of this class: `pq-crystals/kyber` reference, `PQClean`, and `pqcrypto/rustpq` all accept an

encapsulation key with a 12-bit coefficient at $q = 3329$, contrary to FIPS 203 § 7.2's modulus check. This is a conformance gap, not a vulnerability — but the same shape of bug, in a different code path, is.

From the audit floor 10.3: Length-validation gaps at the API boundary

A separate but related class: APIs that accept byte buffers without validating against the parameter set's expected length. The Crucible run found Bouncy Castle 1.80 accepting an ML-KEM-768 decapsulation key one byte short of the expected 2,400. A truncated dk leaves the implicit-rejection secret partially missing, weakening the FO transform. The fix is to validate buffer lengths at the constructor and refuse partial keys.

From the audit floor 10.4: Rejection sampling errors

The secret and error distributions in lattice schemes are sampled by rejection. Mistakes here are subtle: the implementation may sample the right *number* of values but with a marginally wrong distribution, producing keys that interoperate with reference implementations but degrade security claims.

From the audit floor 10.5: Decapsulation oracle leaks

ML-KEM's CCA security depends on a decapsulation procedure that is indistinguishable on success and on (implicit) rejection. We have audited implementations where decapsulation failures were observable via timing, error codes, log lines, exception types, or "optimisation" early returns.

From the audit floor 10.6: Determinism failures in ML-DSA

ML-DSA can be deterministic or randomised. Implementations that pretend to support both but actually only implement one, or that mix randomness sources unpredictably, have shown up in audits.

10.2 | Protocol-integration bug classes

From the audit floor 10.7: Hybrid downgrade

A hybrid construction reduces to its weaker component if any of: the negotiation can be tampered with; the post-quantum half can be selectively failed; or the combiner XORs / discards rather than KDFs.

From the audit floor 10.8: Transcript binding omissions

Hybrid KEMs combined via a KDF *must* bind the public keys and ciphertexts of both halves into the KDF input. Implementations that derive only from the two shared secrets are vulnerable to maliciously chosen public keys. The X-Wing combiner is correct; rolled-your-own combiners often are not.

From the audit floor 10.9: Length-confusion in serialisation

Post-quantum primitives have larger and more varied byte-string lengths than classical ones. We have audited code where length fields were hard-coded for X25519's 32 bytes and silently truncated ML-KEM-768's 1184-byte public keys.

From the audit floor 10.10: Compliance theatre

The codebase contains a post-quantum implementation; the data path does not. We see this in marketing claims, compliance checklists, and self-attestation forms more than in reviewed designs. The fix is to validate the data path under traffic.

10.3 | Operational and key-management bug classes

From the audit floor 10.11: Insufficient entropy at keygen

A keygen function that draws from a non-cryptographic source, an unseeded source, or a deterministic source. Lattice schemes have specific entropy requirements — check that the keygen RNG path is exercised.

From the audit floor 10.12: Leaking long-term keys via test endpoints

Test endpoints, debugging interfaces, and “compatibility” modes that expose private key material under conditions that the threat model does not consider but that are reachable in production.

From the audit floor 10.13: Key-rotation amnesia

A successfully migrated system rotates its post-quantum keys — but the backup, archive, or escrow mechanism continues to encrypt to the original (now-deprecated) key. Three years later the migration is materially undone.

From the audit floor 10.14: Prekey replay in PQ messaging

One-shot post-quantum prekeys distributed without a server-side mechanism to retire them on use. An attacker with read access to the prekey response can replay it, defeating forward secrecy.

10.4 | Library and dependency bug classes

From the audit floor 10.15: Reference implementation in production

The reference implementation is for clarity, not for performance or constant-time. Replacing it with an audited optimised implementation, with explicit verification that the two agree on every KAT and Crucible test, is a common omission.

From the audit floor 10.16: Out-of-date library dependency

A library that shipped post-quantum support six months ago may have shipped a security fix four months ago. Pin to a current version, not to the version that worked when you wrote the integration.

From the audit floor 10.17: FFI boundary mistakes

Foreign-function-interface boundaries are where buffer-length, ownership, and memory-zeroisation invariants get lost. PQ primitives have larger buffers; the chance of a length mismatch at an FFI boundary is correspondingly larger.

10.5 | What to do with this chapter

This is not a checklist. It is a recall list: a way to recognise the shape of a finding when you see it. If you see something in your code that resembles one of these patterns, do not assume it is a benign instance. Either eliminate the pattern or write down explicitly why your instance is safe and have the explanation reviewed.

About Symbolic Software



Symbolic Software¹ is an applied cryptography consultancy founded in 2017 by Dr. Nadim Kobeissi in Paris, France. We deliver design-level security for the AI era: protocol architecture, cryptographic primitive selection, threat modelling, and formal verification. AI systems can now discover zero-day vulnerabilities and write working exploits autonomously; the security that matters now is upstream of that, and that is where we work.

Over the past nine years, we have conducted more than 250 design-level security engagements for organisations including Mozilla, 1Password, Coinbase, Zoom, and the Linux Foundation. We have also published peer-reviewed research software used by cryptographers and engineers worldwide, including the Verifpal[®] formal verification framework, the Noise Explorer protocol analyser, the Kyber-K2SO post-quantum library, and the Crucible conformance-testing harness for NIST PQC.

¹Stay updated on Symbolic Software's latest work by visiting <https://symbolic.software>.

Bibliography

- [1] Bas Westerbaan. *Don't wait for quantum computer factoring records — you'll be too late*. 2026. URL: <https://bas.westerbaan.name/notes/2026/04/02/factoring.html> (visited on 05/05/2026).
- [2] Scott Aaronson. *The Shor of Damocles: a position on post-quantum migration*. 2026. URL: <https://scottaaronson.blog/?p=9718> (visited on 05/05/2026).
- [3] Google Quantum AI and collaborators. *Updated quantum resource estimates for ECDLP-256 (responsible disclosure)*. Whitepaper, March 30 2026. 2026. URL: <https://research.google/blog/safeguarding-cryptocurrency-by-disclosing-quantum-vulnerabilities-responsibly/> (visited on 05/05/2026).
- [4] Manuel Barbosa et al. *X-Wing: The Hybrid KEM You've Been Looking For*. Cryptology ePrint Archive, Paper 2024/039. 2024. URL: <https://eprint.iacr.org/2024/039>.
- [5] Symbolic Software. *Crucible: Conformance Testing Framework for Post-Quantum Cryptography*. 2026. URL: <https://github.com/symbolicsoft/crucible> (visited on 05/05/2026).
- [6] Symbolic Software. *Recommending Post-Quantum Native Design Under Epistemic Duress*. 2026. URL: <https://symbolic.software/blog/2026-04-02-pq-native/> (visited on 05/05/2026).
- [7] Symbolic Software. *Hybrid Constructions Are a Safety Blanket, and That's Fine*. 2026. URL: <https://symbolic.software/blog/2026-04-13-hybrid-constructions/> (visited on 05/05/2026).
- [8] Cloudflare. *Our post-quantum cryptography roadmap*. 2026. URL: <https://blog.cloudflare.com/post-quantum-roadmap/> (visited on 05/05/2026).

-
- [9] National Institute of Standards and Technology. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard*. 2024. URL: <https://csrc.nist.gov/pubs/fips/203/final> (visited on 05/05/2026).
- [10] National Institute of Standards and Technology. *FIPS 204: Module-Lattice-Based Digital Signature Standard*. 2024. URL: <https://csrc.nist.gov/pubs/fips/204/final> (visited on 05/05/2026).
- [11] National Institute of Standards and Technology. *FIPS 205: Stateless Hash-Based Digital Signature Standard*. 2024. URL: <https://csrc.nist.gov/pubs/fips/205/final> (visited on 05/05/2026).
- [12] National Security Agency. *Commercial National Security Algorithm Suite 2.0*. 2022. URL: <https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3148990/nsa-releases-future-quantum-resistant-qr-algorithm-requirements-for-national-se/> (visited on 05/05/2026).
- [13] Richard Barnes et al. *RFC 9420: The Messaging Layer Security (MLS) Protocol*. 2023. URL: <https://www.rfc-editor.org/rfc/rfc9420> (visited on 05/05/2026).
- [14] Eric Rescorla. *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3*. 2018. URL: <https://www.rfc-editor.org/rfc/rfc8446> (visited on 05/05/2026).
- [15] Signal Foundation. *The PQXDH Key Agreement Protocol*. 2023. URL: <https://signal.org/docs/specifications/pqxdh/> (visited on 05/05/2026).

Reference tables

This appendix collects compact reference tables for use during migration. These are not a substitute for the relevant standards, but they are useful at a glance.

A.1 | NIST post-quantum standards (2024)

Table A.1: Finalised NIST PQC standards and their status as of mid-2026.

Standard	Algorithm	Type	Originating submission
FIPS 203	ML-KEM	KEM (lattice)	CRYSTALS-Kyber
FIPS 204	ML-DSA	Signature (lattice)	CRYSTALS-Dilithium
FIPS 205	SLH-DSA	Signature (hash)	SPHINCS+
(forthcoming)	FN-DSA	Signature (lattice)	Falcon

A.2 | TLS named groups for hybrid PQ

Table A.2: Selected TLS 1.3 named groups for hybrid post-quantum key exchange.

Codepoint	Name	Construction
0x11EC	X25519MLKEM768	X25519 + ML-KEM-768 (current default)
0x001D	x25519	X25519 (classical, retain for compat.)

A.3 | Approximate primitive sizes

Table A.3: Public-key, ciphertext / signature, and shared-secret sizes for selected primitives. “Approx.” values come from the standard documents; check the latest revision.

Primitive	Public key	Ciphertext / signature	Shared secret
X25519	32 B	32 B	32 B
Ed25519	32 B	64 B	—
ECDSA-P256	33–65 B	64–72 B	—
ML-KEM-512	800 B	768 B	32 B
ML-KEM-768	1184 B	1088 B	32 B
ML-KEM-1024	1568 B	1568 B	32 B
ML-DSA-44	1312 B	2420 B	—
ML-DSA-65	1952 B	3309 B	—
ML-DSA-87	2592 B	4627 B	—
SLH-DSA-128s	32 B	7856 B	—
SLH-DSA-128f	32 B	17088 B	—

A.4 | Useful pointers

NIST CSRC

<https://csrc.nist.gov/projects/post-quantum-cryptography>

NSA CNSA 2.0

<https://www.nsa.gov/cybersecurity>

BSI TR-02102

<https://www.bsi.bund.de/>

ANSSI positioning papers

<https://www.ssi.gouv.fr/>

IETF TLS working group

<https://datatracker.ietf.org/wg/tls/>

IETF LAMPS working group

<https://datatracker.ietf.org/wg/lamps/>

Symbolic Software Crucible

<https://github.com/symbolicsoft/crucible>

Symbolic Software Kyber-K2SO

<https://github.com/symbolicsoft/kyber-k2so>

Cloudflare PQ research

<https://blog.cloudflare.com/post-quantum-roadmap/>