

Making Post-Quantum Authentication Fit in Embedded Stacks

Luis Chan, PQShield · luis.chan@pqshield.com · September 2026

The increased key and signature sizes of post-quantum schemes have made resource costs a central concern in bringing them to constrained platforms. While optimised implementations help make the primitives practical, the same large objects must also pass through the middleware that turns them into firmware or device authentication. Consider a Zephyr-based product using MCUboot for firmware authentication, Mbed TLS for certificate-based communications, and PSA with TF-M for cryptographic services and isolation.¹ These components provide the interfaces, formats and tools on which the product depends. The constraints driving primitive optimisation must therefore also shape the middleware and authentication designs built around it.

The immediate difficulty is obtaining the necessary support throughout this stack. A provider capable of verifying a PQ signature still needs the surrounding key handling, authentication formats and signing tools to make that operation available to a product. Where this functionality is missing, product teams must obtain or maintain the integrations themselves. The work crosses independently maintained components, making it difficult to treat PQ migration as an ordinary cryptographic dependency update. Recent customer requests to integrate PQ support into this middleware have brought us directly into these gaps, where providing an optimised primitive is only the beginning of the deployment work.

Completing that support also raises the question of what the integrated operation costs. Assumptions that were affordable with smaller classical objects can amplify the resource demands of PQ authentication: defensive copies may coexist across layers, while buffers sized for the largest supported signature can reserve memory even when handling smaller objects. A primitive may consequently fit within a device's budget while the authentication operation built around it does not. Middleware integration must account for the resources consumed across the complete path, alongside the work needed to make each component support the algorithm.

Even efficient integration, however, leaves the costs of the authentication design itself. Formats and protocols determine how much signed material must be transmitted or retained, how often it must be verified, and whether previously authenticated information can be reused. These choices provide opportunities to make PQ authentication more practical beyond improvements to the underlying code. The draft on Merkle Tree Certificate deployment use cases illustrates this direction, exploring batched device-certificate issuance and constrained-device validation beyond the Web PKI. Its use of maintained verifier state to support smaller certificates also introduces storage and update requirements, showing why such optimisations must be assessed within the intended deployment.² The same constraints that determine whether an implementation fits should therefore inform the design of the authentication mechanism it implements.

As a cryptographic vendor working on these integrations, we want to understand where customers encounter, or anticipate, the greatest strain as PQ authentication becomes available through their existing stacks. Our integration experience provides one perspective on these costs; operators and other implementers can help establish which constraints dominate in practice and which trade-offs their deployments can accommodate. Connecting that experience with ongoing IETF work would help identify both opportunities for more efficient authentication designs and the middleware support needed to deploy them. Which embedded authentication workflows would benefit most from changes above the primitive, and what deployment requirements should guide that work?

¹Zephyr TF-M architecture, <https://docs.zephyrproject.org/latest/services/tfm/overview.html>

²MTC deployment use cases, <https://www.ietf.org/archive/id/draft-gray-plants-mtc-deploy-use-cases-01.html>