NIST SPECIAL PUBLICATION 1800-38C

Migration to Post-Quantum Cryptography Quantum Readiness: Testing Draft Standards

Volume C:

Quantum-Resistant Cryptography Technology Interoperability and Performance Report

William Newhouse
Murugiah Souppaya
National Institute of Standards and Technology
Rockville, Maryland

William Barker
Dakota Consulting
Silver Spring, Maryland

Chris Brown
The MITRE Corporation
Mclean, Virginia

Panos Kampanakis
Amazon Web Services, Inc. (AWS)
Arlington, Virginia

Jim Goodman
Crypto4A Technologies, Inc.
Ontario, Canada

Julien Prat
Robin Larrieu
CryptoNext Security
Paris, France

John Gray
Mike Ounsworth
Cleandro Viana
Entrust
Minneapolis, Minnesota

Hubert Le Van Gong
JPMorgan Chase Bank, N.A.
Jersey City, New Jersey

Kris Kwiatkowski
PQShield
Oxford, United Kingdom

Anthony Hu
wolfSSL
Seattle, Washington

Robert Burns
Thales DIS CPL USA, Inc.
Austin, Texas

Christian Paquin
Microsoft
Redmond, Washington

Jane Gilbert
Gina Scinta
Thales Trusted Cyber Technologies
Abingdon, MD

Eunkyung Kim
Samsung SDS Co., Ltd.
Seoul, Republic of South Korea

Volker Krummel
Utimaco
Nordrhein-Westfalen, Germany

December 2023

PRELIMINARY DRAFT

This publication is available free of charge from
https://www.nccoe.nist.gov/crypto-agility-considerations-migrating-post-quantum-cryptographic-algorithms

DISCLAIMER

Certain commercial entities, equipment, products, or materials may be identified by name or company logo or other insignia in order to acknowledge their participation in this collaboration or to describe an experimental procedure or concept adequately. Such identification is not intended to imply special status or relationship with NIST or recommendation or endorsement by NIST or NCCoE; neither is it intended to imply that the entities, equipment, products, or materials are necessarily the best available for the purpose.

While NIST and the NCCoE address goals of improving management of cybersecurity and privacy risk through outreach and application of standards and best practices, it is the stakeholder's responsibility to fully perform a risk assessment to include the current threat, vulnerabilities, likelihood of a compromise, and the impact should the threat be realized before adopting cybersecurity measures such as this recommendation.

National Institute of Standards and Technology Special Publication 1800-38C Natl. Inst. Stand. Technol. Spec. Publ. 1800-38C, 100 pages, (December 2023), CODEN: NSPUE2

FEEDBACK

You can improve this initial public draft by submitting comments.

This initial draft offers: (1) identification of compatibility issues between quantum-ready algorithms; (2) resolution of compatibility issues in a controlled, non-production environment; and (3) reduction of time spent by individual organizations performing similar interoperability testing for their own PQC migration efforts.

You can improve this initial public draft by submitting comments. We are always seeking feedback on our publications and how they support our readers' needs. We are particularly interested in learning from readers if this initial draft is helpful to you and what you want to see covered in future versions of this publication.

Comments on this publication may be submitted to: applied-crypto-pqc@nist.gov

Public comment period: December 19, 2023 through February 20, 2024

All comments are subject to release under the Freedom of Information Act.

National Cybersecurity Center of Excellence
National Institute of Standards and Technology
100 Bureau Drive
Mailstop 2002
Gaithersburg, MD 20899
Email: nccoe@nist.gov

NATIONAL CYBERSECURITY CENTER OF EXCELLENCE

The National Cybersecurity Center of Excellence (NCCoE), a part of the National Institute of Standards and Technology (NIST), is a collaborative hub where industry organizations, government agencies, and academic institutions work together to address businesses' most pressing cybersecurity issues. This public-private partnership enables the creation of practical cybersecurity solutions for specific industries, as well as for broad, cross-sector technology challenges. Through consortia under Cooperative Research and Development Agreements (CRADAs), including technology partners—from Fortune 50 market leaders to smaller companies specializing in information technology security—the NCCoE applies standards and best practices to develop modular, adaptable example cybersecurity solutions using commercially available technology. The NCCoE documents these example solutions in the NIST Special Publication 1800 series, which maps capabilities to the NIST Cybersecurity Framework and details the steps needed for another entity to re-create the example solution. The NCCoE was established in 2012 by NIST in partnership with the State of Maryland and Montgomery County, Maryland.

To learn more about the NCCoE, visit https://www.nccoe.nist.gov/. To learn more about NIST, visit https://www.nist.gov.

NIST CYBERSECURITY PRACTICE GUIDES

NIST Cybersecurity Practice Guides (Special Publication 1800 series) target specific cybersecurity challenges in the public and private sectors. They are practical, user-friendly guides that facilitate the adoption of standards-based approaches to cybersecurity. They show members of the information security community how to implement example solutions that help them align with relevant standards and best practices, and provide users with the materials lists, configuration files, and other information they need to implement a similar approach.

The documents in this series describe example implementations of cybersecurity practices that businesses and other organizations may voluntarily adopt. These documents do not describe regulations or mandatory practices, nor do they carry statutory authority.

KEYWORDS

algorithm; cryptography; encryption; identity management; key establishment and management; post-quantum cryptography; public-key cryptography; quantum-resistant

ACKNOWLEDGMENTS

We are grateful to the following individuals for their generous contributions of expertise and time.

Name	Organization
Dusan Kostic	Amazon Web Services, Inc. (AWS)
Jake Massimo	Amazon Web Services, Inc. (AWS)

Name	Organization
Avani Wildani	Cloudflare, Inc.
Bruno Couillard	Crypto4A Technologies, Inc.
Jean-Charles	CryptoNext Security
Natasha Eastman	Cybersecurity and Infrastructure Security Agency (CISA)
Garfield Jones	Cybersecurity and Infrastructure Security Agency (CISA)
Nancy Pomerleau	Cybersecurity and Infrastructure Security Agency (CISA)
Judith Furlong	Dell Technologies
Corey Bonnell	DigiCert
Jayaram Chandrasekar	Entrust
Boris Balacheff	HP, Inc.
Tommy Charles	HP, Inc.
Thalia Laing	HP, Inc.
Alyson Comer	IBM
Anne Dames	IBM
Richard Kisley	IBM
Bruce Rich	IBM
Kelsey Holler	Information Security Corporation
Roy Basmacier	Keyfactor
David Hook	Keyfactor
Alexander Scheel	Keyfactor

Name	Organization
Ted Shorter	Keyfactor
Janet Jones	Microsoft
Benjamin Rodes	Microsoft
Lily Chen	National Institute of Standards and Technology (NIST)
David Cooper	National Institute of Standards and Technology (NIST)
Daniel Eliot	National Institute of Standards and Technology (NIST)
Dustin Moody	National Institute of Standards and Technology (NIST)
Andy Regenscheid	National Institute of Standards and Technology (NIST)
Rebecca Guthrie	National Security Agency (NSA)
Mike Jenkins	National Security Agency (NSA)
Brendan Zember	National Security Agency (NSA)
Sean Morgan	Palo Alto Networks Public Sector, LLC
Graeme Hickey	PQShield
Michael Hutter	PQShield
Axel Poschmann	PQShield
Evgeny Gervis	SafeLogic, Inc.
Yoonchan Jhi	Samsung SDS Co., Ltd.
Changhoon Lee	Samsung SDS Co., Ltd.
Marc Manzano	SandboxAQ
Tarun Sibal	SandboxAQ

Name	Organization
Mark Carney	Santander
Daniel Cuthbert	Santander
Jaime Gomez	Santander
Suvi Lampila	SSH Communications Security Corp
Eric Amador	Thales DIS CPL USA, Inc.
Daniel Apon	The MITRE Corporation
Kaitlyn Laohoo	The MITRE Corporation
Neil McNab	The MITRE Corporation
Jessica Walton	The MITRE Corporation
Lee E. Sattler	Verizon
Russ Housley	Vigil Security
David Ott	VMWare
Dimitrios Sikeridis	VMWare

The Technology Partners/Collaborators who participated in this build submitted their capabilities in response to a notice in the Federal Register. Respondents with relevant capabilities or product components were invited to sign a Cooperative Research and Development Agreement (CRADA) with NIST, allowing them to participate in a consortium to build this example solution. We worked with:

Migration to Post-Quantum Cryptography Technology Collaborators		
Amazon Web Services, Inc. (AWS)	Information Security Corporation	Samsung SDS Co., Ltd.
Cisco Systems, Inc.	InfoSec Global	SandboxAQ
Cloudflare, Inc.	ISARA Corporation	Santander

Migration to Post-Quantum Cryptography Technology Collaborators		
Crypto4A Technologies, Inc.	JPMorgan Chase Bank, N.A.	SSH Communications Security Corp
CryptoNext Security	Keyfactor	Thales DIS CPL USA, Inc.
Cybersecurity and Infrastructure Security Agency (CISA)	Microsoft	Thales Trusted Cyber Technologies
Dell Technologies	National Security Agency (NSA)	Utimaco
DigiCert	Palo Alto Networks Public Sector, LLC	Verizon
Entrust	PQShield	VMware, Inc.
HP, Inc.	QuantumXchange	wolfSSL
IBM	SafeLogic, Inc.	

DOCUMENT CONVENTIONS

The terms “shall” and “shall not” indicate requirements to be followed strictly to conform to the publication and from which no deviation is permitted. The terms “should” and “should not” indicate that among several possibilities, one is recommended as particularly suitable without mentioning or excluding others, or that a certain course of action is preferred but not necessarily required, or that (in the negative form) a certain possibility or course of action is discouraged but not prohibited. The terms “may” and “need not” indicate a course of action permissible within the limits of the publication. The terms “can” and “cannot” indicate a possibility and capability, whether material, physical, or causal.

CALL FOR PATENT CLAIMS

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

1. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or
2. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: applied-crypto-pqc@nist.gov

Contents

List of Figures

List of Tables

1 Introduction

In recent years, there has been a substantial amount of research on developing *quantum computers* — machines that exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptographic systems currently in use. This would seriously compromise the confidentiality and integrity of electronically accessible digital information on a global scale. NIST has led an effort to develop standards for cryptographic systems that are secure against both quantum and classical computers and can interoperate with existing communications protocols and networks. NIST's National Cybersecurity Center of Excellence (NCCoE) has initiated a project intended to facilitate and accelerate migration from current quantum-vulnerable cryptography to sufficiently quantum-resistant cryptography.

The question of when a cryptanalytically relevant quantum computer (CRQC) computer will be built is uncertain. While in the past it was less clear that large quantum computers were a physical possibility, many scientists now believe them to merely represent a solvable engineering challenge. Some engineers predict that within the next decade, sufficiently large quantum computers will be built to break essentially all public key schemes currently in use.

It has taken almost two decades to deploy our current public key cryptography infrastructure, and historically, it has taken decades to replace cryptographic algorithms in use in our information systems after they have been determined to be vulnerable to cryptanalysis. Even now, intelligence organizations and criminal organizations are recording cryptographically protected information that is sensitive and has long-term value for future exploitation by quantum computers. Therefore, regardless of whether we can accurately estimate when quantum computing will be sufficiently mature to enable exploitation of current public-key cryptographic systems, we must begin now to prepare our information security systems to be able to resist quantum computing-based attacks.

In 2021, the NCCoE formally initiated its Migration to Post-Quantum Cryptography (PQC) project [1] by issuing an open invitation to commercial and open-source software and hardware technology providers, including those experienced in creating cryptographic technologies, to participate in demonstrating technologies and tools that can provide organizations with insights and findings that support their migrations to PQC.

Together with our project consortium members, this project takes a multi-step approach to providing practical demonstrations supporting timely migration from the current set of public-key cryptographic algorithms to replacement post-quantum cryptographic algorithms that are resistant to quantum computer-based attacks. The project will demonstrate technical actions which are consistent with the steps identified in the "Quantum-Readiness: Migration to Post-Quantum Cryptography" factsheet [2] created in partnership with the U.S. Department of Homeland Security's Cybersecurity & Infrastructure Security Agency (CISA), the National Security Agency (NSA), and NIST:

- Establish a Quantum-Readiness Roadmap
- Prepare a Cryptographic Inventory
- Discuss Quantum-Readiness Roadmaps with Technology Vendors

- Determine Supply Chain Quantum-Readiness

2 Project Scope

The Migration to PQC project includes an industry consortium who met in June 2022 for a kickoff meeting in which each consortium member presented their potential technology and areas of expertise as contributions to the overall project. The project established two workstreams focusing on specific aspects of the migration challenge: the Quantum-Vulnerable Cryptography Discovery Workstream and the Interoperability and Performance of PQC Algorithms Workstream. Interested consortium members engage in the development of the scope and outcome of each workstream.

In the Interoperability and Performance Workstream outlined in this volume, a subset of consortium members contributed working implementations of pre-standardized PQC algorithms in a variety of scenarios, which included the Transport Layer Security (TLS) protocol, Secure Shell (SSH) protocol, and hardware security modules (HSMs). NIST's NCCoE has begun the process of testing pre-standardized post-quantum implementations in a lab environment to ensure that PQC will work in practice before standards are complete and commercial implementations are finalized, in alignment with Office of Management and Budget (OMB) M-23-02 [3]. Where interoperability testing has already been ongoing in other venues, such as the X.509 certificate Internet Engineering Task Force (IETF) hackathon, we leverage and highlight the outcomes from our consortium members in those venues.

Interoperability testing of NIST pre-standardized post-quantum cryptographic algorithms was identified as a core focus area to support the ability of technology vendors and standards bodies to migrate and develop new products that utilize PQC. Organizations that procure systems and software implementing PQC will be able to learn about the quantum-readiness of technologies they are already using and technologies they are procuring to protect their systems. Benchmarking performance metrics from tests in our lab will assist our consortium members and any technology vendor in optimizing their implementations as they move toward production-grade status. Understanding performance metrics of post-quantum-ready algorithms will play a crucial role in motivating technology providers to provide technologies that will enable organizations' migrations, and will provide initial data on which post-quantum cryptographic algorithm is best suited for specific use cases.

The primary audience for this report is cryptographic protocol designers and technology developers/producers responsible for implementation of PQC standards. Secondarily, security architects, system administrators, and others responsible for monitoring the state of implementation of PQC standards in technology may also benefit from this report.

The remainder of this document summarizes the outcomes from the Interoperability and Performance Workstream testing that have occurred thus far, in which we identified the challenging problems and bottlenecks that integrators will face when transitioning systems to post-quantum-ready algorithms. Each section details the test participants, methodologies, and lessons learned from each Interoperability and Performance work item.

3 Testing Scope

For the purposes of interoperability and performance testing, the collaborators agreed on a common scope that enabled them to test their implementations with standards that are commonly used and are expected to or have started to migrate to quantum-safe algorithms. In summary, interoperability testing within this context enables:

- identification of compatibility issues between quantum-ready algorithms;
- resolution of compatibility issues in a controlled, non-production environment; and
- reduction of time spent by individual organizations performing similar interoperability testing for their own migration efforts.

3.1 Selected Post-Quantum Algorithms

Workstream participants experimented with post-quantum algorithms listed below.

- CRYSTALS-Kyber as the preferred post-quantum Key Encapsulation Mechanism (KEM)
- CRYSTALS-Dilithium as the preferred post-quantum signature algorithm
- Falcon as a post-quantum signature algorithm
- SPHINCS+ as a post-quantum signature algorithm picked by NIST at the end of Round 3
- Stateful hash-based signatures standardized in NIST Special Publication (SP) 800-208 [4] tested in the context of HSMs

At the time of this testing, there were limited implementations and evaluations of the candidate KEMs still in the running in the fourth round of NIST's Post-quantum Cryptography Standardization process. As a result, we did not include any Round 4 KEMs or new additional signatures in our experiments. NIST has also requested comments on the standardization of key establishment and digital signature schemes specified in:

- FIPS 203, Module-Lattice-Based Key-Encapsulation Mechanism Standard
- FIPS 204, Module-Lattice-Based Digital Signature Standard
- FIPS 205, Stateless Hash-Based Digital Signature Standard

3.2 Protocols, Standards, and Use-Cases

The protocols, standards, and use-cases outlined in this section were selected to leverage existing work that was prioritized by the participating organizations. These included transport protocols TLS 1.3 (Section 6), SSH (Section 5), and QUIC (Section 7). They also included X.509 certificates (Section 8), which are ubiquitously used for authentication. As many of the participating organizations were HSM manufacturers (Section 9), they also chose to perform interoperability testing of implemented quantum-safe algorithms for HSM use-cases.

Additionally, the collaborators explored the topic of stateful hash-based signatures in Appendix C. In contrast to traditional signature schemes where the input is hashed before signing, new quantum-ready signature schemes can sign arbitrary messages without a pre-hash requirement. In this analysis, we evaluated each approach and summarized the advantages and disadvantages for different use-cases and standards.

3.3 Out of Scope

There are key protocols, standards, and use cases that are not addressed by the initial testing outlined in this document. They were deemed out of scope due to a prioritization effort to make the most efficient use of the resources available. In the bullets below, we offer additional details as to why specific transport protocols were not chosen.

- **TLS 1.2.** There had been work on post-quantum TLS 1.2 [5][6], but retrofitting post-quantum algorithms in TLS 1.2 introduces downgrade concerns where a man-in-the-middle can force the two parties to negotiate classical algorithms even though implementations can support and prefer the PQC algorithms. These concerns are not new. They existed in TLS 1.2 because the data signed in a TLS 1.2 connection does not include the server public key. The IETF recently has been moving towards declaring TLS 1.2 frozen [7], so no new features are expected to make it into the protocol. Thus, we decided to not experiment with PQC and TLS 1.2.
- **IKEv2/IPsec VPNs.** Quantum-safe Internet Key Exchange version 2 (IKEv2) and Internet Protocol Security (IPsec) VPNs have been tested in other efforts [8][9]. Because IKEv2/IPsec VPNs usually stay up for long periods of time and transfer large amounts of data, the performance impact of PQC is considered negligible, as it is amortized over the life of the tunnel.
- **Datagram Transport Layer Security** (**DTLS).** DTLS 1.3 is a protocol similar to TLS 1.3 that runs over UDP. Other than wolfSSL, there were no other PQC DTLS implementations in the project, so we chose not to experiment with it. DTLS is expected to see similar effects to TLS 1.3, but further testing is necessary to confirm that. There have not been enough studies of PQC DTLS by the research community.
- **Message Queuing Telemetry Transport** (MQTT) is a message protocol used in the Internet of Things (IoT) space. Other than wolfSSL's wolfMQTT, no other collaborators supported post-quantum MQTT. Thus, we chose not to experiment with it. Note that MQTT uses TLS for tunnel establishment, so it is expected to see similar impact by the new algorithms as TLS. Depending on how quick and short MQTT transactions are, the impact of PQC may not be amortized as with web TLS connections or with IKEv2/IPsec or SSH tunnels, which transfer larger amounts of data.

The participants also chose to exclude firmware signing and IoT uses, as collaborators were focusing on different technological uses of cryptography at the time.

4 Collaborators and Their Contributions

Organizations participating in this project workstream submitted their capabilities in response to an open call in the Federal Register for all sources of relevant security capabilities from academia and industry (vendors and integrators). The following respondents with relevant capabilities or product components (identified as "Technology Partners/Collaborators" herein) signed a Cooperative Research and Development Agreement (CRADA) to collaborate with NIST in a consortium to provide pre-standardized post-quantum algorithm implementations. Note that not all respondents will have results published in this version of the report.

Amazon Web Services (AWS)

AWS research and engineering efforts focus on the continuation of providing cryptographic security for customers, while developing and testing new cryptographic systems that exceed current customers' demands and protect against projected future adversaries like quantum computing. AWS has invested in the migration to post-quantum cryptography by contributing to post-quantum key agreement and signature schemes to protect customer data, deploying the new algorithms to AWS services, contributing to quantum-safe standardization, and investigating solutions to migration challenges.

Crypto4a

Crypto4A Technologies Inc. is a Canadian cybersecurity technology company providing industry-leading, fifth-generation, quantum-safe, crypto-agile HSM, hardware security platforms (HSPs), and PQC migration solutions. Its products and solutions provide processing capabilities for classic and quantum-safe cryptography that is built in — not bolted on. Crypto4A enables the cryptographic agility, mobility, and scalability demanded by enterprises and government agencies to secure their digital assets and infrastructure while adapting to changing markets, standards, and requirements.

CryptoNext Security

Founded in 2019 by Jean-Charles Faugère after over twenty years of academic research in quantum-resistant cryptography, CryptoNext Security is a pioneer and leading software startup vendor in PQC technology and solutions, having its headquarters based in Paris. CryptoNext and its founders have been fully engaged in the NIST standardization PQC efforts and a participant to the initial 2016 PQC contenders and is pursuing with the current new calls. CryptoNext is a deeply involved member of various IETF PQC-related workgroups and interoperability trials.

CryptoNext offers its Quantum Safe Library (C-QSL), a fully optimized PQC library for various environments, and its Quantum Safe Remediation suite (C-QSR), a multi-layer, natively crypto-agile, PQC standards-compliant and interoperable software suite of technology tools (C-QST) and application plugins (C-QSA) for a broad range of uses such as business applications, secure messaging, HSM, VPN encryptors, PKI, signature and certificate solutions, IoT, and blockchain. CryptoNext works with multiple global industries such as finance, defense, critical infrastructure, and government customers as industry hardware and software technology partners to support them in their post-quantum migration roadmap for long-term efficiency.

Entrust

Entrust keeps the world moving safely by enabling strong identities, secure payments, and protected data. Entrust offers an unmatched breadth of solutions that are critical to the future of secure enterprises, governments, the people they serve, and the data and transactions associated with them. The company is one of the world’s leading providers of high-assurance, PQ-ready network and data security solutions, and pioneered the application of encryption standards decades ago to release the world’s first public key infrastructure.

Entrust is a participating member of the IETF. With NIST recently announcing draft standards for post-quantum cryptography, Entrust has incorporated the proposed quantum-safe algorithms to help organizations prepare for the post-quantum world. The company is working with customers on PQ readiness planning and roadmaps, which includes taking inventory of cryptographic assets; building maturity and crypto agility into management of keys, certificates, and cryptography; and deploying post quantum-ready security infrastructures.

IBM

IBM is one of the largest multinational technology companies with operations in over 170 countries and is known for its research and development, hardware and software products, servers, storage systems, and networking equipment. It also provides consulting, technology, and business services, such as cloud computing, data analytics, and artificial intelligence (AI). IBM's research and development efforts have contributed to numerous technological innovations, including the development of the first programmable computer and now technological breakthroughs in quantum computing.

IBM has scientists and researchers around the globe who deeply believe in the power of the scientific method to invent what's next for IBM, our clients, and the world. Security and cryptography have long been important areas of research. IBM researchers with their academic and industry partners developed three of the four post-quantum cryptographic algorithms to be standardized by NIST.

IBM z16 enterprise server leverages hybrid key agreement schemes and dual signing schemes to protect its infrastructure, and relevant to the project it provides an HSM and software libraries which allow its clients to experiment with FIPS 203 (CRYSTALS Kyber) and FIPS 204 (CRYSTALS Dilithium), two of the primary post-quantum algorithms slated to be standardized. Also, the z16 has been instrumented to support tools which allow users of the cryptographic capabilities of the system to discover the use of vulnerable cryptography, which is an essential step in the migration to quantum-safe algorithms. Additional details about the z16 and the use of the tools for that environment can be found in this IBM Redbook.

Information Security Corporation (ISC)

Since 1989, Information Security Corporation (ISC) has specialized in the design and development of cybersecurity solutions for PKI credential management, confidentiality, authentication, and automated provisioning of relying applications. ISC has developed a variety of certificate lifecycle management applications and cryptographic web services employing classical as well as quantum-safe public key cryptography.

ISC is a member of the OASIS PKCS#11 Technical Committee, several IETF working groups, and various National Information Assurance Partnership (NIAP) Technical Communities that focus on advancing the rapid adoption of standards-based PQC algorithms. ISC's participation in the NCCoE PQC Migration Project includes providing expertise, historical perspective, and interoperability testing between ISC products and other consortium members' certificates and HSM APIs to ensure that customers are able to transition to PQC algorithms as quickly as possible.

Keyfactor

Keyfactor brings digital trust to the hyper-connected world with identity and authentication for every machine, workload, human, and connected thing. By modernizing PKI, automating machine identities, and protecting critical software and product supply chains with secure digital signing and cryptography, Keyfactor helps organizations establish digital trust – then maintain it.

Keyfactor is committed to making quantum-ready PKI, signing, and cryptography solutions available to the world, founding and actively supporting widely adopted open-source projects, including EJBCA, SignServer, and the FIPS 140-validated Bouncy Castle Cryptography APIs. As a participating member of

X9 and the IETF PQC Hackathon, Keyfactor has been following the evolution of NIST PQC standards and has incorporated the proposed algorithms into the Bouncy Castle APIs, which serves as the engine behind their commercial PKI, signing, and certificate management solutions. With quantum-ready solutions and expertise, Keyfactor is working with customers to protect their business and remain resilient in the post-quantum world.

Kudelski IoT

Kudelski IoT is the Internet of Things division of Kudelski Group and provides end-to-end IoT solutions, IoT product design, and full-lifecycle services to IoT semiconductor and device manufacturers, ecosystem creators, and end-user companies. These solutions and services leverage the group's 30+ years of innovation in digital business model creation; hardware, software, and ecosystem design and testing; state-of-the-art security lifecycle management technologies and services; and managed operation of complex systems.

Kudelski IoT is investing in quantum-resistant technology and the migration to PQC, with a broad products and services portfolio and active research contributions. Kudelski IoT is expanding its Security IP portfolio, adding quantum-resistant algorithms, with optimized performance and minimized resource impacts. These algorithms are designed to be upgradable and are also resilient against side-channel and fault attacks. The expansion also involves extending its key management system (keySTREAM) to facilitate quantum-resistant device lifecycle management, supporting customers throughout the migration to post-quantum solutions.

Kudelski IoT has two specialized laboratories that are highly engaged in evaluating the security robustness of algorithms, including quantum-resistant cryptography, by conducting attacks.

Microsoft

Microsoft is committed to providing secure and trustworthy products and services to its customers. As such, Microsoft has been investing in PQC research, development, experimentation, and collaboration since 2014, playing a role in the emergence of PQC and public standards. In particular, Microsoft submitted four algorithms in NIST's standardization effort. Microsoft is proud to participate in the Open Quantum Safe project, where they help develop the liboqs library used in this project and by many PQC industry vendors. Microsoft established the Quantum Safe Program, aiming to accelerate and advance all quantum-safe efforts across the company from both technical and business perspectives.

PQShield

PQShield is a cybersecurity company specializing in PQC, that aims to deliver security and privacy in an increasingly digital world, protecting today's technology from tomorrow's attacks. PQShield was the first company to develop quantum-safe technology on microchips, in applications, and in the cloud, and it is focusing on empowering organizations, industries, and nations with the ultimate quantum-resistant cryptography solutions in software, hardware, and research IP.

PQShield began as a spin-off from the University of Oxford, and has grown to become a world-class collaboration of leading engineers and researchers. With teams in Europe, Japan, the US, and the UK, it is the industry hub of expertise in PQC. PQShield employees are also contributors to the NIST post-quantum cryptography standardization project, with researchers and advisory boards co-authoring the standards announced by NIST. It's contributed multiple cryptographic extensions to RISC-V, the open standard instruction set architecture (ISA) that is gaining traction from proprietary competitors such as ARM and Intel, and is also working with many other organizations such as the World Economic Forum, IETF, ETSI, Groupe Special Mobile Association (GSMA), NCCoE, and GlobalPlatform, to advise and define their positions.

PQShield is committed to helping modernize the cryptographic components and supply chain that keep organizations safe.

Samsung SDS

Samsung SDS provides cloud and digital logistics services. Samsung SDS builds optimized cloud environments with Samsung Cloud Platform and provides all-in-one management service as well as SaaS solutions proven successful in many use cases. One of their core capabilities for delivering their service is cybersecurity, and cryptographic technology plays a fundamental role to enhance security. To this end, Samsung SDS is engaged in various cryptographic research and development activities, including the design, implementation, and architecting of cryptographic techniques, including post-quantum cryptography.

Thales

Thales is the worldwide leader in data security, providing everything an organization needs to protect and manage its data, identities, and intellectual property – through encryption, advanced key management, tokenization, and authentication and access management. Whether it's securing the cloud, digital payments, blockchain, or IoT, security professionals around the globe rely on Thales to confidently accelerate their organization's digital transformation.

Thales has been actively involved in PQC R&D and various standardization efforts since at least 2013. Thales co-authored the Falcon digital signature algorithm, which was selected by NIST as a candidate for PQC standardization in July 2022. The company is engaged in multiple research projects in the United States, France (RISQ), and across Europe, and is also financing numerous doctoral theses on the subject. Additionally, Thales Trusted Cyber Technologies and the NSA signed a CRADA for evaluating the NIST-selected PQC algorithms when operating on an HSM.

Thales Digital Identity and Security (DIS) (a global business area) and Thales Trusted Cyber Technologies (TCT) (a U.S.-based business area exclusively serving the U.S. Federal Government) are both participants in the NCCoE's Migration to PQC Project. Thales has already submitted the products below to the NCCoE lab to help develop practices to ease migration from current algorithms to replacement post-quantum algorithms:

- Thales Luna 7 Hardware Security Module (HSM)
- Thales TCT Luna T-Series HSM (for the U.S. Government)
- Thales CipherTrust Manager for key management
- Thales High Speed Encryptors (HSEs) for network encryption

Implementing both quantum-vulnerable classical public key algorithms and PQC algorithms, the Thales products contributed to the NCCoE PQC project provide the unique capability to be identified as quantum-vulnerable while also providing platforms for PQC interoperability testing. Thales has long been an advocate for crypto agility, facilitating it across its product lines. Existing customer product deployments and Thales contributions to the NCCoE lab can be field-updated with NIST-selected PQC algorithms as they mature through the standardization process. Thales has actively prototyped NIST PQC algorithm finalists within its products and is now focusing on the selected PQC algorithms. In keeping with crypto agility, Thales is now accelerating to practical proof of concepts with customers, notably for hybrid algorithms in digital signatures and key exchange mechanisms.

At Thales, we recognize organizations must adopt a strong post-quantum crypto-agile strategy. In preparation for the transition, Thales encourages organizations to practice crypto agility now, to help your organization evolve and avoid expensive security retrofitting in the future as quantum computing becomes more established. This design principle facilitates changes to the cryptography even after deployment and allows you to prepare for the transition to quantum-safe solutions once the NIST standardization process is completed. To this end, Thales already offers crypto-agile HSMs, key management, and network encryption solutions that you can take advantage of today.

Utimaco

Utimaco is a global platform provider of trusted cybersecurity and compliance solutions and services with headquarters in Aachen (Germany) and Campbell, California (USA). Utimaco develops on-premise and cloud-based HSMs, and solutions for key management, data protection, and identity management, as well as data intelligence solutions for regulated critical infrastructures and public warning systems. Utimaco is one of the world's leading manufacturers in its key market segments.

500+ employees around the globe create innovative solutions and services to protect data, identities, and communication networks with responsibility for global customers and citizens. Customers and partners in many different industries value the reliability and long-term investment security of Utimaco's high-security products and solutions.

Quantum resistance is one of Utimaco´s strategic focus areas. Utimaco´s GP-HSM series "u.trust anchor" and "CryptoServer" provide a trustworthy use of PQC-algorithms and PQC-keys in a secure environment. Hence, Utimaco supports post-quantum relevant use cases either directly or in hybrid mode, to enable a smooth migration of their customers into the post-quantum era.

Utimaco is active in various standardization committees like the European Telecommunications Standards Institute (ETSI), Organization for the Advancement of Structured Information Standards (OASIS) PKCS#11, GSM Association (GSMA), and Accredited Standards Committee (ASC) X9.

wolfSSL

wolfSSL focuses on providing lightweight and embedded security solutions with an emphasis on speed, size, portability, features, and standards compliance. With its SSL/TLS products and crypto library, wolfSSL is supporting high-security designs in automotive, avionics, and other industries. In avionics, wolfSSL supports Radio Technical Commission for Aeronautics Software Considerations in Airborne Systems and Equipment Certification. In automotive, wolfSSL supports MISRA-C capabilities. For government consumers, wolfSSL has a valid FIPS 140-2 certificate. wolfSSL supports industry standards up to the current TLS 1.3 and DTLS 1.3, offers a simple API and an OpenSSL compatibility layer, is backed by the wolfCrypt cryptography library, and provides 24x7 support and much more. wolfSSL's products are open source, giving customers the ability to examine them.

The organizations listed above have contributed technologies described in Table 1. Here, we provide the type of component, product name, and the function the technology will serve in the demonstration.

Table 1 Products and Technologies

Component	Product	Function
Quantum-ready Algorithm Implementation	CryptoNext Quantum Safe Library (C-QSL)	A fully optimized post-quantum library that provides: • NIST-selected post quantum ready algorithms security levels, side-channel protection, and deterministic Random Bit Generator; • Top performance with optimized implementation for most common CPU/operating system platforms and tuning for constrained hardware such as IoT; • Full crypto-agility with the most comprehensive set of PQC algorithms, as well as a full set of language wrappers; and • Evolutionary support for US/EU standards and certifications.
Quantum-ready Protocol Implementation	CryptoNext Quantum Safe Crypto Services (C-QSC)	A set of PQC enabled, optimized, crypto-agile and hybridization-capable implementations of protocols and crypto-objects, including: • Communication protocols such as IKE (IPSec), TLS, and Secure/Multipurpose Internet Mail Extensions (S/MIME); • Programming interfaces such as PKCS#11 libraries; • X.509 post-quantum certificates; and • Identity management.
Quantum-ready Tools and Application Plugins Implementation	CryptoNext Quantum Safe Tools (C-QST) and Application Plugins (C-QSA)	A set of crypto-agile, pure PQ and PQ hybridization-capable, user-transparent, quantum safe integration tools and application plugins for: • Cryptography toolkits • Network infrastructure: IPSec VPN, SSL VPN, SSH • Security infrastructure: PKI, HSM, blockchain • Proxies/connectors • Messaging tools • Web application servers and clients

Component	Product	Function
Quantum-ready Algorithm Implementation	(Microsoft) Open Quantum Safe (OQS) project	An open-source project that aims to support the development and prototyping of quantum-resistant cryptography. OQS consists of two main lines of work: liboqs, an open-source C library for quantum-resistant cryptographic algorithms, and prototype integrations into protocols (TLS and SSH) and applications, including the widely used OpenSSL library. These tools support research by Microsoft and others.
Quantum-ready Algorithm Implementation	aws-lc	A software library implementing cryptographic algorithms for AWS use-cases.
Quantum-ready Algorithm Implementation	(AWS) s2n-tls	A software library implementing the TLS protocol for AWS use-cases.
Quantum-ready Algorithm Implementation	(AWS) s2n-quic	A software library implementing the QUIC protocol for AWS use-cases.
Quantum-ready Algorithm Implementation	AWS SSH implementation	A software library implementing the SSH protocol for AWS use-cases.
Quantum-ready Algorithm Implementation	(crypto4A) QxHSM™	An HSM built around Crypto4A's FIPS Level 3+ QASM™ cryptographic module that provides built-in quantum-safe cryptographic agility. The QxHSM comes in an easy-to-deploy network-attached blade form factor that can accommodates a variety of deployment topologies, be it a single instance (development or root purposes) to multiple instances arranged in either local and/or geo-distributed clusters. The QxHSM can be called via multiple application programming interface (API) standards such as Representational State Transfer (REST), PKCS#11, Key Management Interoperability Protocol (KMIP), Java Cryptography Extension (JCE), and Cryptography API Next Generation (CNG).
Quantum-ready Algorithm Implementation	(crypto4A) QxEDGE™	A fully integrated and hyper-converged HSP that combines Crypto4A'S FIPS Level 3+ QASM quantum-safe crypto-agile cryptographic module with both general-purpose processing engines and confidential compute engines to deliver highly integrated cybersecurity solutions for a diverse set of cybersecurity use cases. Each internal server gets access to their cryptographic services and individual isolated key stores provided by the QASM. The QxEDGE comes in a 19-inch rack 1U server form factor with redundant power supplies.

Component	Product	Function
Quantum-ready Protocol Implementation	(Samsung SDS) s-pqc-tls	A software library that provides the functionality of hybrid key exchanges with classic and PQC cryptography algorithms in the TLS protocol version 1.3 for Java applications, which supports the Java Secure Socket Extension (JSSE) standard API to integrate with existing applications.
Quantum-ready Protocol Implementation	wolfSSL	A software library that implements TLS and DTLS 1.3 supporting quantum-safe symmetric and asymmetric ciphers to be standardized by NIST.
Quantum-ready Protocol Implementation	(wolfSSL) wolfSSH	A software library that implements SSHv2 supporting ecdh-nistp256-kyber-512r3-sha256-d00@openquantumsafe.org for your post-quantum key exchange needs.
Quantum-ready Protocol Implementation	(wolfSSL) wolfMQTT	A software library that implements MQTT up to version 5 and runs on top of wolfSSL, thus leveraging its support for quantum-safe TLS 1.3.
Quantum-ready Protocol Implementation	(wolfSSL) NGINX	A version of NGINX, a high-performance, high-concurrency web server compiled with the wolfSSL cryptographic library.
Quantum-ready Protocol Implementation	(wolfSSL) cURL	A version of cURL, a command-line tool and library for transferring data with URLs compiled with the wolfSSL cryptographic library.
Quantum-ready Algorithm Implementation	Thales Luna A/S790 Network HSM	Helps organizations prepare for a post-quantum future in the following ways: • With a customizable Functionality Module (FM) available today that provides several quantum-resistant algorithms for you to utilize for prototyping; • Using several Thales technology partners that have created their own FM variants that implement these algorithms within their own PQC applications; • Alternatively, create your own FM implementing any of the available quantum-resistant algorithms.
Quantum-ready Algorithm Implementation	Thales TCT Luna T-5000 Network HSM	A dedicated crypto processor designed to protect cryptographic keys. HSMs serve as the trust anchors to protect an organization's cryptographic infrastructure by securely managing, processing, and storing cryptographic keys inside a hardened, tamper-resistant device. The Luna T-Series HSM is FIPS 140-2 L3 validated and CNSS approved. It is the root of trust to numerous partner integrations utilizing asymmetric keys that are

Component	Product	Function
		at risk to the quantum threat. Thales TCT has released firmware for the Luna T-Series HSM that includes pre-standard implementations of NIST-selected PQC algorithms to facilitate PQC interoperability testing.
Quantum-ready Protocol Implementation	Thales CipherTrust Manager & Connectors	Industry-leading enterprise key management solution enabling organizations to centrally manage encryption keys, provide granular access control, and configure security policies. CipherTrust Manager is the central management point for the CipherTrust Data Security Platform. It manages key lifecycle tasks including generation, rotation, destruction, import, and export, provides role-based access control to keys and policies, supports robust auditing and reporting, and offers developer-friendly REST API. CipherTrust Manager is available in both virtual and physical appliances that integrates with FIPS 140-2 compliant Thales Luna or third-party HSMs for securely storing keys with a root of trust. These appliances can be deployed on-premises in physical or virtualized infrastructures and in public cloud environments to efficiently address compliance requirements, regulatory mandates, and industry best practices for data security. With a unified management console, it makes it easy to set policies, discover and classify data, and protect sensitive data wherever it resides using the CipherTrust Data Security Platform products.
Quantum-ready Algorithm Implementation	Thales CN Series Network Encryptors	The Thales High Speed Encryptors (HSE) are widely deployed, FIPS-validated network encryption solutions that encrypt critical network communications and employ quantum-vulnerable classical public key algorithms. The current release includes pre-standard implementations of the NIST-selected PQC algorithms. Thales HSE can be deployed in the NCCoE lab and identified as quantum-vulnerable. Then a firmware upgrade to the most recent version could be applied and the encryptors configured to operate using PQC.
Quantum-ready Algorithm Implementation	(Entrust) PQ-enabled nShield HSM	PQ-enabled nShield HSM supports testing and implementing PQC in a secure HSM.

Component	Product	Function
Quantum-ready Algorithm Implementation	(Entrust) PKIaaS PQ Beta, Quantum-safe Java Toolkit	PKIaaS for Post Quantum Beta is a cloud-based "PKI as a Service" that supports both composite and pure quantum certificate authority (CA) hierarchies. In combination with the Quantum-safe Java Toolkit, this gives the ability to test multi-certificates or composite certificates with applications.
Quantum-ready Algorithm Implementation	(PQShield) PQCryptoLib	A generic software library with a C/C++ interface of FIPS 140-3-ready, post-quantum and classical cryptographic algorithms. It can be used to design your own software development kit (SDK), or be implemented as part of PQShield's SDK, PQSDK. PQCryptoLib is designed to provide post-quantum security using multiple algorithms, including those supported by NIST. The goal of PQCryptoLib is to help organizations transition to quantum-resistant cryptographic schemes by providing support for classical and hybrid key derivation, as well as providing an implementation within the TLS key schedule.
Quantum-ready Algorithm Implementation	(PQShield) PQSDK	Easy-to-use software implementations of both post-quantum and classical cryptographic primitives. It consists of an integration of PQShield's PQCryptoLib library with popular high-level cryptography libraries. PQSDK enables you to experiment with deployments of PQC and to prototype your post-quantum TLS solutions (including TLS X.509) and PKI management before progressing to full deployment.
Quantum-ready algorithm implementation	ISC CDKpqc	A linkable library providing classical and NIST-selected quantum-safe algorithms.
Quantum-ready Certificate Authority	ISC CertAgent	A PQC-enabled X.509 CA.
Quantum-ready Encryption Application	ISC SecretAgent	A PQC-enabled file encryption and digital signature utility.
Quantum-ready Algorithm Implementation	(Kudelski IoT) KSE	A hardware Security Enclaves Portfolio that provides a full range of security and cryptographic services, including quantum-resistant cryptography and classical cryptography, to SoC vendors targeting a high level of robustness and stringent certification schemes with rigorous requirements. The current implementation of quantum-resistant cryptography is upgradable to facilitate adaptation to evolving standards and security countermeasures that have to be completed.

Component	Product	Function
Quantum-ready Algorithm Implementation	(Kudelski IoT) Lab Services	Kudelski IoT is deeply involved in assessing the security robustness of algorithms. This includes the evaluation of quantum-resistant cryptography through the execution of attacks and the analysis of performance data related to the implementation of quantum-resistant algorithms, key management, and other aspects.
Quantum-ready Algorithm Implementation	(Kudelski IoT) keySTREAM	The Kudelski IoT Device Security Lifecycle and keys Management platform for IoT devices. keySTREAM enables provisioning and management of security credentials directly from the cloud to the chipset for the following use-cases: personalization, in-field late provisioning, and in-field credential management. keySTREAM supports asset provisioning for running certain quantum-resistant cryptographic algorithms and is set to undergo an upgrade to cover NIST's range of quantum-resistant cryptographic algorithms.
Quantum-ready Algorithm Implementation	(Keyfactor) Legion of the Bouncy Castle Cryptography APIs	(In partnership with the Legion of the Bouncy Castle Inc.) The Bouncy Castle libraries (for Java, Kotlin, and C#) now include support for both classical and quantum-safe algorithms (upcoming NIST standards included), together with support for protocols such as TLS/DTLS, CMS, Time-Stamp Protocol, OpenPGP, and a variety of protocols around X.509 certificate management.
Quantum-ready Algorithm Implementation	(Utimaco) u.trust anchor	Utimaco's next-generation HSM is designed with a leap forward in security and innovation. u.trust Anchor brings together robust encryption and secure key management, with unprecedented processing power and capabilities within tamper-proof hardware for seamless integrations. Inspired by cloud technology, u.trust Anchor is designed for containerized HSMs. It supports important features like load balancing, high availability, customization of firmware, and total control of each containerized HSM based on business requirements. The customer migration journey is assisted with a software-simulator as well as a full SDK for firmware enhancements.

5 Secure Shell (SSH)

5.1 Interoperability and Performance Discussion

SSH is a widely used protocol for management, configuration, and secure file transfers. The PQC SSH testing prioritized protecting against harvest-now-decrypt-later attacks. We tested a set of PQC key exchange methods to identify gaps and issues. Protecting SSH authentication is considered less urgent since attacks require an active quantum computer during session establishment.

As there is no ratified post-quantum SSH Request for Comments (RFC), we decided to code to version 01 of the current draft [10] which was submitted to the IETF (and has not been picked up for standardization). This draft specifies how to combine elliptic curve cryptography with Kyber, NIST's Round 3 key exchange mechanism, to provide hybrid quantum-safe key exchange methods in SSH. All NCCoE collaborator components implemented the conventions in the draft specification. Some implementations included a subset of the methods at the time of testing.

The collaborator components used for testing SSH were:

- OQS OpenSSH v8
- wolfSSH (June 2023)
- AWS SSH implementation (also used for Secure File Transfer Protocol [SFTP] in AWS Transfer Family)

OQS OpenSSH and wolfSSH were run on Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-72-generic x86_64) with an Intel' XI(R) Gold 6126 CPU @ 2.60 GHz (2 Core) and 32 GB RAM. The AWS SSH implementation was run in Amazon Linux 2 on Intel Xeon Platinum 8175M CPU @ 2.50 GHz with 32 GB RAM.

5.2 Interoperability Testing

5.2.1 PQC Hybrid Key Exchange Test Profile

SSH Testing Profile 1 included the following algorithm parameters:

- Kyber-512, Kyber-768, Kyber-1024
- P256+Kyber-512, x25519+Kyber-512, P384+Kyber-768, P521+Kyber-1024

For each test profile, and for each algorithm supported by both the client and the server, we tested successful SSH connections. Table 2 contains the results of interoperability testing. Key exchange methods not supported by a component at the time of the testing are depicted as "N/A". Table 2 shows that all supported algorithm implementations interoperated between the components.

Table 2 Profile 1 interoperability test results for PQC key exchange in SSH with NCCoE collaborator components

Algorithm Parameters	Client	Server: OQS-OpenSSH	Server: wolfSSH	Server: AWS
Kyber-512	OQS-OpenSSH	Success	N/A	N/A
	wolfSSH	N/A	N/A	N/A
	AWS	N/A	N/A	N/A
Kyber-768	OQS-OpenSSH	Success	N/A	N/A
	wolfSSH	N/A	N/A	N/A
	s2n-tls	N/A	N/A	N/A

Algorithm Parameters	Client	Server: OQS-OpenSSH	Server: wolfSSH	Server: AWS
Kyber-1024	OQS-OpenSSH	Success	N/A	N/A
	wolfSSH	N/A	N/A	N/A
	AWS	N/A	N/A	N/A
P256-Kyber-512	OQS-OpenSSH	Success	Success	Success
	wolfSSH	Success	Success	Success
	AWS	Success	Success	Success
X25519-Kyber-512	OQS-OpenSSH	N/A	N/A	N/A
	wolfSSH	N/A	N/A	N/A
	AWS	N/A	N/A	Success
P384-Kyber-768	OQS-OpenSSH	Success	N/A	Success
	wolfSSH	N/A	N/A	N/A
	AWS	Success	N/A	Success
P521-Kyber-1024	OQS-OpenSSH	Success	N/A	Success
	wolfSSH	N/A	N/A	N/A
	AWS	Success	N/A	Success

5.2.2 PQC Hybrid Key Exchange and Authentication Test Profiles

In terms of PQC SSH authentication, we decided to generate two testing profiles, one that would support the CNSA Suite 2.0 for key exchange and authentication, and one that includes other algorithm combinations.

Profile 2 used Kyber-1024 and Dilithium-4 at level 5, notably excluding hybrids and complying with CNSA 2.0 in the long-term. At the time of the initial testing, only OQS OpenSSH had support for PQC authentication, so we deferred further testing until additional collaborator components had support.

Profile 3 was a profile to test PQC and PQC-hybrid KEMs and authentication algorithm combinations. Given that only OQS OpenSSH supported PQC authentication for SSH at the time of the initial testing, we deferred further testing until additional collaborator components had support.

5.3 Performance Testing

Contrary to TLS 1.3, which was designed to start encryption after one round-trip, SSH as a protocol includes multiple round-trip message exchanges before bringing up the tunnel and exchanging data. That means that most PQC algorithms will not have a significant impact on the overall handshake time, as most of it is spent on the round-trip messages. Even sending more data for authentication will not affect SSH significantly, especially since most SSH connections transfer sizable amounts of data.

Sikeridis et al. evaluated the impact of PQC algorithms to SSH in 2020 [11]. Their study confirmed that Kyber-512, Kyber-768, and Dilithium-4 would have single-digit percentage impact on an SSH handshake at the 50th and 95th percentiles. This confirms the intuition that PQC algorithms will not impact SSH significantly, so we decided against duplicating work and further assessing the performance of PQC SSH for the purposes of this testing.

5.4 Lessons Learned

While collaborators were performing interoperability testing for Profile 1, they had to work through some issues with their implementation components. Below we summarize the lessons learned:

- When working on early implementations of a standard which is not yet ratified, sometimes implementations have to revisit the version of the standard they implement and make changes as the standard evolves to comply with it. For example, one collaborator's SSH component was following an early version of the PQC-hybrid key exchange. After we switched to using the methods in a subsequent draft, the collaborator components could not interoperate. This issue would not occur when implementers start from a ratified, stable specification.
- Implementers could sometimes interpret draft specification details differently. An example is key encoding in the PQC SSH draft [10]. The draft originally did not specify the exact key encodings and representations or was slightly ambiguous, so SSH implementers took different approaches for encoding the keys. Writing prescriptive and clear specifications can limit such issues.
- Using new SSH names, like `ecdh-nistp256-kyber-512-sha256`, in our implementations is prone to introducing interoperability issues for implementations that do not get updated at the same time. Someone supporting `ecdh-nistp256-kyber-512-sha256` in the `-00` version of an early draft specification may not interoperate with an implementation of the `-05` version. Backwards compatibility is important because switching to a new draft could mean the early adopters may no longer be able to use PQC SSH.

  The solution we picked was to use temporary names which are expected to change in the final ratified draft. Every time there is a backwards compatibility breaking change to a method in the draft specification, we introduce a new temporary name specific to the time or the version of the algorithm used. For example, in the first version of the draft which was at the end of Round 3 of the NIST PQC Project, we chose to use `ecdh-nistp256-kyber-512r3-sha256-d00` for combining Elliptic Curve Diffie Hellman (ECDH) P256 with Kyber-512. If the next version of the draft, while in Round 4 of NIST's PQC Project, introduced a change which would break existing implementations of `ecdh-nistp256-kyber-512r3-sha256-d00`, then we would change the SSH method name to `ecdh-nistp256-kyber-512r4-sha256-d01`. New implementations would negotiate with the new method name. Older implementations could still use PQC SSH with implementations that support both the newer and older methods.

  When the specification is ratified, the final standardized name would be different, something like `ecdh-nistp256-kyber-512-sha256`. Support for older, temporary method names can be removed in a phased fashion to allow early implementers to switch to the ratified name. More details about this methodology can be found in the relevant OQS OpenSSH git issue.

6 Transport Layer Security (TLS)

6.1 Interoperability and Performance Discussion

The Transport Layer Security (TLS) protocol is arguably the most deployed online security protocol, so it is critical to make sure it supports post-quantum protection. Moreover, its wide use makes it a prime target for harvest-now-decrypt-later attacks. It is therefore no surprise that TLS has been one of the first protocols on which PQC was prototyped (before even the NIST PQC standardization effort) [12], that numerous academic studies have been performed,[1] and that large-scale industrial experiments[2] have been conducted to study the feasibility of PQC integration and its performance.

Since then, many open-source and commercial TLS 1.3 implementations have added support for PQC and hybrid ciphersuites, even before the availability of the final PQC FIPS standards and their inclusions in the TLS specification. Most implementations (and all the ones by NCCoE collaborating participants) have implemented the draft IETF draft-ietf-tls-hybrid-design-05 [13] specification for hybrid TLS 1.3 key exchange. Our goal was to test interoperability between compliant implementations, and to measure performance between the various algorithms to understand their impact.

It is important to note that we only considered PQC and hybrid key exchange and not authentication (except for the Commercial National Security Algorithm Suite [CNSA] 2.0 profile that tested Dilithium-5 authentication) for two reasons: 1) the pressing record-now-decrypt-later concern only affects encryption (depending on the key exchange part of the session establishment),[3] and 2) there is no industry-wide agreement on how to perform hybrid authentication or if it is necessary (see Section 8).

We tested both the client and server capabilities of the following collaborator components:

- Open Quantum Safe (OQS) OpenSSL Provider
- wolfSSL
- AWS s2n-tls
- Samsung SDS PQC-TLS (s-pqc-tls)
- OQS NGINX

The algorithm identifiers we used for post-quantum negotiations in TLS were the ones defined in OQS OpenSSL[4]. At the time of this testing, draft-ietf-tls-hybrid-design [13] did not have any assigned identifiers, and most collaborator implementations did not support the temporary identifiers defined in draft-kwiatkowski-tls-ecdhe-kyber [14] and draft-tls-westerbaan-xyber768d00 [15], so we chose to only work with the OQS OpenSSL ones.

6.2 Interoperability Testing

We tested two algorithmic profiles for TLS: the first one only uses the key exchange part of the protocol (PQC and hybrid), while the other follows the CNSA Suite 2.0. Following the efforts of the X.509 workstream, we might perform more tests to include PQC and hybrid authentication.

The tests were run in Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-72-generic x86_64) with an Intel Xeon Gold 6126 CPU @ 2.60 GHz (2 Core), 32 GB for RAM virtual instances in the NCCoE lab.

---

[1] See, e.g., Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH (iacr.org).
[2] See, e.g., Google and Cloudflare's public experiment: TLS Post-Quantum Experiment (cloudflare.com).
[3] An attacker would need access to a quantum computer to mount an attack against the authentication portion of the TLS handshake.
[4] https://github.com/open-quantum-safe/oqs-provider/blob/main/ALGORITHMS.md#code-points--algorithm-ids

6.2.1 PQC Hybrid Key Exchange Test Profile

Kyber, the first KEM picked for standardization by NIST, was used in profiles either by itself or in combination with the NIST elliptic prime curve of corresponding strength. The tested key exchange algorithm combinations were:

- Kyber-512, Kyber-768, Kyber-1024
- P256+Kyber-512, P384+Kyber-768, P521+Kyber-1024

For each test profile, and for each supported algorithm by both the client and the server, we tested successful TLS 1.3 connection. Table 3 contains the results of interoperability testing. Key exchange methods not supported by a component at the time of the testing are depicted as “N/A”. Table 3 shows that all supported algorithm implementations interoperated between the components.

Table 3 Profile 1 interoperability test results for PQC key exchange in TLS 1.3 with NCCoE collaborator components

Profile 1	Client	Server: OQS-OpenSSL	Server: wolfSSL	Server: AWS s2n-tls	Server: OQS NGINX	Server: Samsung SDS PQC-TLS
Kyber-512	OQS-OpenSSL	Success	Success	N/A	Success	Success
	wolfSSL	Success	Success	N/A	Success	Success
	AWS s2n-tls	N/A	N/A	N/A	N/A	N/A
	Samsung SDS PQC-TLS	Success	Success	N/A	Success	Success
Kyber-768	OQS-OpenSSL	Success	Success	N/A	Success	Success
	wolfSSL	Success	Success	N/A	Success	Success
	AWS s2n-tls	N/A	N/A	N/A	N/A	N/A
	Samsung SDS PQC-TLS	Success	Success	N/A	Success	Success
Kyber-1024	OQS-OpenSSL	Success	Success	N/A	Success	Success
	wolfSSL	Success	Success	N/A	Success	Success
	AWS s2n-tls	N/A	N/A	N/A	N/A	N/A
	Samsung SDS PQC-TLS	Success	Success	N/A	Success	Success
P256+Kyber-512	OQS-OpenSSL	Success	Success	Success	Success	Success
	wolfSSL	Success	Success	Pending	Success	Success
	AWS s2n-tls		Success	Success	Success	Success
	Samsung SDS PQC-TLS	Success	Success	Success	Success	Success
P384+Kyber-768	OQS-OpenSSL	Success	Success	N/A	Success	Success
	wolfSSL	Success	Success	N/A	Success	Success
	AWS s2n-tls	N/A	N/A	N/A	N/A	N/A

Profile 1	Client	Server: OQS-OpenSSL	Server: wolfSSL	Server: AWS s2n-tls	Server: OQS NGINX	Server: Samsung SDS PQC-TLS
	Samsung SDS PQC-TLS	Success	Success	N/A	Success	Success
P521+Kyber-1024	OQS-OpenSSL	Success	Success	N/A	Success	Success
	wolfSSL	Success	Success	N/A	Success	Success
	AWS s2n-tls	N/A	N/A	N/A	N/A	N/A
	Samsung SDS PQC-TLS	Success	Success	N/A	Success	Success

6.2.2 PQC Hybrid Key Exchange and Authentication Test Profile

Profile 2 used Kyber-1024 and Dilithium at level 5, notably excluding hybrids and complying with CNSA Suite 2.0 in the long-term.

We tested successful TLS 1.3 connection for both the client and the server. Table 4 contains the results of interoperability testing. Methods not supported by a component at the time of the testing are depicted as “N/A”. Table 4 shows that all supported algorithm implementations interoperated between the components.

Table 4 Profile 2 interoperability test results for PQC key exchange and authentication in TLS 1.3 with NCCoE collaborator components

Profile 2	Client	Server: OQS-OpenSSL	Server: wolfSSL	Server: AWS s2n-tls	Server: OQS NGINX	Server: Samsung SDS PQC-TLS
Kyber-1024 / Dilithium5	OQS-OpenSSL	Success	Success	N/A	Success	N/A
	wolfSSL	Success	Success	N/A	Success	N/A
	AWS s2n-tls	N/A	N/A	N/A	N/A	N/A
	Samsung SDS PQC-TLS	N/A	N/A	N/A	N/A	N/A

6.3 Performance Testing

Our performance testing results are discussed below. Note that the goal of this testing is not to compare performance between implementations. We want to compare the impact of the different algorithmic choices within one implementation at a time and observe if the impact of the new algorithms is similar between implementations.

6.3.1 OQS-OpenSSL

We tested performance with OQS OpenSSL for Profiles 1 and 2. The tests were performed using the OQS benchmarking server[5] on the loopback interface, running on an m5n.large AWS instance (Intel Xeon Platinum 8259CL CPU @ 2.50 GHz with 2 CPU and 8 GB of memory). We measured the maximum TLS 1.3 handshake rate, which is shown in Table 5 for Profile 1 and Table 6 for Profile 2.

Table 5 Profile 1 performance test results for PQC key exchange and authentication in TLS 1.3 with NCCoE collaborator components

Security Level	Algorithm (Key Exchange / Auth)	handshake / s
1	Elliptic Curve Diffie-Hellman Exchange (ECDHE) P-256 / Elliptic Curve Digital Signature Algorithm (ECDSA) P-256	1236.67
	Kyber-512 / ECDSA P-256	1591.13
	P256-Kyber-512 / ECDSA P-256	531.67
3	ECDHE P-384 / ECDSA P-384	223.47
	Kyber-768 / ECDSA P-384	681.19
	P384-Kyber-768 / ECDSA P-384	184.44
5	ECDHE P-521 / ECDSA P-521	192.19
	Kyber-1024 / ECDSA P-521	667.65
	P521-Kyber-1024 / ECDSA P-521	109.78

Table 6 Profile 2 performance test results for PQC key exchange and authentication in TLS 1.3 with NCCoE collaborator components

Security Level	Algorithm (Key Exchange / Auth)	handshake / s
5	ECDHE P-521 / ECDSA P-521	192.19
	Kyber-1024 / Dilithium-5	1293.23

These tables can be interpreted as measuring the load on a TLS server. The results show that PQC hybrid can have a significant impact on the maximum connection throughput of a heavily loaded server. We can see that Kyber's performance is high at all security levels. When compared with ECDH with P384 and P521, Kyber-768 and Kyber-1024 render much higher performance. When compared with highly optimized P256, Kyber-512 is slightly less efficient, but of similar performance. In combined PQC hybrid key exchanges, Kyber-512 and ECDH P256 used together have half the handshake throughput, as both algorithms of similar performance are used. When used with non-optimized P384 and P521, Kyber-768 and Kyber-1024 have little impact on the slowdown, as the NIST curves were the bottleneck for these connections.

[5] Handshake performance (openquantumsafe.org)

6.3.2 Samsung SDS PQC-TLS (s-pqc-tls)

The same tests for Profile 1 were conducted with s-pqc-tls on an Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-72-generic x86_64) with an Intel Xeon Gold 6126 CPU @ 2.60 GHz (2 Core) and 32 GB RAM. The connections were taking place over the loopback interface using the widely adopted JSSE in an enterprise IT environment to assess the impact of PQC on performance.

Table 7 Performance test results for PQC key exchange and authentication in TLS 1.3 using Samsung SDS PQC-TLS (s-pqc-tls)

Security Level	Algorithm (Key Exchange / Auth)	handshake / s
1	ECDHE P-256 / ECDSA P-256	333.62
	Kyber-512 / ECDSA P-256	419.18
	P256-Kyber-512 / ECDSA P-256	301.70
	X25519-Kyber-512 / ECDSA P-256	367.86
3	ECDHE P-384 / ECDSA P-384	187.08
	Kyber-768 / ECDSA P-384	259.84
	P384-Kyber-768 / ECDSA P-384	169.08
	X25519-Kyber-768 / ECDSA P-384	242.59
5	ECDHE P-521 / ECDSA P-521	105.35
	Kyber-1024 / ECDSA P-521	157.19
	P521-Kyber-1024 / ECDSA P-521	99.58

As indicated in Table 8, we observe similar behavior as with OQS OpenSSL. Kyber is efficient and performs faster than ECDH, especially for the higher security curves P384 and P521. Combining ECDH with Kyber decreases throughput but not detrimentally. We also see that combining X25519 with Kyber is slightly more efficient than ECDH with Kyber. It is important to emphasize that these results are specific to the test environment, and actual performance may vary depending on the operational environment.

6.3.3 AWS s2n-tls

We tested PQC hybrid key exchange with P256 and Kyber-512, and compared it with X25519 key exchange in TLS 1.3 with s2n-tls. The tests were run on an Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-72-generic x86_64) with an Intel Xeon Gold 6126 CPU @ 2.60 GHz (2 Core), 32 GB for RAM in the NCCoE lab to test.openquantumsafe.org. The round-trip between client and server was 96 ms. Figure 1 shows the mean handshake time and standard deviation for 1000 sequential connections. The server certificate was ECDSA P256 public key signed by an RSA-2048 CA.

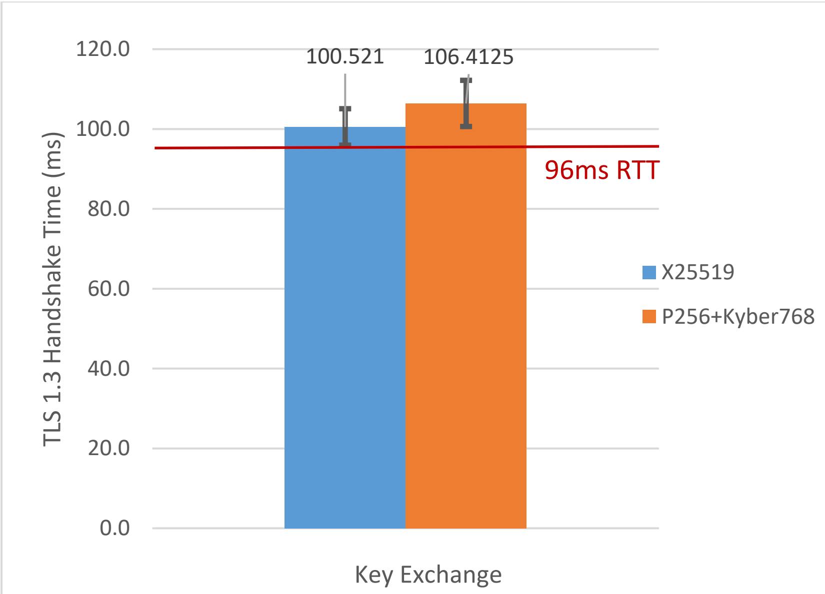

Figure 1 TLS 1.3 PQC hybrid key exchange performance between NCCoE lab s2n-tls clients and OQS server test.openquantumsafe.org

We can see that PQC hybrid TLS handshakes with Kyber-512 and ECDH P256 are a few milliseconds slower than classical ECDH P256 ones. The slowdown due to Kyber is within one standard deviation of the classical key exchange. Kyber-512 is an efficient algorithm and although it slows down these handshakes, the absolute additional time is insignificant for a typical Internet connection. For highly optimized and regional connections, a few milliseconds may be more significant, but for average web or machine-to-machine communications over the internet, the PQC connections will perform satisfactorily. If we consider P384 or P512, which are not optimized like P256, Kyber-768 or Kyber-1024 will have even less impact.

We then tested higher security levels of Kyber in PQC hybrid key exchange in TLS 1.3 with s2n-tls. We compared it with classical key exchange with P256 and P384. The tests were between a client and server with a simulated delay between them to achieve 133ms round-trip time. The server certificate was an ECDSA P256 public key signed by an RSA-2048 CA. Figure 2 shows the mean handshake time and standard deviation for 1000 sequential connections. The standard deviation was negligible because this was a simulated environment between locally connected client and server. The measurements include an extra round trip compared to Figure 1 because we chose to include the TCP handshake time to represent the actual connection experience.

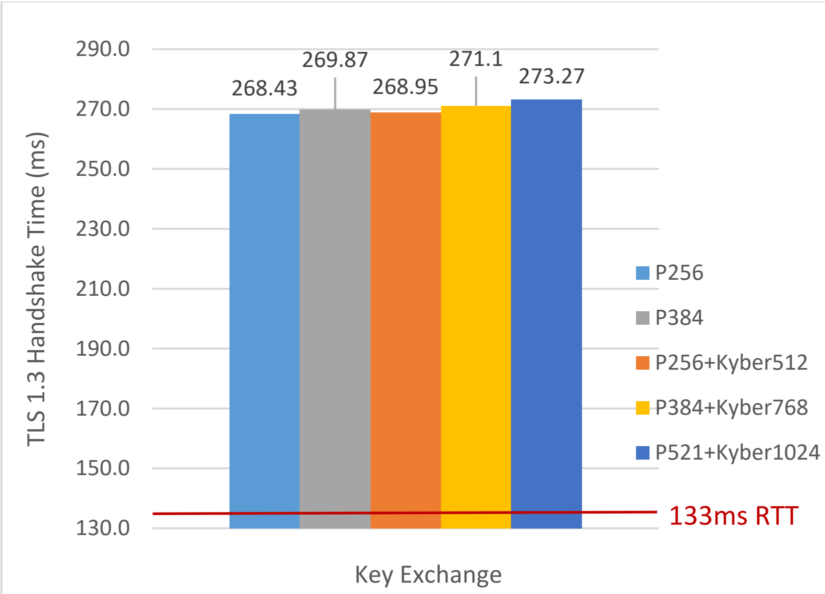

Figure 2 TLS 1.3 PQC hybrid key exchange performance between locally connected s2n-tls client and server using simulated round-trip delay

The results show there is essentially minimal impact on the handshake time. The PQC hybrid exchange even with Kyber-1024 is just a few milliseconds slower than very efficient P256. Such performance differences will not have a noticeable impact on user experience. Lossy conditions could be affected more as Kyber-1024 or Kyber-768 will include more TCP segments, which means higher total loss probability per packet.

To prove this point, we extended the simulation to include a 3% loss probability between the client and the server. The results are in Figure 3.

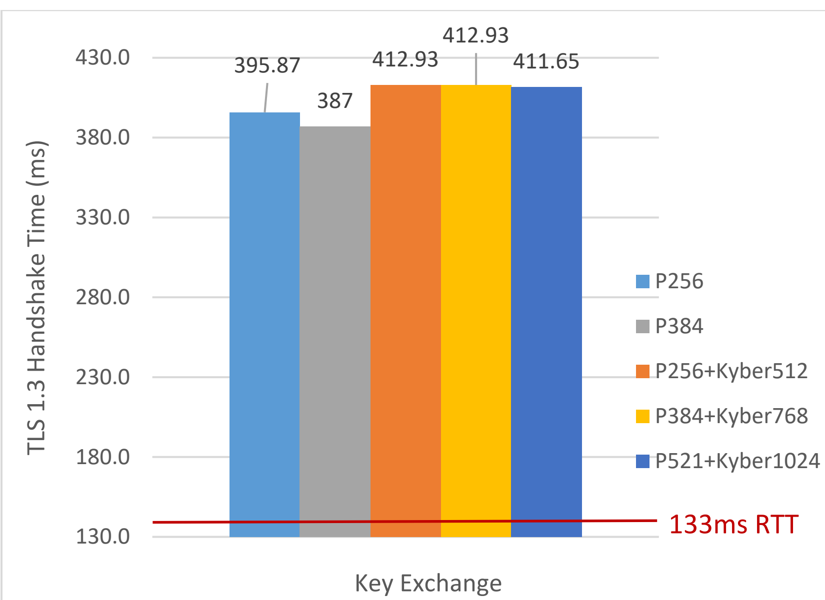

Figure 3 TLS 1.3 PQC hybrid key exchange performance between locally connected s2n-tls client and server using simulated round-trip delay and 3% loss probability

We can see that 3% loss probability leads to almost an extra round-trip's delay. We can also observe that Kyber-768 and 1024's bigger key and ciphertext sizes lead to more losses and higher mean handshake time due to higher loss probability (5.9% instead of 3%). Overall, all these connections are significantly affected by the higher loss probability. The post-quantum handshakes do not seem to be more materially impacted than the classical ones. 20 ms in a handshake that takes 400 ms will not likely be noticed. It is also worth noting that variance for these times was as much as the handshake itself. Higher network losses will completely "randomize" handshake performance. Although limited, the results also show that a 3% packet loss leads to another RTT slowdown in the handshake on average.

In summary, performance testing showed that Kyber is very efficient and when used by itself can slightly speed up handshakes compared to using ECDH. When combining Kyber with ECDH, there is a slight slowdown which will be unnoticeable for most connections. Given that Kyber-768 or Kyber-1024 could be carried over two TCP packets, Kyber could have more impact on lossy connections. These results generally are in line with other performance studies [11][16][17][18] conducted by academia and industry for Kyber and other, less efficient PQC KEMs.

6.4 Lessons Learned

While collaborators were performing interoperability testing, they had to work through some issues with their implementation components. Below we summarize the lessons learned:

- Compliance with older versions of a draft standard specification could cause interoperability issues. As RFC drafts evolve over time, tracking changes to implement them in code takes effort and paying attention to incremental differences in the diffs. This was observed with an issue between s2n-tls and OQS OpenSSL. s2n-tls key shares were compliant with an older version of the draft, but OQS OpenSSL had switched to a more recent version which ended up failing the handshake.
- Using final names or identifiers in implementations is prone to interoperability issues for early implementations that do not all get updated at the same time. Someone supporting TLS 1.3 group 0x2f3a for P256+Kyber-512 in the -00 version of the draft RFC may not interoperate with someone that implements the -05 version. A solution is to use a new temporary group identifier specific to the time or the version of the algorithm, every time there is a backwards compatibility breaking change. Examples include `X25519Kyber768Draft00` and `SecP256r1Kyber768Draft00` assigned for temporary use by draft specifications draft-tls-westerbaan-xyber768d00 [15] and draft-kwiatkowski-tls-ecdhe-kyber [14] until we have the final standards. The shortcoming of this approach is that you may have multiple old codepoints in use which will end up getting deprecated and phased out.
- Supported group order and `key_shares` in the `ClientHello` could lead to unexpected/non-intuitive key exchanges. For example, a client sending a `key_share` for only X25519+Kyber-512 and advertising support for X25519+Kyber-512, P256+Kyber-512, X25519, and P256 could negotiate plain X25519 with the server because the server does not support X25519+Kyber-512. This was not a violation of the TLS 1.3 standard, but we were expecting that the client would negotiate P256+Kyber-512 after seeing a Hello Retry Request with P256+Kyber-512.

7 QUIC

7.1 Interoperability and Performance Discussion

QUIC is a widely used protocol for the web, video, and streaming. Because QUIC uses TLS 1.3 to establish its shared keys, testing QUIC heavily depended on PQC TLS 1.3. Our PQC QUIC testing prioritized protecting against harvest-now-decrypt-later attacks, so we wanted to test a set of PQC key exchange methods to identify gaps and issues. Protecting QUIC authentication is considered less urgent since attacks require an active quantum computer during session establishment. We generated testing profiles for PQC key exchange and authentication, which has not been implemented by all vendors and thus received limited interoperability testing. On the other hand, PQC authentication would have more significant impact on QUIC's performance, so we focused more on authentication for our performance testing.

The collaborator component used for testing QUIC was:

- AWS s2n-quic implementation (built with s2n-tls and AWS-LC)

The algorithm identifiers for post-quantum negotiations in TLS 1.3 (used in QUIC) were the ones defined in OQS OpenSSL[6]. At the time of this testing, draft-ietf-tls-hybrid-design [13] did not have any assigned identifiers, and collaborator implementations did not support the temporary identifiers defined in draft-kwiatkowski-tls-ecdhe-kyber [14] and draft-tls-westerbaan-xyber768d00 [15], so we chose to only work with the OQS OpenSSL ones.

7.2 Interoperability Testing

7.2.1 PQC Hybrid Key Exchange Test Profile

QUIC establishes its encrypted tunnels over UDP with AES-GCM or Chacha20/Poly1305 as the authenticated encryption algorithm. It uses keys negotiated in TLS 1.3 sent over QUIC frames. Thus, we had to use a quantum-safe version of TLS 1.3 to ensure the encryption in QUIC is quantum-safe. As there is no ratified post-quantum TLS 1.3 RFC, we decided to code to the draft IETF draft-ietf-tls-hybrid-design-05 [13] specification for hybrid TLS key exchange in QUIC as we did with our TLS testing.

The QUIC Profile included the following algorithm parameters:

- Kyber-512, Kyber-768, Kyber-1024
- P256+Kyber-512, x25519+Kyber-512, P384+Kyber-768, P521+Kyber-1024

At the time of the initial testing, there was only one PQC implementation of the protocol, s2n-quic. Thus, we were not able to complete any interoperability testing. oqs-demos/quic may be tested with s2n-quic in the future as a QUIC integration with OQS.

[6] https://github.com/open-quantum-safe/oqs-provider/blob/main/ALGORITHMS.md#code-points--algorithm-ids

7.2.2 PQC Hybrid Key Exchange and Authentication Test Profiles

In terms of PQC authentication, we decided to generate one more testing profile which supports both PQC hybrid key exchange and authentication combinations. The QUIC Profile 2 included:

- ECDHE-Kyber hybrid (L1: P256-512, L3: P384-768, L5: P521-1024).
- Auth: Dilithium-2, Dilithium-3, Dilithium-4

We were not able to complete any interoperability testing for this profile at the time of the initial testing because only one collaborator component, s2n-quic, supported PQC QUIC. oqs-demos/quic may be tested with s2n-quic in the future as a QUIC integration with OQS.

7.3 Performance Testing

Post-quantum key exchange has been extensively tested in TLS 1.3 connections with Kyber. The impact of 0.8-1.2 KB with Kyber-512 or Kyber-768 key shares will be insignificant for regular TLS or QUIC connections. Thus, we wanted to evaluate the impact of post-quantum authentication in QUIC performance. Post-quantum authentication adds more complexity to QUIC connections, as it interferes with QUIC Amplification Protection and Congestion Control mechanisms. As this had not been evaluated before, to the best of our knowledge, we chose to use s2n-quic's netbench benchmarking tool. Our measurements evaluated the following QUIC key exchange and authentication options:

- client-server with X25519+Kyber-512 and 2048-bit RSA certificates with one intermediate CA
- client-server with X25519+Kyber-512 and 18 KB PEM encoded cert chain (Dilithium-2 WebPKI equivalent with one intermediate CA)
- client-server with X25519+Kyber-512 and 10 KB PEM encoded cert chain (Dilithium-2 WebPKI equivalent omitting the intermediate CA)
- client-server with X25519+Kyber-512 and 22 KB PEM encoded cert chain (Dilithium-3 WebPKI equivalent with one intermediate CA)

Note that due to lack of support of Dilithium in s2n-tls/s2n-quic at the time of the testing, the big certificate chains were using specially crafted, bloated RSA certificates of similar size to Dilithium-2, 3 WebPKI certificates (with 2 Signed Certificate Timestamps [SCTs]). Given the performance of Dilithium, this emulation is expected to be very close to using Dilithium certificates themselves.

The parameters tweaked while testing included:

- QUIC's initial congestion window (`initcwnd`), which can introduce a round-trip if the Dilithium authentication data from the server exceeds ~14 KB
- QUIC's amplification window, which can introduce a round-trip if the Dilithium authentication data from the server exceeds ~3.6 KB
- QUIC's initial round-trip estimate (`kInitialRtt`), which could cause connection slowdowns due to packet pacing (Section 7.7 of RFC 9002 [19]). This parameter was not part of the initial testing, but we observed pacing was affecting these connections and decided to study it more.

Our experiments measured the QUIC handshake time for the `connect netbench` scenario, which creates 1000 connections with 1-byte bidirectional streams before it closes each connection down. The client and server were run in Amazon Linux 2 running on Intel Xeon Platinum 8175M CPU @ 2.50 GHz with 32 GB RAM. s2n's benchmark utility, `netbench`, plotted the results for the scenario. Figure 4 shows the handshake time for:

- a classical handshake with RSA-2048 certificate chains with one intermediate CA;
- a handshake with ~10 KB authentication data which corresponds to a Dilithium-2 leaf WebPKI certificate (assuming the ICA was omitted to trim down the data as per draft-kampanakis-tls-scas-latest-03 [20] or draft-jackson-tls-cert-abridge-00 [21];
- a handshake with an 18 KB Dilithium-2 leaf WebPKI certificate chain with one intermediate CA; and
- a handshake with a 22 KB Dilithium-3 leaf WebPKI certificate chain with one intermediate CA.

The client-server round-trip for the experiments was about 60 ms.

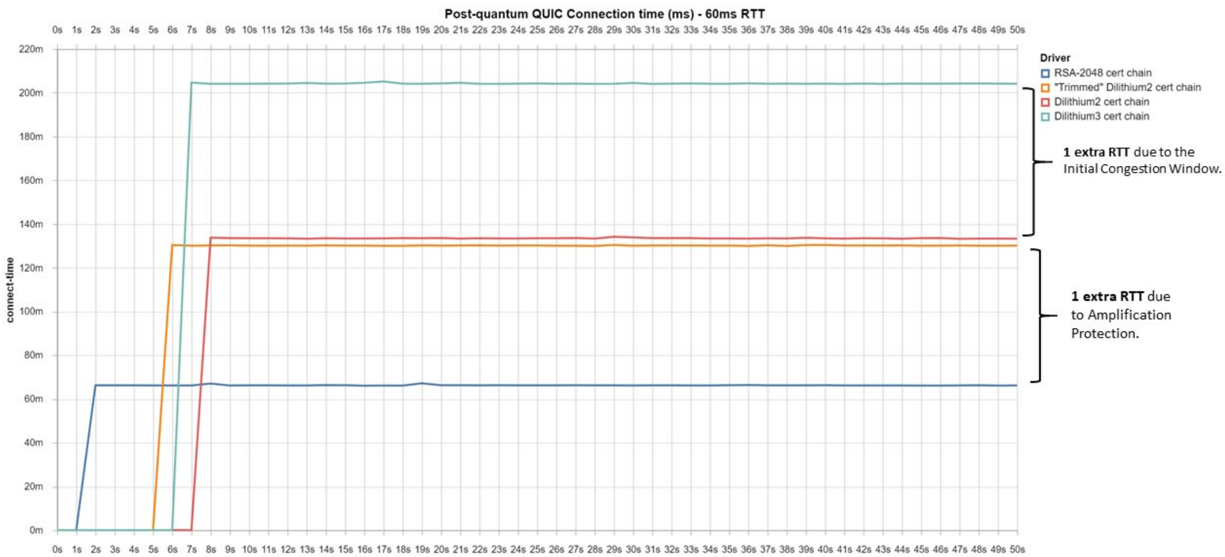

Figure 4 QUIC handshake time with classical and Dilithium-2, 3 WebPKI with QUIC's default congestion control (~14 KB), default initial round-trip `kInitialRtt` (333 ms), and amplification protection (3x)

We can see that the classical handshake completes in just one round trip (typical TLS 1.3 1-RTT). The 10 KB cert introduces one extra round trip due to QUIC's amplification window (~3.6 KB). Amplification protection also introduces a round trip for the 18 KB chain. Congestion control does not add a second round-trip in this case because the initial congestion window has not filled up after the client acknowledges the first 3.6 KB. The 22 KB chain ends up including two round trips, one from amplification protection and one from congestion control.

We then wanted to evaluate how tweaking QUIC network parameters could speed up these handshakes by eliminating the round trips. Figure 5 shows the times for PQC QUIC handshakes with a Dilithium-3 WebPKI certificate chain with one intermediate CA. We measured the handshake time with the following combinations of amplification window (depicted as Amp), initial congestion window `initcwnd` (depicted as icwnd), and the initial QUIC RTT `kInitialRtt` (depicted as irtt).

- Amp=3.6 KB, icwnd=14 KB, irtt=333

- Amp=20 KB, icwnd=14 KB, irtt=333
- Amp=3.6 KB, icwnd=25 KB, irtt=333
- Amp=20 KB, icwnd=25 KB, irtt=333
- Amp=20 KB, icwnd=25 KB, irtt=50

The default corresponding values are 3x the client request size (~3.6 KB), 10x the maximum datagram size (~14 KB) as per [Section 7.2 of RFC 9002][19], and 333 ms as per [Section 6.2.2 of RFC 9002][19].

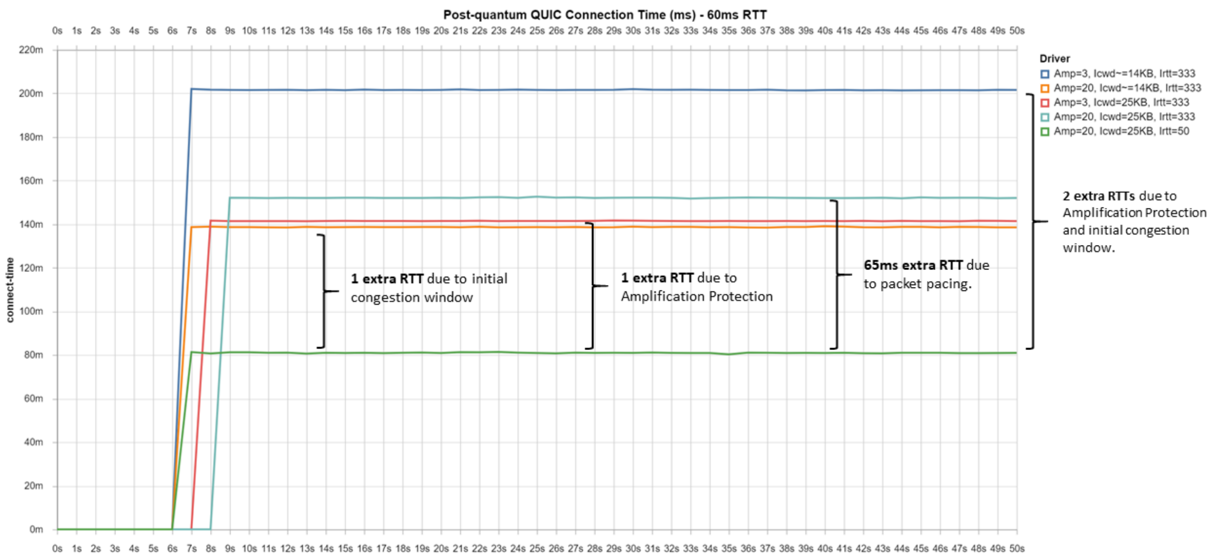

Figure 5 PQC QUIC handshake time with PQC hybrid key exchange and Dilithium-3 WebPKI equivalent signatures with various QUIC amplification window, `initcwnd` and `kInitialRtt`

We can see that the amplification window adds a round-trip, and increasing it eliminates the extra round-trip. The same goes for the initial congestion window. We notice that when increasing both Amp and icwnd, we still see an extra 65 ms slowdown. This is due to irtt and QUIC's packet pacing. Packet pacing is built to prevent packet bursts which could trigger short-term packet loss. While the server does not know the RTT from the client, it uses the initial default value of 333 ms and calculates the time needed to pause after sending 10 packets. The pausing time ends up amounting to 65-70 ms. The effects of packet pacing are not experienced when we have an extra round trip due to amplification protection or congestion control because the server has a more accurate estimate of the RTT after it observes the round trip, and packet pacing has little impact on the handshake. When dropping irtt to a more realistic value (50 ms), packet pacing pauses much less and thus the handshake completes in almost one round-trip as expected.

It is clear that QUIC's network parameters affect the impact that PQC authentication will have on these handshakes. To prevent the extra round trips, we would need to significantly increase the amplification window, which increases amplification attack risks. We would also need to increase the initial congestion window, which could affect network congestion. We would need to increase the initial RTT to a more realistic value instead of the default assumed 333 ms, so that packet pacing does not affect the handshake by 65 ms or more. Other mechanisms to alleviate these handshakes include trimming the authentication data sent or using validation tokens. These changes, although possible, should not be taken lightly, as they have tradeoffs. Some of these options are also discussed in Section 2 of a recent vision paper [22].

At the time of the initial testing, we focused on experimenting with PQC certificate sizes and QUIC network parameters. Future testing could include varying the network hops and loss probabilities between client and server to investigate performance in different network conditions. We may also look into the time-to-last-byte per QUIC connection instead of time-to-first-byte (handshake time) to more accurately evaluate the impact PQC certs would have on user experience.

7.4 Lessons Learned

While experimenting with QUIC performance, we identified issues we had not anticipated. Below we summarize the lessons learned:

- Although we expected the extra round trips due to QUIC amplification protection and congestion control, we did not anticipate that the initial RTT would add 65 ms to the handshake after eliminating the other round trips. We learned that experimentation can sometimes reveal issues which theoretical intuition does not. Hands-on experiments should be used before evaluating technical solutions.
- While testing QUIC connections with s2n-quic's `netbench`, we noticed that when the data transferred was much larger than the authentication data in the handshake, the impact of PQC decreased. We did not collect the results of these experiments because `netbench` did not fully support them, but we noticed this while capturing data. So far, like we also did in the experiments above, researchers have been measuring the handshake time for a PQC connection and comparing it to classical connections. We have been showing that in the worst of these connections, PQC affects the handshake more, which could be inaccurate. The tail-ends of these measurements may be overestimating the impact. For example, mobile clients perform ~12 connections per page to fetch ~2 MB of total data on average. That means that each TCP connection carries ~160 KB of data. A bad connection, which suffers from 20 KB extra PQC authentication data, is likely to be suffering already carrying all 160 KB per connection. We may just not be measuring it because we have been focusing on the time-to-first-byte. Our research efforts, when evaluating PQC impact on transport protocols, should focus on the time-to-last-byte or the time-to-mid-byte, which will be more indicative of what a user would notice.

8 X.509

8.1 Interoperability and Performance Discussion

8.1.1 Introduction

X.509 certificates will be important artifacts in the migration process towards PQC, as they are the main way to transport and communicate public keys between endpoints. X.509 certificates can be used to carry signature or encryption keys and are therefore used in protocols such as TLS/SSL, QUIC, S/MIME, and IPsec.

Many formats have been proposed to adapt the current X.509 certificate structure to PQC. Some of them are pure PQC artifacts transporting only a PQC public key and signed by a PQC signature algorithm, while others are hybrid artifacts including both traditional and PQC public keys and signed by both traditional and PQC signature algorithms.

The different X.509 certificate formats that have been tested are clarified in Section 8.1.2.

8.1.2 X.509 Certificate Formats

8.1.2.1 PURE PQC

This X.509 certificate is a pure PQC certificate, meaning that it only contains the PQC material (PQC key and PQC signature). It uses the legacy X.509 structure and replaces with traditional objects with quantum-safe objects ones:

- For the algorithm identifier, new OIDs for post-quantum algorithms and parameter sets are used.
- Keys and signatures follow the usual ASN.1 syntax, except the byte string corresponds to a post-quantum object.

The details of this format can be found in the following documents:

- RFC 5280 [23]
- draft-ietf-lamps-dilithium-certificates [24]
- draft-ietf-lamps-kyber-certificates [25]

8.1.2.2 HYBRID CONCATENATED

This X.509 certificate is a hybrid certificate. It basically concatenates the classic and post-quantum objects without changing the structure of the ASN.1 tree:

- For the algorithm identifier, a specific OID for the selected combination of traditional + post-quantum algorithm is used.
- Keys and signatures follow the usual ASN.1 syntax, except the byte string corresponds to the concatenation of a traditional and a post-quantum object.

There is no specification available so far.

8.1.2.3 HYBRID BOUND

This X.509 certificate is a hybrid certificate. It uses two certificates, one traditional and one post-quantum. The traditional certificate is built as usual, and the post-quantum certificate is built according to the PURE PQC model (see Section 8.1.2.1). In addition, the post-quantum certificate contains an extension that links itself to the traditional certificate. The traditional certificate may contain a similar extension linking to the post-quantum certificate, so that each certificate has an authenticated pointer to the other.

The details of this format can be found in the following:

- draft-becker-guthrie-cert-binding-for-multi-auth [26]

8.1.2.4 HYBRID COMPOSITE

This X.509 certificate is a hybrid certificate. This version is a refinement of the previous HYBRID CONCATENATED format (see Section 8.1.2.2) that uses ASN.1 encoding to separate the traditional and post-quantum objects.

- The algorithm identifier is a special OID for "composite".
- Keys and signatures are "composite" objects: a composite public key is an ASN.1 sequence of public key fields, each with its own algorithm identifier and contents (and similarly for the signature).

The details of this format can be found in the following documents:

- draft-ounsworth-pq-composite-sigs [27]
- draft-ietf-lamps-pq-composite-kem [28]

8.1.2.5 HYBRID USING EXTENSIONS (Catalyst)

This X.509 certificate is a **hybrid** certificate. This format stores the post-quantum objects in X.509 extensions. Except for these extensions, the certificate looks exactly like a traditional X.509 certificate, so an unmodified tool should be able to parse and verify it, assuming it treats unknown non-critical extensions as opaque data. In principle, this format is therefore retro-compatible.

The details of this format can be found in the following documents:

- draft-truskovsky-lamps-pq-hybrid-x509 [29]
- ITU-T X.509 (10/2019) [30]

8.1.2.6 HYBRID DELTA EXTENSIONS (Chameleon)

This X.509 certificate is a hybrid certificate. This format encodes the differences between two certificates in a single extension. One certificate is the "base" or outer certificate, and the second one is the "delta" or inner certificate. Only the differences between the base and delta certificate are contained in the extension. Except for the extension, the certificate looks exactly like a traditional X.509 certificate, so an unmodified tool should be able to parse and verify it, assuming it treats unknown non-critical extensions as opaque data. In principle, this format is therefore retro-compatible. A delta certificate can be reconstructed by the base certificate into a fully verifiable secondary certificate.

The details of this format can be found in the following:

- draft-bonnell-lamps-chameleon-certs [31]

8.2 Interoperability Testing

8.2.1 Testing Procedure

An interoperability test aims at verifying that:

- all the public keys contained in the X.509 certificate can be extracted and used by another vendor application; and

- all the signatures contained in the X.509 certificate can be verified by another vendor application.

The most basic interoperability testing between applications A and B consists of the following steps:

In case of a SIG algorithm:

1. Application A generates a Root CA certificate (self-signed).
2. Application B verifies the Root CA certificate.

This test checks both the PQC public key usability and the PQC signature correctness.

In case of a KEM algorithm:

1. Application A generates an end-entity certificate holding a KEM key. This certificate is signed by the private key of the Root CA certificate generated in the signature algorithm test case.
2. Application B verifies the end-entity certificate holding the KEM key.

See https://github.com/IETF-Hackathon/pqc-certificates/tree/master#zip-format-r3.

8.2.2 Test Profiles

8.2.2.1 PURE_PQ_SIG

The PURE_PQ_SIG test profile tests a PURE PQ X.509 certificate transporting a PQC signature key. This test profile includes the algorithm configurations listed in Table 8:

Table 8 Algorithm configurations included in the PURE_PQ_SIG test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
Dilithium-2 (ML-DSA-44-ipd)	Dilithium-2 (ML-DSA-44-ipd)
Dilithium-3 (ML-DSA-65-ipd)	Dilithium-3 (ML-DSA-65-ipd)
Dilithium-5 (ML-DSA-87-ipd)	Dilithium-5 (ML-DSA-87-ipd)
Falcon-512	Falcon-512
Falcon-1024	Falcon-1024
SPHINCS+-SHAKE-128f (SLH-DSA-SHAKE-128f-ipd)	SPHINCS+-SHAKE-128f (SLH-DSA-SHAKE-128f-ipd)
SPHINCS+-SHAKE-192f (SLH-DSA-SHAKE-192f-ipd)	SPHINCS+-SHAKE-192f (SLH-DSA-SHAKE-192f-ipd)
SPHINCS+-SHAKE-256f (SLH-DSA-SHAKE-256f-ipd)	SPHINCS+-SHAKE-256f (SLH-DSA-SHAKE-256f-ipd)
SPHINCS+-SHA2-128f (SLH-DSA-SHA2-128f-ipd)	SPHINCS+-SHA2-128f (SLH-DSA-SHA2-128f-ipd)
SPHINCS+-SHA2-192f (SLH-DSA-SHA2-192f-ipd)	SPHINCS+-SHA2-192f (SLH-DSA-SHA2-192f-ipd)
SPHINCS+-SHA2-256f (SLH-DSA-SHA2-256f-ipd)	SPHINCS+-SHA2-256f (SLH-DSA-SHA2-256f-ipd)

8.2.2.2 PURE_PQ_KEM

The PURE_PQ_KEM test profile tests a PURE PQ X.509 certificate transporting a PQC KEM key. This test profile includes the algorithm configurations listed in Table 9:

Table 9 Algorithm configurations included in the PURE_PQ_KEM test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
Kyber-512	Dilithium-2 (ML-DSA-44-ipd)
Kyber-768	Dilithium-3 (ML-DSA-65-ipd)
Kyber-1024	Dilithium-5 (ML-DSA-87-ipd)

8.2.2.3 HYBRID_CONCATENATED

The HYBRID_CONCATENATED test profile tests a HYBRID CONCATENATED X.509 certificate transporting a PQC SIG key. This test profile includes the algorithm configurations listed in Table 10:

Table 10 Algorithm configurations included in the HYBRID_CONCATENATED test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
RSA (3072)+Dilithium-2	RSA_PKCSv1.5_SHA256 (3072)+Dilithium-2
ECDSA (P-256)+Dilithium-2	ECDSA_SHA256 (P-256)+Dilithium-2
ECDSA (P-521)+Dilithium-5	ECDSA_SHA512 (P-521)+Dilithium-5

8.2.2.4 HYBRID_BOUND

The HYBRID_BOUND test profile tests a HYBRID BOUND X.509 certificate transporting a PQC SIG key. This test profile includes the algorithm configurations listed in Table 11:

Table 11 Algorithm configurations included in the HYBRID_BOUND test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
RSA (3072)+Dilithium-2	RSA_PKCSv1.5_SHA256 (3072)+Dilithium-2
ECDSA (P-256)+Dilithium-2	ECDSA_SHA256 (P-256)+Dilithium-2
ECDSA (P-521)+Dilithium-5	ECDSA_SHA512 (P-521)+Dilithium-5

8.2.2.5 HYBRID_COMPOSITE

The HYBRID_COMPOSITE test profile tests a HYBRID COMPOSITE X.509 certificate transporting a PQC SIG key. This test profile includes the algorithm configurations listed in Table 12:

Table 12 Algorithm configurations included in the HYBRID_COMPOSITE test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
RSA (3072)+Dilithium-2	RSA_PKCSv1.5_SHA256 (3072)+Dilithium-2
ECDSA (P-256)+Dilithium-2	ECDSA_SHA256 (P-256)+Dilithium-2
ECDSA (P-521)+Dilithium-5	ECDSA_SHA512 (P-521)+Dilithium-5

8.2.2.6 HYBRID_CATALYST

The HYBRID_CATALYST test profile tests a HYBRID USING EXTENSIONS X.509 certificate transporting a PQC SIG key. This test profile includes the algorithm configurations listed in Table 13:

Table 13 Algorithm configurations included in the HYBRID_CATALYST test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
RSA (3072)+Dilithium-2	RSA_PKCSv1.5_SHA256 (3072)+Dilithium-2

X.509 Public Key Algorithm	X.509 Signature Algorithm
ECDSA (P-256)+Dilithium-2	ECDSA_SHA256 (P-256)+Dilithium-2
ECDSA (P-521)+Dilithium-5	ECDSA_SHA512 (P-521)+Dilithium-5

8.2.2.7 HYBRID_CHAMELEON

The HYBRID_CHAMELEON test profile tests a HYBRID DELTA EXTENSIONS X.509 certificate transporting a PQC SIG key. This test profile includes the algorithm configurations listed in Table 14:

Table 14 Algorithm configurations included in the HYBRID_CHAMELEON test profile

X.509 Public Key Algorithm	X.509 Signature Algorithm
RSA (3072)+Dilithium-2	RSA_PKCSv1.5_SHA256 (3072)+Dilithium-2
ECDSA (P-256)+Dilithium-2	ECDSA_SHA256 (P-256)+Dilithium-2
ECDSA (P-521)+Dilithium-5	ECDSA_SHA512 (P-521)+Dilithium-5

8.2.3 Test Results

All interoperability tests have been performed within the IETF PQC X.509 Hackathon. The results can be found here: IETF Hackathon Results. [32]

8.2.3.1 PURE_PQ_SIG

See IETF Hackathon Results.

8.2.3.2 PURE_PQ_KEM

See IETF Hackathon Results.

8.2.3.3 HYBRID_CONCATENATED

See IETF Hackathon Results.

8.2.3.4 HYBRID_BOUND

See IETF Hackathon Results.

8.2.3.5 HYBRID_COMPOSITE

See IETF Hackathon Results.

8.2.3.6 HYBRID_CATALYST

See IETF Hackathon Results.

8.2.3.7 HYBRID_CHAMELEON

See IETF Hackathon Results.

8.3 Performance Testing

Performance was not investigated during this first testing phase. It will be investigated in future phases of interoperability testing.

8.4 Lessons Learned

While collaborators were going through interoperability testing, they had to work through issues with their implementation components. Below we summarize the lessons learned:

- Falcon signature has variable size and caused some issues.
- There is a lot of interest in hybrid certificate formats. For example, we have seen many implementations of composite signature signed certificates. There is also interest in using certificate structures to convey hybrid information in the following X.509 certificate components:
  - New V3Extensions types:
    - The Chameleon Delta Certificate Descriptor Extension (DCD)
    - The RelatedCertificate Extension for multi-certificate authentication
    - The Catalyst AltSignature and AltPublicKey extension
  - Existing V3Extensions:
    - Using the Subject Info Access (SIA) extension for certificate discovery
  - SubjectPublicKey
    - Composite keys
    - An external public key structure
  - Signature structures
    - Composite signatures
- ASN.1 encoding issues – some specifications had different encodings for Dilithium keys (for example). We settled on using OCTET_STRING, which looks like the way the standards are going.
- Having OIDs to reference specification versions is critical. Changing specifications has required numerous updates to the prototype OIDs to try and avoid compatibility issues. See OID Mapping table for the latest prototype OIDs.
- PEM and Distinguished Encoding Rules (DER) encoding issues sometimes caused edge cases, which required special parsers.
- X509-related issues like basic constraints rules and encodings sometimes came into play. We recognized that these are not PQ algorithm-related; it just means X.509 in general can be difficult to implement and work together.

9 Hardware Security Modules (HSMs)

9.1 Discussion about Interoperability and Performance

HSMs serve as foundational elements in establishing the online trust required to facilitate digital commerce and identity in today's connected world. They provide hardware-based protection for high-value cryptographic assets and perform complicated cryptographic processing using those assets. Given their foundational nature and the value of the assets they protect, it is imperative that we be able to migrate HSMs from the current classic cryptographic mechanisms such as RSA and ECC over to the next generation of PQC algorithms such as Kyber, Dilithium, Falcon, eXtended Merkle Signature Scheme (XMSS)/Multi-Tree eXtended Merkle Signature Scheme (XMSS^{MT}), Leighton-Micali Signature (LMS)/Hierarchical Signature System (HSS), and SPHINCS+.

HSMs are available from a number of vendors, all of whom must ensure the cryptographic keys they are generating and consuming, as well as the cryptographic algorithms they are performing, are compatible with HSMs from other vendors to ensure the system as a whole can function, providing a solid and secure foundation for many of the digital systems we rely on today.

As such, we have endeavored to validate the ability for HSMs to interoperate in the following ways:

- Public keys generated on one vendor's HSMs can be successfully exported and then imported into another vendor's HSM to create a valid public key object.
- Digital signatures generated on one vendor's HSMs can be successfully read and verified on another vendor's HSMs.
- A key encapsulated on one vendor's HSMs can be successfully read and decapsulated on another vendor's HSMs, with both HSMs generating the same shared secret key value.

Performance was not investigated during this initial effort, nor was interoperability across specific APIs such as PKCS#11. These will be investigated in future phases of interoperability testing.

9.1.1 OID Usage

One detail that deserves mention is the OID allocation used during the interoperability validation effort. Currently, new PQC algorithms such as Kyber, Dilithium, Falcon, and SPHINCS+ do not have official OIDs allocated to them. NIST will request official values once the standardization process is completed. In the meantime, temporary OIDs have been identified and we have leveraged the following OID allocation sources for the purposes of this interoperability testing exercise:

- Kyber, Dilithium, Falcon, SPHINCS+ SHA2 variants: IETF Hackathon[7]
- SPHINCS+ SHAKE variants: libOQS
- LMS, HSS, XMSS, XMSS^{MT}: C509 Signature Algorithms

The OID allocations are summarized in Table 15.

Table 15 Summary of OID allocations

	Algorithm & Variant	OID
PQC Hackathon/ libOQS	Kyber-512	1.3.6.1.4.1.22554.5.6.1
	Kyber-768	1.3.6.1.4.1.22554.5.6.2
	Kyber-1024	1.3.6.1.4.1.22554.5.6.3
	Dilithium-2	1.3.6.1.4.1.2.267.7.4.4
	Dilithium-3	1.3.6.1.4.1.2.267.7.6.5
	Dilithium-5	1.3.6.1.4.1.2.267.7.8.7
	Falcon-512	1.3.9999.3.6
	Falcon-1024	1.3.9999.3.9

[7] Note that this list appears to have been extracted from the libOQS mappings.

	Algorithm & Variant	OID
	SPHINCS+-SHA2-128fs	1.3.9999.6.4.4
	SPHINCS+-SHA2-128ss	1.3.9999.6.4.10
	SPHINCS+-SHA2-192fs	1.3.9999.6.5.3
	SPHINCS+-SHA2-192ss	1.3.9999.6.5.7
	SPHINCS+-SHA2-256fs	1.3.9999.6.6.3
	SPHINCS+-SHA2-256ss	1.3.9999.6.6.7
libOQS	SPHINCS+-SHAKE-128fs	1.3.9999.6.7.4
	SPHINCS+-SHAKE-128ss	1.3.9999.6.7.10
	SPHINCS+-SHAKE-192fs	1.3.9999.6.8.3
	SPHINCS+-SHAKE-192ss	1.3.9999.6.8.7
	SPHINCS+-SHAKE-256fs	1.3.9999.6.9.3
	SPHINCS+-SHAKE-256ss	1.3.9999.6.9.7
C509	HSS (all variants)	1.2.840.113549.1.9.16.3.17
	XMSS (all variants)	0.4.0.127.0.15.1.1.13.0
	XMSS^{MT} (all variants)	0.4.0.127.0.15.1.1.14.0

9.1.2 Algorithm Versions Tested

The specific PQC algorithm versions that were used for this exercise are summarized in Table 16, which includes hyperlinks to the relevant reference documents.

Table 16 Algorithm versions tested

Algorithm	Version Tested (w/hyperlink)
Kyber	v3.02 (August 4, 2021)
Dilithium	v3.1 (February 8, 2021)
SPHINCS+	v3.1 (June 10, 2022)
LMS/HSS	RFC 8554
XMSS/XMSS^{MT}	RFC 8391

9.2 Interoperability Test Results

This section contains the detailed test results from all of the interoperability testing that was performed as part of this exercise. The following subsections describe the vendor-declared list of basic capabilities for each of their implementations, as well as the detailed results from performing interoperability tests on key import/export, digital signature generation/verification, and key encapsulation/decapsulation.

9.2.1 Basic Capabilities

Each HSM vendor provided an outline of the PQC capabilities that they supported, which are summarized in the tables below using three generic categories: key generation (Table 17), digital signatures (Table 18), and key encapsulation (Table 19). Each category is organized by algorithm and variant, for which the vendors marked their capability using the following notation:

- ☑ = available/supported
- ☐ = supported but not tested

- ■ = not supported at this time

Table 17 Key generation capabilities by HSM vendor

Key Generation Algorithm and Parameters	Crypto4A	Entrust	Thales DIS	Thales TCT	Utimaco[8]
Kyber L1: 512	Success	Success	Success	Success	Not tested
Kyber L3: 768	Success	Success	Success	Success	Not tested
Kyber L5: 1024	Success	Success	Success	Success	Not tested
Dilithium L2	Success	Success	Success	Success	Not tested
Dilithium L3	Success	Success	Success	Success	Not tested
Dilithium L5	Success	Success	Success	Success	Not tested
Falcon L1: 512	N/A	Success	Success	Success	N/A
Falcon L5: 1025	N/A	Success	Success	Success	N/A
SPHINCS+-SHAKE-128ss	Success	Success	Success	N/A	N/A
SPHINCS+-SHAKE-128fs	Success	Success	Success	N/A	N/A
SPHINCS+-SHAKE-192ss	Success	Success	Success	N/A	N/A
SPHINCS+-SHAKE-192fs	Success	Success	Success	N/A	N/A
SPHINCS+-SHAKE-256ss	Success	Success	Success	N/A	N/A
SPHINCS+-SHAKE-256fs	Success	Success	Success	N/A	N/A
SPHINCS+-SHA2-128ss	Success	Success	Success	N/A	N/A
SPHINCS+-SHA2-128fs	Success	Success	Success	N/A	N/A
SPHINCS+-SHA2-192ss	Success	Success	Success	N/A	N/A
SPHINCS+-SHA2-192fs	Success	Success	Success	N/A	N/A
SPHINCS+-SHA2-256ss	Success	Success	Success	N/A	N/A
SPHINCS+-SHA2-256fs	Success	Success	Success	N/A	N/A
XMSS-SHA2_10_256	Success	N/A	Success	N/A	Not tested
XMSS-SHA2_16_256	Success	N/A	Success	N/A	Not tested
XMSS-SHA2_20_256	Success	N/A	Not tested	N/A	Not tested
XMSSMT-SHA2_20/2_256	Success	N/A	Success	N/A	Not tested
XMSSMT-SHA2_40/2_256	Success	N/A	Not tested	N/A	Not tested
XMSSMT-SHA2_60/3_256	Success	N/A	Not tested	N/A	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {1, 10, 8, 32}	Success	N/A	Success	Success	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {1, 20, 8, 32}	Success	N/A	Not tested	N/A	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {1, 10, 8, 24}	Success	N/A	Success	Success	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {1, 20, 8, 24}	Success	N/A	Not tested	N/A	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {1, 5, 8, 32}	Success	N/A	Success	Success	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {1, 5, 8, 24}	Success	N/A	Success	Success	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {2, {10, 8, 32}, {10, 8, 32}}	Success	N/A	Success	Success	Not tested

[8] We have documented the capabilities reported by Utimaco because the test results were not fully available at the deadline of this document. The full set of test results will be added to a future version of this document.

Key Generation Algorithm and Parameters	Crypto4A	Entrust	Thales DIS	Thales TCT	Utimaco[8]
LMS/HSS {L, h_i, w_i, n_i} = {2, {10, 8, 32}, {20, 8, 32}}	Success	N/A	Not tested	N/A	Not tested
LMS/HSS {L, h_i, w_i, n_i} = {2, {20, 8, 32}, {20, 8, 32}}	Success	N/A	Not tested	N/A	Not tested

For the digital signature capabilities, the vendors indicated their ability to generate (i.e., sign) and verify signatures separately for each algorithm and variant.

Table 18 Digital signature capabilities by HSM vendor

Digital Signature Generation/Verification Algorithm	Crypto4A	Entrust	Thales DIS	Thales TCT	Utimaco[9]
• **Dilithium**	(S/V)	(S/V)	(S/V)	(S/V)	(S/V)
○ L2	☑/☑	☑/☑	☑/☑	☑/☑	☐/☐
○ L3	☑/☑	☑/☑	☑/☑	☑/☑	☐/☐
○ L5	☑/☑	☑/☑	☑/☑	☑/☑	☐/☐
• **Falcon**					
○ L1: 512	■/■	☑/☑	☑/☑	☑/☑	■/■
○ L5: 1024	■/■	☑/☑	☑/☑	☑/☑	■/■
• **SPHINCS+**					
○ SPHINCS+-SHAKE-128ss	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHAKE-128fs	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHAKE-192ss	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHAKE-192fs	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHAKE-256ss	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHAKE-256fs	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHA2-128ss	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHA2-128fs	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHA2-192ss	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHA2-192fs	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHA2-256ss	☑/☑	☑/☑	☑/☑	■/■	■/■
○ SPHINCS+-SHA2-256fs	☑/☑	☑/☑	☑/☑	■/■	■/■
• **XMSS/XMSS^{MT}**					
○ XMSS-SHA2_10_256	☑/☑	■/■	☑/☑	■/■	☐/☐
○ XMSS-SHA2_16_256	☑/☑	■/■	☑/☑	■/■	☐/☐
○ XMSS-SHA2_20_256	☑/☑	■/■	☐/☐	■/■	☐/☐
○ XMSSMT-SHA2_20/2_256	☑/☑	■/■	☑/☑	■/■	☐/☐
○ XMSSMT-SHA2_40/2_256	☑/☑	■/■	☐/☐	■/■	☐/☐
○ XMSSMT-SHA2_60/3_256	☑/☑	■/■	☐/☐	■/■	☐/☐

[9] We have documented the capabilities reported by Utimaco because the test results were not fully available at the deadline of this document. The full set of test results will be added to a future version of this document.

Digital Signature Generation/Verification Algorithm	Crypto4A	Entrust	Thales DIS	Thales TCT	Utimaco[9]
• **LMS/HSS**					
○ $\{L, h_i, w_i, n_i\} = \{1, 10, 8, 32\}$	☑/☑	■/■	☑/☑	☑/☑	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{1, 20, 8, 32\}$	☑/☑	■/■	☐/☐	■/■	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{1, 10, 8, 24\}$	☑/☑	■/■	☑/☑	☑/☑	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{1, 20, 8, 24\}$	☑/☑	■/■	☐/☐	■/■	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{1, 5, 8, 32\}$	☑/☑	■/■	☑/☑	☑/☑	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{1, 5, 8, 24\}$	☑/☑	■/■	☑/☑	☑/☑	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{2, \{10, 8, 32\}, \{10, 8, 32\}\}$	☑/☑	■/■	☑/☑	☑/☑	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{2, \{10, 8, 32\}, \{20, 8, 32\}\}$	☑/☑	■/■	☐/☐	■/■	☐/☐
○ $\{L, h_i, w_i, n_i\} = \{2, \{20, 8, 32\}, \{20, 8, 32\}\}$	☑/☑	■/■	☐/☐	■/■	☐/☐

Similarly, for key encapsulation mechanisms, the vendors indicated their ability to perform encapsulation and decapsulation operations separately as well for each algorithm and variant.

Table 19 Key encapsulation capabilities by HSM vendor

Key Encapsulation Mechanism Algorithm	Crypto4A	Entrust	Thales DIS	Thales TCT	Utimaco[10]
• **Kyber**	(E/D	(E/D)	(E/D)	(E/D)	(E/D)
○ L1: 512	☑/☑	☑/☑	☑/☑	☑ /☑	☐/☐
○ L3: 768	☑/☑	☑/☑	☑/☑	☑ /☑	☐/☐
○ L5: 1024	☑/☑	☑/☑	☑/☑	☑ /☑	☐/☐

9.2.2 PQC Key Generation, Export, and Import

The first set of interoperability tests involved having each HSM vendor generate a variety of public key objects which they then exported in a PEM-based format. These public keys were then imported into other vendors' HSMs to see if they would result in valid public key objects that could be used for digital signature verification and key encapsulation.

The results of these tests are summarized in Table 20 where each row summarizes whether or not the given vendor's HSM was able to generate and export the given algorithm's public key, and if the other HSM vendors were able to import the key successfully, using the notation:

- ☑ = successfully imported the public key object

[10] We have documented the capabilities reported by Utimaco because the test results were not fully available at the deadline of this document. The full set of test results will be added to a future version of this document.

- ☒ = unable to import the public key object
- ☐ = supported but not tested
- ■ = not supported at this time

Table 20 Test results for HSM key generation, export, and import

Export	Import: Crypto4A	Import: Entrust	Import: Thales DIS	Import: Thales TCT	Import: Utimaco[11]
Kyber-512 (L1)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Kyber-768 (L3)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Kyber-1024 (L5)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Dilithium-2 (L2)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Dilithium-3 (L3)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Dilithium-5 (L5)					
Crypto4A	☑	☑	☑	☑	☐

[11] We have documented the capabilities reported by Utimaco because the test results were not fully available at the deadline of this document. The full set of test results will be added to a future version of this document.

Export	**Import: Crypto4A**	**Import: Entrust**	**Import: Thales DIS**	**Import: Thales TCT**	**Import: Utimaco**[11]
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Falcon-512 (L1)					
Crypto4A	■	■	■	■	■
Entrust	■	☑	☑	☑	■
Thales DIS	■	☑	☑	☑	■
Thales TCT	■	☑	☑	☑	■
Utimaco	■	■	■	■	■
Falcon-1024 (L5)					
Crypto4A	■	■	■	■	■
Entrust	■	☑	☑	☑	■
Thales DIS	■	☑	☑	☑	■
Thales TCT	■	☑	☑	☑	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-128ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-128fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-192ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-192fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-256ss					

Export	Import: Crypto4A	Import: Entrust	Import: Thales DIS	Import: Thales TCT	Import: Utimaco[11]
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-256fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-128ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-128fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-192ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-192fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-256ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■

Export	**Import: Crypto4A**	**Import: Entrust**	**Import: Thales DIS**	**Import: Thales TCT**	**Import: Utimaco**[11]
SPHINCS+-SHA2-256fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
XMSS-SHA2_10_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSS-SHA2_16_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSS-SHA2_20_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSSMT-SHA2_20/2_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSSMT-SHA2_40/2_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSSMT-SHA2_60/3_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■

Export	Import: Crypto4A	Import: Entrust	Import: Thales DIS	Import: Thales TCT	Import: Utimaco[11]
Utimaco	☐	■	☐	■	☐
LMS/HSS {L, h_i, w_i, n_i} = {1, 10, 8, 32}					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS {L, h_i, w_i, n_i} = {1, 20, 8, 32}					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS {L, h_i, w_i, n_i} = {1, 10, 8, 24}					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	☐
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS {L, h_i, w_i, n_i} = {1, 20, 8, 24}					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS {L, h_i, w_i, n_i} = {1, 5, 8, 32}					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS {L, h_i, w_i, n_i} = {1, 5, 8, 24}					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS {L, h_i, w_i, n_i} = {2, {10, 8, 24}, {10, 8, 24}}					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐

Export	Import: Crypto4A	Import: Entrust	Import: Thales DIS	Import: Thales TCT	Import: Utimaco[11]
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{2, \{10, 8, 24\}, \{20, 8, 24\}\}$					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{2, \{20, 8, 24\}, \{20, 8, 24\}\}$					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐

9.2.3 PQC Signature Generation and Verification

The second set of interoperability tests performed involved having an HSM vendor perform a digital signature and export the public key component of the signing key (a.k.a., the verification key). The other HSM vendors then attempted to import the verification key and verify the generated signature using the same message that was signed.

The results of these tests are summarized in Table 21 where each row summarizes whether or not the given vendor's HSM was able to generate a digital signature for the corresponding algorithm, and if the other HSM vendors were able to import the verification key and verify the generated signature successfully, using the notation:

- ☑ = successfully imported the verifying key and verified the digital signature
- ☒ = did NOT successfully import the key and verify the digital signature
- ☐ = supported but not tested
- ■ = not supported at this time

Table 21 Test results for HSM signature generation and verification

Signer	Verifier: Crypto4A	Verifier: Entrust	Verifier: Thales DIS	Verifier: Thales TCT	Verifier: Utimaco[12]
Dilithium-2 (L2)					
Crypto4A	☑	☑	☑	☑	☐

[12] We have documented the capabilities reported by Utimaco because the test results were not fully available at the deadline of this document. The full set of test results will be added to a future version of this document.

Signer	Verifier: Crypto4A	Verifier: Entrust	Verifier: Thales DIS	Verifier: Thales TCT	Verifier: Utimaco[12]
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Dilithium-3 (L3)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Dilithium-5 (L5)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Falcon-512 (L1)					
Crypto4A	■	■	■	■	■
Entrust	■	☑	☑	☑	■
Thales DIS	■	☑	☑	☑	■
Thales TCT	■	☑	☑	☑	■
Utimaco	■	■	■	■	■
Falcon-1024 (L5)					
Crypto4A	■	■	■	■	■
Entrust	■	☑	☑	☑	■
Thales DIS	■	☑	☑	☑	■
Thales TCT	■	☑	☑	☑	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-128ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-128fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-192ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■

Signer	Verifier: Crypto4A	Verifier: Entrust	Verifier: Thales DIS	Verifier: Thales TCT	Verifier: Utimaco[12]
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-192fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-256ss					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHAKE-256fs					
Crypto4A	☑	☑	☑	■	■
Entrust	☑	☑	☑	■	■
Thales DIS	☑	☑	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-128ss					
Crypto4A	☑	☒	☑	■	■
Entrust	☒	☑	☒	■	■
Thales DIS	☑	☒	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-128fs					
Crypto4A	☑	☒	☑	■	■
Entrust	☒	☑	☒	■	■
Thales DIS	☑	☒	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-192ss					
Crypto4A	☑	☒	☑	■	■
Entrust	☒	☑	☒	■	■
Thales DIS	☑	☒	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-192fs					
Crypto4A	☑	☒	☑	■	■
Entrust	☒	☑	☒	■	■
Thales DIS	☑	☒	☑	■	■

Signer	Verifier: Crypto4A	Verifier: Entrust	Verifier: Thales DIS	Verifier: Thales TCT	Verifier: Utimaco[12]
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-256ss					
Crypto4A	☑	☒	☑	■	■
Entrust	☒	☑	☒	■	■
Thales DIS	☑	☒	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
SPHINCS+-SHA2-256fs					
Crypto4A	☑	☒	☑	■	■
Entrust	☒	☑	☒	■	■
Thales DIS	☑	☒	☑	■	■
Thales TCT	■	■	■	■	■
Utimaco	■	■	■	■	■
XMSS-SHA2_10_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSS-SHA2_16_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSS-SHA2_20_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSSMT-SHA2_20/2_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
XMSSMT-SHA2_40/2_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■

Signer	Verifier: Crypto4A	Verifier: Entrust	Verifier: Thales DIS	Verifier: Thales TCT	Verifier: Utimaco[12]
Utimaco	☐	■	☐	■	☐
XMSSMT-SHA2_60/3_256					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{1, 10, 8, 32\}$					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{1, 20, 8, 32\}$					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{1, 10, 8, 24\}$					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{1, 20, 8, 24\}$					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{1, 5, 8, 32\}$					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{1, 5, 8, 24\}$					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐

Signer	**Verifier: Crypto4A**	**Verifier: Entrust**	**Verifier: Thales DIS**	**Verifier: Thales TCT**	**Verifier: Utimaco**[12]
LMS/HSS $\{L, h_i, w_i, n_i\} = \{2, \{10, 8, 24\}, \{10, 8, 24\}\}$					
Crypto4A	☑	■	☑	☑	☐
Entrust	■	■	■	■	■
Thales DIS	☑	■	☑	☑	☐
Thales TCT	☑	■	☑	☑	☐
Utimaco	☐	■	☐	☐	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{2, \{10, 8, 24\}, \{20, 8, 24\}\}$					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐
LMS/HSS $\{L, h_i, w_i, n_i\} = \{2, \{20, 8, 24\}, \{20, 8, 24\}\}$					
Crypto4A	☑	■	☑	■	☐
Entrust	■	■	■	■	■
Thales DIS	☐	■	☐	■	☐
Thales TCT	■	■	■	■	■
Utimaco	☐	■	☐	■	☐

9.2.4 PQC Key Encapsulation and Decapsulation

The last set of interoperability tests performed involved having an HSM vendor (a.k.a., HSM_A) perform a key encapsulation operation by importing another HSM vendor's (a.k.a., HSM_B) public encapsulation key to generate the required ciphertext and shared secret value. HSM_B then performs a key decapsulation on the ciphertext generated by HSM_A, and verifies that the generated shared secret matches the one produced by HSM_A during the encapsulation operation.

The results of these tests are summarized in Table 22 where each row summarizes whether or not the given vendor's HSM was able to successfully perform the encapsulation and decapsulation operations described in the previous paragraph, using the notation:

- ☑ = successfully imported encapsulation ciphertext and generated valid shared secret
- ☒ = did NOT successfully generate the correct shared secret
- ☐ = supported but not tested
- ■ = not supported at this time

Table 22 Test results for HSM key encapsulation and decapsulation

Encapsulate	Decapsulate: Crypto4A	Decapsulate: Entrust	Decapsulate: Thales DIS	Decapsulate: Thales TCT	Decapsulate: Utimaco[13]
Kyber-512 (L1)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Kyber-768 (L3)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐
Kyber-1024 (L5)					
Crypto4A	☑	☑	☑	☑	☐
Entrust	☑	☑	☑	☑	☐
Thales DIS	☑	☑	☑	☑	☐
Thales TCT	☑	☑	☑	☑	☐
Utimaco	☐	☐	☐	☐	☐

9.3 Summary of Results

No single vendor at this time has a complete offering of PQC algorithm support that has been validated. However, a high degree of interoperability was achieved for the capabilities that are currently supported across the whole suite of HSM vendors who participated in this exercise.

At this point in time, the only incompatibility that was found was with Entrust's SPHINCS+ SHA2-based variants, which couldn't be verified by either Crypto4A or Thales DIS (and vice versa).

The high degree of interoperability is a good indicator of the level of effort that HSM vendors have put into providing properly functioning PQC capabilities in their next generation of products. Having a high degree of interoperability between HSM vendors is an essential element of minimizing the difficulty of migrating our existing quantum-vulnerable cryptographic capabilities to quantum-safe variants, though there is still an enormous amount of work to be done.

[13] We have documented the capabilities reported by Utimaco because the test results were not fully available at the deadline of this document. The full set of test results will be added to a future version of this document.

10 Overall Status and Themes

Migration to post-quantum cryptography is a complex effort. Fortunately, the activities performed by the NCCoE project participants will help the ecosystem prepare for quantum readiness.

Thanks to early prototyping by academia and industry, we observed only a few challenges with the collaborator's core TLS and SSH protocol implementations, giving confidence that migrating to the new PQC standards will be straightforward for most implementations. The implementations used draft versions of the protocols that have been proposed by experimenters versus the standard bodies themselves (e.g., the IETF) who have been waiting for the final FIPS documents. These drafts are likely to serve as a basis for the respective working groups to define PQC integration, and the interoperability experiments in this publication can serve as supporting material to accelerate their adoption.

Few implementations for derived protocols (DTLS, MQTT, QUIC, etc.) were available; these will be tested in later phases of the project as more partners add support to their components.

The performance testing conducted demonstrated that the cost of Kyber, the recommended algorithm for key exchange, is competitive when compared with the current elliptic curve state of the art, and even their hybrid combination would be practical for most use cases. This is encouraging for organizations planning to transition sooner than later, wishing to add quantum resistance on top of existing standards.

Our workstream is planning many more activities, including testing more algorithms and parameter sets and more protocols, onboarding more partner implementations in both hardware and software, and demonstrating more scenarios. In tandem, we will start to release a more detailed technical view of our testbed so that interested parties can replicate our test procedures.

Appendix A List of Acronyms

AES-GCM	Advanced Encryption Standard with Galois/Counter Mode
AI	Artificial Intelligence
API	Application Programming Interface
ASC	Accredited Standards Committee
ASN.1	Abstract Syntax Notation One
AWS	Amazon Web Services
C-QSA	CryptoNext Quantum Safe Application Plugins
C-QSC	CryptoNext Quantum Safe Crypto Services
C-QSL	CryptoNext Quantum Safe Library
C-QSR	CryptoNext Quantum Safe Remediation
C-QST	CryptoNext Quantum Safe Tools
CA	Certificate Authority
CFRG	(IRTF) Crypto Forum Research Group
CISA	Cybersecurity & Infrastructure Security Agency
CMS	Cryptographic Message Syntax
CNG	Cryptography API Next Generation
CNSA	Commercial National Security Algorithm Suite
CPU	Central Processing Unit
CRADA	Cooperative Research and Development Agreement
CRQC	Cryptanalytically Relevant Quantum Computer
DCD	Delta Certificate Descriptor
DER	Distinguished Encoding Rules
DIS	(Thales) Digital Identity and Security
DTLS	Datagram Transport Layer Security
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie Hellman
ECDHE	Elliptic Curve Diffie-Hellman Exchange
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-Curve Digital Signature Algorithm
ETSI	European Telecommunications Standards Institute
FIPS	Federal Information Processing Standard
FM	(Thales) Functionality Module
GB	Gigabyte
GHz	Gigahertz
GSMA	Groupe Speciale Mobile Association

HSE	High Speed Encryptor
HSM	Hardware Security Module
HSP	Hardware Security Platform
HSS	Hierarchical Signature System
IETF	Internet Engineering Task Force
IKEv2	Internet Key Exchange Version 2
IoT	Internet of Things
IPsec	Internet Protocol Security
IRTF	Internet Research Task Force
ISA	Instruction Set Architecture
ISC	Information Security Corporation
JCE	Java Cryptography Extension
JSSE	Java Secure Socket Extension
KB	Kilobyte
KEM	Key Encapsulation Mechanism
KMIP	Key Management Interoperability Protocol
LAMPS	(IETF) Limited Additional Mechanisms for PKIX and SMIME
LMS	Leighton-Micali Signature
MB	Megabyte
MQTT	Message Queuing Telemetry Transport
NCCoE	National Cybersecurity Center of Excellence
NIAP	National Information Assurance Partnership
NSA	National Security Agency
OASIS	Organization for the Advancement of Structured Information Standards
OCSP	Online Certificate Status Protocol
OID	Object Identifier
OMB	Office of Management and Budget
OQS	(Microsoft) Open Quantum Safe
PKCS	Public-Key Cryptography Standard
PKI	Public Key Infrastructure
PQC	Post-Quantum Cryptography
PQSDK	PQShield Software Development Kit
R&D	Research and Development
RAM	Random Access Memory
REST	Representational State Transfer
RFC	Request for Comments

RTT	Round-Trip Time
S/MIME	Secure/Multipurpose Internet Mail Extensions
SCT	Signed Certificate Timestamp
SDK	Software Development Kit
SFTP	Secure File Transfer Protocol
SHA2	Secure Hash Algorithm 2
SHAKE	Secure Hash Algorithm and KECCAK
SIA	Subject Info Access
SP	Special Publication
SSH	Secure Shell
TCP	Transmission Control Protocol
TCT	(Thales) Trusted Cyber Technologies
TLS	Transport Layer Security
TPM	Trusted Platform Module
TSA	Time Stamp Authority
UDP	User Datagram Protocol
WG	Working Group
XMSS	eXtended Merkle Signature Scheme
$XMSS^{MT}$	Multi-Tree eXtended Merkle Signature Scheme

Appendix B References

[1] "National Cybersecurity Center of Excellence (NCCoE) Migration to Post-Quantum Cryptography; Notice," 86 *Federal Register* 56898 (October 13, 2021), pp. 56898-56900. https://www.federalregister.gov/d/2021-22223

[2] Cybersecurity & Infrastructure Security Agency, National Security Agency, and National Institute of Standards and Technology (2023) Quantum-Readiness: Migration to Post-Quantum Cryptography. (CISA, Arlington, Virginia), August 21, 2023. Available at https://www.cisa.gov/resources-tools/resources/quantum-readiness-migration-post-quantum-cryptography

[3] Office of Management and Budget (2022) Migrating to Post-Quantum Cryptography. (The White House, Washington, DC), OMB Memorandum M-23-02, November 18, 2022. Available at https://www.whitehouse.gov/wp-content/uploads/2022/11/M-23-02-M-Memo-on-Migrating-to-Post-Quantum-Cryptography.pdf

[4] Cooper DA, Apon D, Dang QH, Davidson MS, Dworkin MJ, Miller CA (2020) Recommendation for Stateful Hash-Based Signature Schemes. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-208. https://doi.org/10.6028/NIST.SP.800-208

[5] Crockett E, Paquin C, Stebila D (2019) Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. Available at https://eprint.iacr.org/2019/858

[6] Weibel A (2020) Round 2 Hybrid Post-Quantum TLS Benchmarks. Available at https://aws.amazon.com/blogs/security/round-2-hybrid-post-quantum-tls-benchmarks/

[7] Salz R, Aviram N (2023) TLS 1.2 is in Feature Freeze. (Internet Engineering Task Force (IETF)), Internet-Draft draft-rsalz-tls-tls12-frozen. Available at https://datatracker.ietf.org/doc/draft-rsalz-tls-tls12-frozen/

[8] Bae S, Chang Y, Park H, Kim M, Shin Y (2022) A Performance Evaluation of IPsec with Post-Quantum Cryptography. *Information Security and Cryptology - ICISC 2022* (Springer, Seoul, South Korea), pp. 249-266. https://doi.org/10.1007/978-3-031-29371-9

[9] Gazdag SL, Grundner-Culemann S, Heider T, Herzinger D, Schartl F, Cho JY, Guggemos T, Loebenberger D (2023) Quantum-Resistant MACsec and IPsec for Virtual Private Networks. *International Conference on Research in Security Standardization* (Springer), pp. 1-21. https://doi.org/10.1007/978-3-031-30731-7_1

[10] Kampanakis P, Stebila D, Hansen T (2023) Post-quantum Hybrid Key Exchange in SSH. (Internet Engineering Task Force (IETF)), Internet-Draft draft-kampanakis-curdle-ssh-pq-ke-01. Available at https://datatracker.ietf.org/doc/draft-kampanakis-curdle-ssh-pq-ke/01/

[11] Sikeridis D, Kampanakis P, Devetsikiotis M (2020) Assessing the overhead of post-quantum cryptography in TLS 1.3 and SSH. *CoNEXT '20: Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pp. 145-156. https://doi.org/10.1145/3386367.3431305

[12] Bos JW, Costello C, Naehrig M, Stebila D (2014) Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. *Proc. IEEE Symposium on Security and Privacy (S&P) 2015*, pp. 553-570. https://eprint.iacr.org/2014/599

[13] Stebila D, Fluhrer S, Gueron S (2022) Hybrid key exchange in TLS 1.3. (Internet Engineering Task Force (IETF)), Internet-Draft draft-ietf-tls-hybrid-design-05. Available at https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-05

[14] Kwiatkowski K, Kampanakis P (2023) Post-quantum hybrid ECDHE-Kyber Key Agreement for TLSv1.3. (Internet Engineering Task Force (IETF)), Internet-Draft draft-kwiatkowski-tls-ecdhe-kyber. Available at https://datatracker.ietf.org/doc/html/draft-kwiatkowski-tls-ecdhe-kyber

[15] Westerbaan BE, Stebila D (2023) X25519Kyber768Draft00 hybrid post-quantum key agreement. (Internet Engineering Task Force (IETF)), Internet-Draft draft-tls-westerbaan-xyber768d00. Available at https://datatracker.ietf.org/doc/html/draft-tls-westerbaan-xyber768d00

[16] Paquin C, Stebila D, Tamvada G (2020) Benchmarking Post-Quantum Cryptography in TLS. Available at https://eprint.iacr.org/2019/1447.pdf

[17] Kwiatkowski K, Valenta L (2019) The TLS Post-Quantum Experiment. Available at https://blog.cloudflare.com/the-tls-post-quantum-experiment/

[18] Sikeridis D, Kampanakis P, Devetsikiotis M (2020) Post-Quantum Authentication in TLS 1.3: A Performance Study. *Network and Distributed Systems Security (NDSS) Symposium 2020*. Available at https://www.ndss-symposium.org/wp-content/uploads/2020/02/24203-paper.pdf

[19] Iyengar J, Swett I (2021) QUIC Loss Detection and Congestion Control. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 9002. https://doi.org/10.17487/RFC9002

[20] Kampanakis P, Bytheway C, Westerbaan BE, Thomson M (2023) Suppressing CA Certificates in TLS 1.3. (Internet Engineering Task Force (IETF)), Internet-Draft draft-kampanakis-tls-scas-latest-03. Available at https://datatracker.ietf.org/doc/html/draft-kampanakis-tls-scas-latest-03

[21] Jackson D (2023) Abridged Compression for WebPKI Certificates. (Internet Engineering Task Force (IETF)), Internet-Draft draft-jackson-tls-cert-abridge-00. Available at https://datatracker.ietf.org/doc/html/draft-jackson-tls-cert-abridge-00

[22] Kampanakis P, Lepoint T (2023) Vision Paper: Do We Need to Change Some Things? *SSR 2023: International Conference on Research in Security Standardization* (Springer), pp. 78-102. https://doi.org/10.1007/978-3-031-30731-7_4

[23] Cooper D, Santesson S, Farrell S, Boeyen S, Housley R, Polk W (2008) Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 5280. https://doi.org/10.17487/RFC5280

[24] Massimo J, Kampanakis P, Turner S, Westerbaan BE (2023) Internet X.509 Public Key Infrastructure: Algorithm Identifiers for Dilithium. (Internet Engineering Task Force (IETF)), Internet-Draft draft-ietf-lamps-dilithium-certificates. Available at https://datatracker.ietf.org/doc/draft-ietf-lamps-dilithium-certificates/

[25] Turner S, Kampanakis P, Massimo J, Westerbaan BE (2023) Internet X.509 Public Key Infrastructure: Algorithm Identifiers for Kyber. (Internet Engineering Task Force (IETF)), Internet-Draft draft-ietf-lamps-kyber-certificates. Available at https://datatracker.ietf.org/doc/draft-ietf-lamps-kyber-certificates/

[26] Becker A, Guthrie R, Jenkins M (2023) Related Certificates for Use in Multiple Authentications within a Protocol. (Internet Engineering Task Force (IETF)), Internet-Draft draft-becker-guthrie-cert-binding-for-multi-auth. Available at https://datatracker.ietf.org/doc/draft-becker-guthrie-cert-binding-for-multi-auth/

[27] Ounsworth M, Gray J, Pala M, Klaubner J (2023) Composite Signatures For Use in Internet PKI. (Internet Engineering Task Force (IETF)), Internet-Draft draft-ounsworth-pq-composite-sigs. Available at https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-sigs/

[28] Ounsworth M, Gray J (2023) Composite KEM For Use in Internet PKI. (Internet Engineering Task Force (IETF)), Internet-Draft draft-ietf-lamps-pq-composite-kem. Available at https://datatracker.ietf.org/doc/draft-ietf-lamps-pq-composite-kem

[29] Truskovsky A, Van Geest D, Fluhrer S, Kampanakis P, Ounsworth M, Mister S (2023) Multiple Public-Key Algorithm X.509 Certificates. (Internet Engineering Task Force (IETF)), Internet-Draft draft-truskovsky-lamps-pq-hybrid-x509. Available at https://datatracker.ietf.org/doc/draft-truskovsky-lamps-pq-hybrid-x509/

[30] International Telecommunication Union Telecommunication Standardization Sector (ITU-T) (2019) *ITU-T X.509 – Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks* (ITU-T, Geneva, Switzerland). Available at https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=14033&lang=en

[31] Bonnell C, Gray J, Hook D, Okubo T, Ounsworth M (2023) A Mechanism for Encoding Differences in Paired Certificates. (Internet Engineering Task Force (IETF)), Internet-Draft draft-bonnell-lamps-chameleon-certs. Available at https://datatracker.ietf.org/doc/draft-bonnell-lamps-chameleon-certs/

[32] IETF PQC Hackathon Interoperability Results. Available at https://ietf-hackathon.github.io/pqc-certificates/pqc_hackathon_results_certs_r3.html

[33] Kampanakis P, Panburana P, Curcio M, Shroff C, Alam MM (2021) Post-Quantum LMS and SPHINCS+ Hash-Based Signatures for UEFI Secure Boot. Available at https://eprint.iacr.org/2021/041

[34] McGrew D, Curcio M, Fluhrer S (2019) Leighton-Micali Hash-Based Signatures. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8554. https://doi.org/10.17487/RFC8554

[35] Huelsing A, Butin D, Gazdag S, Rijneveld J, Mohaisen A (2018) XMSS: eXtended Merkle Signature Scheme. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8391. https://doi.org/10.17487/RFC8391

[36] Kakvi S (2020) SoK: Comparison of the Security of Real World RSA Hash-and-Sign Signatures. *International Conference on Research in Security Standardization* (Springer), pp. 91-113. https://doi.org/10.1007/978-3-030-64357-7_5

[37] Josefsson S, Liusvaara I (2017) Edwards-Curve Digital Signature Algorithm (EdDSA). (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8032. https://doi.org/10.17487/RFC8032

[38] Bernstein DJ, Josefsson S, Lange T, Schwabe P, Yang B-Y (2015) EdDSA for more curves. Available at http://ed25519.cr.yp.to/eddsa-20150704.pdf

[39] Mattsson J, Migault D (2018) ECDHE_PSK with AES-GCM and AES-CCM Cipher Suites for TLS 1.2 and DTLS 1.2. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8442. https://doi.org/10.17487/RFC8442

[40] Rescorla E (2018) The Transport Layer Security (TLS) Protocol Version 1.3. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8446. https://doi.org/10.17487/RFC8446

[41] Josefsson S, Schaad J (2018) Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8410. https://doi.org/10.17487/RFC8410

[42] Nir Y (2018) Using the Edwards-Curve Digital Signature Algorithm (EdDSA) in the Internet Key Exchange Protocol Version 2 (IKEv2). (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8420. https://doi.org/10.17487/RFC8420

[43] Harris B, Velvindron L (2020) Ed25519 and Ed448 Public Key Algorithms for the Secure Shell (SSH) Protocol. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8709. https://doi.org/10.17487/RFC8709

[44] Liusvaara I (2017) CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE). (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8037. https://doi.org/10.17487/RFC8037

[45] Housley R (2018) Use of Edwards-Curve Digital Signature Algorithm (EdDSA) Signatures in the Cryptographic Message Syntax (CMS). (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 8419. https://doi.org/10.17487/RFC8419

[46] Myers M, Ankney R, Malpani A, Galperin S, Adams C (1999) X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP. (Internet Engineering Task Force (IETF)), IETF Request for Comments (RFC) 2560. https://doi.org/10.17487/RFC2560

[47] Zimman C, Bong D (2020) PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0 (OASIS Open, Boston, Massachusetts). Available at https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/os/pkcs11-curr-v3.0-os.html

[48] Xiao J, Ito T (2020) Performance Comparisons and Migration Analyses of Lattice-based Cryptosystems on Hardware Security Module. Available at https://eprint.iacr.org/2020/990.pdf

Appendix C Hash and Sign Analysis

NIST's Post-Quantum Cryptography Project is in the process of standardizing new, quantum-safe signatures. These signatures operate on arbitrary size messages, which is different than traditional uses of classical signatures which were digesting the message and signing the digest. As we proceed with standardizing and adopting the use of quantum-safe signatures, we ought to evaluate which choice is more suitable for various common use-cases and what implications it would have. This appendix analyzes the options and summarizes public discussions on the topic in various fora. The goal is for engineers or standards bodies that will use these signatures to make informed decisions between using the quantum-safe signatures as they are or choosing an option which digests the message before signing it.

This analysis finds that most use-cases can leverage post-quantum signatures as they are without pre-digesting the message. This approach offers better collision resistance than digesting before signing. Some contexts could still benefit from pre-digesting, particularly cases that cannot tolerate holding an entire large message in memory or where digesting can speed up performance. Pre-digesting could still remain possible for uses which can offer a protocol-level signature envelope.

C.1 Introduction of the Digest-then-Sign Dilemma

Asymmetric cryptographic primitives have generally been limited to fixed-size inputs, typically a few hundred bytes, that are mapped to a particular mathematical object. To construct signature schemes with arbitrarily sized input, the natural approach is to first hash the message, then sign the resulting digest. Traditional signature schemes like RSA/PKCS#1 (RFC 8017) and ECDSA (FIPS 186-5) are constructed in such a way that it is easy to separate the digesting step from the asymmetric primitive. This approach is commonly called digest-then-sign or hash-then-sign.

Newer signature algorithms use a slightly different method by injecting a random nonce or data from the public key into the digest. This may bring additional security properties, such as improved resistance against hash collisions or exclusive ownership and message bindings. In this approach, it is no longer possible to separate the digesting step from the public/private key operation. EdDSA (RFC 8032) is a notable example of a signature scheme using this method.

In all three quantum-safe signatures picked in Round 3 of NIST's Post-Quantum Project, the digest step is intrinsically linked to the asymmetric primitive. Before generating the signature, the input message is hashed with additional algorithm-specific data. More precisely,

- Dilithium takes the whole message M as input and hashes it with tr, a hash of the public key {ρ, t}, to create a digest μ=SHAKE256(tr||M). It then proceeds to sign that value. Dilithium requires collision resistance for the digest function, but collisions are specific to a given public key.
- Falcon calculates the HashToPoint(r||M, q, n) of message M where r is a random value and q, n are Falcon parameters. HashToPoint hashes r||M to a point in the lattice which is then used to generate the signature. This randomized hashing does not require collision resistance for the hash function.
- SPHINCS+ calculates a proper output size digest of the message M by using (R||PK.seed||PK.root||M) as inputs to a variable/extendable output function (e.g., SHAKE256,

MGF1-SHA256). R is a random value generated from a secret random value and the message. PK.seed and PK.root are public values for the signer. This randomized hashing does not require collision resistance for the digest function. The calculated digest is then used to generate the SPHINCS+ signature. Note that SPHINCS+ does two passes on the message, one to generate R and a second to digest the message. Two passes could affect performance for large messages.

Thus, the digest step and the public/private key operation cannot be separated in the case of these quantum-safe signatures. This change causes difficulties when migrating solutions based on RSA or ECDSA to PQC. For example, a paper from 2021 on hash-based signatures for secure boot [33] came across an OpenSSL API incompatibility between classical ECDSA signatures, which were assuming a digest, and SPHINCS+, which includes the message digest in the signature parameter itself.

Note that the analysis below applies to stateful hash-based signatures [4] (RFC 8554 [34], RFC 8391 [35]), as they also digest the message internally by using a pseudorandom value in order to generate the one-time signature which is included in the message signature.

C.1.1 Terminology

To avoid any confusion, this rest of this appendix will use the following terminology:

- "message" denotes the raw data to be signed (contents of a file, attributes in a certificate, etc.).
- "Internal Digest" denotes the hashing done as part of the post-quantum signature algorithm, for example the step μ=SHAKE256(tr||M) in Dilithium, as explained above.
- "Signed Data" denotes the actual input to the signature algorithm. The Signed Data may or may not be the same as the Message. For example, in a hash-then-sign scenario, the Signed Data would be the hash of the Message.

Notice that the Internal Digest is always performed on the Signed Data, so in a hash-then-sign scenario there would be two consecutive hashes: first a plain hash (e.g., SHAKE256) of the Message to generate the Signed Data, then the Internal Digest (randomized, as above) on the Signed Data during the signature.

C.2 Performance for PQC Signatures

First, we analyze performance of the quantum-safe signatures and compare it against digesting the messages before signing. Table 23 was generated using [liboqs 0.8.0-rc1]() and OpenSSL 1.1.1 (see Appendix D.4 for additional details). We used absolute times instead of CPU cycles in our measurements, as these are only provided for comparison in this context.

Table 23 Mean time (µs) of post-quantum signature sign and verify for plaintext sizes of 1K, 10K, 100K, 1MB, 100MB on Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz

	Time (µs) for 1KB message				
	1KB	**10KB**	**100KB**	**1MB**	**100MB**
SHA256	2.64	25.07	253.9	2496	250000
SHA512	1.84	17.00	166.7	1669	167778
SHAKE256	3.52	32.88	304.4	3046	306000

Dilithium-3 sign	267.5	299.55	615.1	3421	306767
Dilithium-3 verify	123.9	156.10	470.4	3308	305914
Dilithium-5 sign	359.6	393.2	707.0	3498	306448
Dilithium-5 verify	203.6	236.7	549.2	3361	306240
Falcon-512 sign	2373	2399	2671	5428	313660
Falcon-512 verify	1972	1997	2273	5013	307683
Falcon-1024 sign	6711	6749	7022	9768	312870
Falcon-1024 verify	6080	6045	6336	9071	312147
SPHINCS+-SHA2-192f-simple sign	15119	15149	15435	18455	351220
SPHINCS+-SHA2-192f-simple verify	1060	1079	1228	2725	169114
SPHINCS+-SHAKE-192s-simple sign	590634	590486	591140	602835	1202268
SPHINCS+-SHAKE-192s-simple verify	671	692	988	3721	306412

Table 23 shows that Dilithium and Falcon signing and verification performance is affected by the message size increases. The flavor of randomized hashing (i.e., SHAKE256, HashToPoint) these algorithms use ends up showing up as the message exceeds 1 MB. In absolute numbers, even for 100 MB plaintexts, both Falcon and Dilithium performance stayed below 400 ms in our platform, which is acceptable.

The two SPHINCS+ parameters we tested were at NIST's Level 3. One used SHAKE256 as the hash and was optimized for size. The other parameter was using SHA512 and was optimized for performance. We notice that with SPHINCS+-SHAKE-192s-simple, signing is barely affected by smaller message sizes. That is because signing is dominated by the FORS and WOTS+ hash calculations and not by the two SHAKE256 internal hashes (i.e., H_{msg}, PRF_{msg}), especially for small message sizes (1, 10, 100, and 1000 KB). At 1MB, the internal hash's cost increases and affects signing. Since verification is faster, the cost of the internal SHAKE256 digest becomes noticeable for messages of 1 MB and 100 MB. The observations are essentially the same for the SPHINCS+-SHA2-192f-simple sign parameter set. Signing and verification show noticeable slowdowns at 100 MB message sizes when SHA512 is used in the Hmsg and PRFmsg calculation of the message and is significant compared to the rest of the FORS and WOTS+ hashing in SPHINCS+. In absolute performance numbers, signing stayed within the same magnitude, so if the signer could afford SPHINCS+ signing performance, it could afford signing bigger messages. Similarly, SPHINCS+ verification performance stayed within acceptable levels even for big messages.

When evaluating SHA-256, SHA512, and SHAKE256 performance in a hash-then-sign scenario, we can see that it is highly efficient even for 100 MB. If someone was following the digest-then-sign paradigm with the post-quantum signatures, they would get better overall performance when using more efficient SHA256 or SHA512 than a slower Internal Digest function like SHAKE256. Using SHAKE256 pre-digests, the improvement will be insignificant. For example, if the function used to digest the message was SHAKE256, which is also used internally in the Dilithium μ calculation, then digesting before signing would not have a significant performance impact. Note that in absolute numbers, even for 100MB plaintexts, Falcon and Dilithium performance stays below 400 ms in our platform, which is acceptable. Using digest-then-sign in SPHINCS+ would improve signing and verification performance when SHA256 or SHA512 is used to pre-hash the message, especially for the SPHINCS+-SHAKE parameter sets, but overall the improvement will be noticeable only for large messages.

C.3 The EdDSA Precedent

Although hash-then-sign has been the status quo, the message digest was traditionally specified in the use-case itself (e.g., X.509, CMS, TLS), not internally in the signature. RSA and ECDSA signature specifications themselves have been used with a hash function which digested the message. The message digest was subsequently signed by the RSA or ECDSA primitive. Digesting was decoupled from signing with the private key operation. The paper SoK: Comparison of the Security of Real World RSA Hash-and-Sign Signatures [36] lays out all the standardized hash-then-sign RSA signatures, which mostly were what all RSA signature variants used.

This paradigm changed with EdDSA. The EdDSA signature was relatively recently standardized in RFC 8032 [37]. Initially it was specified taking the whole message as input, but later was ratified with two versions, Pure and Prehash. The former takes the whole message as input and passes through it twice in order to sign. The latter takes the digest of the message as input. To explain the rationale, RFC 8032 states:

> Choosing which variant to use depends on which property is deemed to be more important between 1) collision resilience and 2) a single-pass interface for creating signatures. The collision resilience property means EdDSA is secure even if it is feasible to compute collisions for the hash function. The single-pass interface property means that only one pass over the input message is required to create a signature. PureEdDSA requires two passes over the input. Many existing APIs, protocols, and environments assume digital signature algorithms only need one pass over the input and may have API or bandwidth concerns supporting anything else.
>
> The Ed25519ph and Ed448ph variants are prehashed. This is mainly useful for interoperation with legacy APIs, since in most of the cases, either the amount of data signed is not large or the protocol is in the position to do digesting in ways better than just prehashing (e.g., tree hashing or splitting the data). The prehashing also makes the functions greatly more vulnerable to weaknesses in hash functions used. These variants SHOULD NOT be used.

Additionally, the EdDSA paper [38] explains that pre-digesting the messages with PrehashEdDSA introduces collision concerns by saying:

> PureEdDSA is resilient to collisions in the underlying hash function H. HashEdDSA is not resilient to collisions in H0: if the attacker finds messages M1 and M2 with H0(M1)=H0(M2), and convinces the legitimate H0-EdDSA signer to sign M1, then the attacker can forge the same signature as a signature of M2. Modern hash functions are designed to resist collisions, and in principle it should be safe to design signature systems to rely on this, but it is more conservative to design signature systems so that collisions serve merely as early-warning signals. PureEdDSA is therefore recommended by default.

Many common use-cases that sign small size messages (a few KB) use the PureEdDSA version:

- RFC 8442 [39] defines only PureEdDSA for TLS 1.2 and earlier.
- RFC 8446 [40] specifies only the use of PureEdDSA to sign the TLS 1.3 transcript.

- RFC 8410 [41] standardizes the use of PureEdDSA in X.509 certificates.
- RFC 8420 [42] defines usage in the Internet Key Exchange Protocol Version 2 (IKEv2).
- RFC 8709 [43] defines usage in Secure Shell (SSH).
- RFC 8037 [44] defines usage in JOSE JWS Signatures.
- For Cryptographic Message Syntax (CMS), RFC 8419 [45] defines a digest of the message by saying:

> [...] In most situations, the CMS SignedData includes signed attributes, including the message digest of the content. Since HashEdDSA offers no benefit when signed attributes are present, only PureEdDSA is used with the CMS.

XML Signatures also include a digest of the message to be signed. In OCSP and OCSP staples (RFC 2560 [46]), the signature is "computed on the hash of the DER encoding ResponseData." What's more, OpenPGP digests the message before signing it. So these use-cases all sign the full message or a digest of it and do not depend on a Prehash version of the signature itself. The size of the data is small enough for the signature to be generated or verified on the fly without issues.

Regarding APIs, traditionally crypto APIs were assuming digests as inputs to a signature. OpenSSL historically had only one API `EVP_PKEY_sign` which assumed digest-then-sign. Of course, that did not satisfy the PureEdDSA variant, so BoringSSL and OpenSSL, two popular open-source cryptographic libraries, distinguish between the PureEdDSA and other hash-then-sign signature schemes in their EVP_MD, EVP_DigestSign*, and EVP_PKEY_Sign and EVP_PKEY_SignInit APIs. For more details, refer to the relevant github discussion.

C.4 The PKCS#11 Challenge

In PKCS#11, RSA and ECDSA signatures can be used with or without pre-hashing. According to the PKCS#11 specification for ECDSA [47], the CKM_ECDSA mechanism assumes a digest of the message or a message truncated to the right size. Arbitrary-length messages are signed with the CKM_ECDSA_SHA256 mechanism, which digests and then signs the message. In either case, ECDSA signs a "short version" of the message. It is either digested externally to the signer (CKM_ECDSA) or inside the signer (CKM_ECDSA_SHA256). RSA is used in similar ways with more legacy options. Note that if the signer is a FIPS-certified module, digesting usually takes place in the signer FIPS boundary as required by the FIPS 140 certification. In this context, 2020/990 [48] proposed for HSMs to use different boundaries for the randomized message digesting and asymmetric signing/verification which is up to the HSM vendor.

PKCS#11 includes a multi-part/incremental API when large messages cannot be stored in memory for the signer and uses `C_SignUpdate` to incrementally digest the message in chunks until it completes the digest and signs it (`C_SignFinal`). The CKM_ECDSA_SHA256 mechanism is used with the incremental API which allows the signer/verifier to take the message piece by piece until it completes the digest and signs/verifies it. In a typical large message scenario, streaming the message to the signer can affect performance. For example, an HSM attached to a network could see significant performance impact for large messages using the incremental API.

If PKCS#11 was taking arbitrary-size as inputs without digesting them beforehand, the incremental API would not work for big messages that cannot be buffered. The whole input would not be available at the signer (`C_SignInit/Update/Final`) before signing or the verifier (`C_VerifyInit/Update/Final`) who receives the signature after the message. PureEdDSA would suffer from that probem. The EdDSA paper [38] explains cases like PKCS#11 where PureEdDSA would not work with the incremental API by saying:

> The main motivation for HashEdDSA is the following storage issue (which is irrelevant to most well-designed signature applications). Computing the PureEdDSA signature of M requires reading through M twice from a buffer as long as M, and therefore does not support a small-memory “Init-Update-Final” interface for long messages. Every common hash function H0 supports a small-memory “Init-Update-Final” interface for long messages, so H0-EdDSA signing also supports a small-memory “Init-Update-Final” interface for long messages.

The PKCS#11 specification also acknowledges the issue by stating:

> Note that for EdDSA in pure mode, Ed25519 and Ed448 the data must be processed twice. Therefore, a token might need to cache all the data, especially when used with C_SignUpdate/C_VerifyUpdate. If tokens are unable to do so they can return CKR_TOKEN_RESOURCE_EXCEEDED.

The latest PKCS#11 API includes only one mechanism for EdDSA, CKM_EDDSA. CKM_EDDSA takes optional CK_EDDSA_PARAMS which indicates if it is the Pure or Prehash variant. PureEdDSA is used by default, which assumes arbitrary message inputs. In cases where the message is big and can’t be cached, CKM_EDDSA is used in its Prehash version. The signer/verifier can keep taking the message as input piece by piece with the incremental API (`C_SignUpdate` / `C_VerifyUpdate`) until it can complete the digest used for PrehashEdDSA signing/verification. Thales seems to also have created its own digest EdDSA mechanisms like CKM_SHA256_EDDSA which hard-codes PrehashEdDSA and its digest function, but no further information is available about these mechanisms.

C.5 Options for Standardization

As new PQC signatures are getting standardized, adopters will need to decide if they want to follow the digest-then-sign paradigm. The options available are:

- All digest operations are handled internally to the sign/verify operation, which is what the three PQC signatures are doing.
- Using a digest-then-sign methodology with or without randomized metadata
  - Digesting takes place before passing the digest to the signature algorithm
  - Digesting takes place inside the signing operation (Prehash signature mode)
- Digest-then-sign by externalizing the PQC signature Internal Digest (which has security implications)

This section describes these options, giving possible use-cases, pros, and cons for each one.

C.5.1 No-digest Before Signing

One option, since post-quantum signatures support it, is to not digest the message and just feed it whole to the signing operation. Without digests, we do not need to depend on collision resistance for the hash function for Falcon and SPHINCS+. We still need collision resistance of the hash function for

Dilithium. This approach also allows for easier security analysis of the signature scheme. Additionally, as shown in section C.5.2, digesting before signing may only have noticeable impact with large messages when the pre-digest function is more efficient than the Internal Digest. For more details on the advantages of this approach, refer to the discussions in Appendix D.

We expect post-quantum signatures that do not digest to be the default approach for standardization and adoption in most use-cases. An example where this method worked is with PureEdDSA, which got adopted although previous cryptographic APIs assumed a digest for RSA and ECDSA. Most uses of post-quantum signatures (e.g., TLS, SSH, X.509, SSH, IKEv2) will operate fine with signing the whole message, as they typically sign relatively short messages. They adopted PureEdDSA for the same reason. Standards like CMS and OpenPGP also sign relatively short data called “signed attributes,” so this method can be used there as well. Since the signed attributes contain a digest of the message, these standards can be considered as an instance of the digest-then-sign paradigm, but they would still make use of a PQC signature primitive that does not pre-digest.

A potential shortcoming of not digesting the message before signing would be the cost of streaming it to the signing entity if it was different than the holder of the message. For example, the cost of I/O for an HSM getting streamed a large message over PCKS#11’s multi-part API could affect signing performance.

PKCS#11 could provide mechanisms corresponding to the pure signature paradigm for each algorithm (e.g., CKM_DILITHIUM, CKM_FALCON, CKM_SPHINCSPLUS). To avoid the challenge with long messages explained in section C.4, PKCS#11 could assume a relatively short input for these mechanisms, for example up to a few tens of kilobytes. These mechanisms would work mainly with the one-part interface (`C_Sign` / `C_Verify`).

Vendors may also support the multi-part/incremental API (`C_SignInit`/`Update`/`Final`) in the same CKM_DILITHIUM mechanism only for Dilithium with a complication. Dilithium digests the message as SHAKE256(tr||M) where tr is the public key. A big input message could be streamed piece by piece when calculating SHAKE256(tr||M) since tr is known to the signer/verifier before receiving the message. This would work well if PKCS#11 adopted only the Dilithium signature. However, doing the same thing for the multi-part interface for Falcon or SPHINCS+ would impose challenges to a Falcon or SPHINCS+ verifier because the nonce is not available at `C_VerifyInit`, and to a SPHINCS+ signer that requires two passes on the message. A multi-part interface for the same mechanism for Falcon and SPHINCS+ (e.g., CKM_FALCON, CKM_SPHINCSPLUS) would require buffering the message, which imposes hard size limits. If PKCS#11 pure PQC signature mechanisms support the incremental API, it may need to be done consistently for all signatures (not just CKM_DILITHIUM) to prevent confusion and inadvertent mistakes for users, and the size constraints should be clearly documented.

C.5.2 Digest-then-sign

Other use-cases may need a hash-then-sign for performance reasons, especially if the message is large. For example, certain applications have messages in the MB or GB range (e.g., firmware and software, large legal documents, CAD files, high-resolution images and scans, video surveillance artifacts). If the pre-hash is faster than the Internal Digest (e.g., SHA2-512 vs SHAKE256), then digest-then-sign would perform better than no-digest before signing for these messages. Using two different primitives increases code size, on the other hand. Similarly, if signing or verification happens in a constrained device, then pre-hashing the message locally and sending only the digest to the signing module may be more efficient. Use-cases that combine classical with post-quantum signatures could also benefit from calculating just one digest for both signatures instead of two.

Although it has generally worked well in the past, digest-then-sign has some issues. One is the potential collision risk if the digest function is found to have collisions. Falcon and SPHINCS+ are naturally resistant to collisions. Introducing a pre-hash re-introduces these risks. Dilithium can be affected by collision attacks but is less vulnerable than plain hash-then-sign since its collisions are specific to a given public key.

To ease the concern, we could use an appropriately large output digest like SHA2-512 or SHAKE256 with 64-byte output size. Arguably, collisions are not a realistic risk, as the SHA-2 and SHA-3 families are unlikely to be found weak against collision attacks and the crypto community knows how to design hash functions now more than it did 20 or 30 years ago. Although it seems unlikely for SHA-2 or SHA-3 to fall victim of new collision attacks, no one can be certain of what the future holds for newer hash functions. Additionally, not requiring collision resistance for the digest simplifies the security proofs for these signatures which use the whole message as input. For more details on the concerns of the digest-then-sign approach, refer to the discussions in Appendix D.

Optionally, the local pre-hashing step may process additional metadata to improve security against collision attacks. That is the digest-with-something-then-sign idea discussed in Appendix D. This must be carefully specified for each use-case. There are two ways to implement digest-with-something-then-sign:

1. Sign a randomized hash of the message with a proper hash function. Schemes could sign the H(nonce||M) where nonce is a random value and M is the message. Given that the nonce is a random value, such an approach would require it to be included as part of the signature envelope. The security would depend on the nonce generation process using proper entropy.

2. Sign an H(pk||M) where pk is the public key of the signer and M is the message which only protects from collisions against multiple signers. That means that we no longer need to include a new nonce value in the signature envelope. The benefit of this approach would be potentially better performance for pre-calculating the hash, but it does not offer general collision resistance.

The first digest-with-something-then-sign option would generally require changes for implementers that now need to parse a nonce along with a signature. It would not work with large messages signed in the context of PKCS#11 because the signature is provided after the message, which means the message is not available to the verifier at the time the nonce is available. Both approaches would also require a change in signing APIs and seem challenging to adopt for the general case.

C.5.2.1 Externally to the Signing Operation

Digest-then-sign or digest-with-something-then-sign can be implemented outside of the signature and fed as input to the signing or verification operation. This has been the approach for RSA and ECDSA in various use-cases. The process can be broken into three steps:

1. Digest the message and protocol-defined metadata (if any).

2. Optionally, append protocol-defined attributes to the digest.

3. Sign the digest and the attributes using the general method (Signed Data is short).

Certain standards, like CMS, S/MIME, and OpenPGP, explicitly require hashing the message and some metadata before signing. These uses follow the digest-then-sign approach, but they would still make use of a signature primitive that does not pre-hash.

The digest-then-sign approach can also be used as an additional PKCS#11 mechanism (e.g., CKM_DILITHIUM_SHAKE256, CKM_FALCON_SHAKE256, CKM_SPHINCSPLUS_SHAKE256) similar to CKM_ECDSA_SHA256. Only hash functions with conservative collision resistance (such as SHA2-512 or SHAKE256 with 64-byte output size) should be supported to alleviate the collision concern. These mechanisms would be readily compatible with both the one-part and the multi-part APIs. Note that hash-with-something-then-sign could also be achieved in PKCS#11 by streaming the signing public key to the module before the message.

Specifically for Dilithium, PCKS#11 could use one mechanism for both the one-part and the multi-part API as explained in Appendix C.5.1.

C.5.2.2 Internally (Prehash signature mode)

Alternatively, pre-hashing could take place in the signature algorithm like with PrehashEdDSA. Specific standardization of a Prehash variant of each post-quantum signature scheme would be necessary by NIST.

The difference between PureEdDSA and PrehashEdDSA lies in the Internal Digest step:

- With PureEdDSA, the Internal Digest is H(r, PK, M).
- With PrehashEdDSA, the Internal Digest is H(str, r, PK, PH(M)), where str is some domain separation string, and PH denotes the pre-hash function.

The additional input in Prehash mode provides domain separation between digest-then-sign and direct signature use-cases. As this requires a modification in the Internal Digest, it is not a generic transformation that would use the signature algorithm as a black box.

This approach prevents the use of improper hashes and ensures the digest is performed inside the crypto module without user manipulation.

The issue with offering two options, one with digesting in the signature itself and one without, is that it reduces interoperability. It also increases technical debt for implementers that now need to support two variants. The wide adoption of PureEdDSA and the limited use of PrehashEdDSA demonstrate that. What is more, giving the option to use a signature that pre-hashes internally could be an unnecessary impediment. If there is no hard-coded digest option in the signature, then a use-case would need to consciously choose the slightly less secure digest-then-sign option.

If it was standardized, use-cases requiring a digest-then-sign workflow should use it within the Prehash variant of the signature scheme. One use-case that could make use of this with large messages is PKCS#11. Support for the Pure and Prehash variants in PKCS#11 could be achieved similarly to what was done for CKM_EDDSA by using different mechanism parameters (like CK_EDDSA_PARAMS). For example, CKM_FALCON would mean pure Falcon by default, and it would support an optional parameter CK_FALCON_PARAMS = { prehash: SHAKE256 } to switch to hash-then-sign. Obviously, the Prehash variant would support the multi-part API without size constraints, unlike the pure variant.

C.5.2.3 By Externalizing the Internal Digest

To avoid streaming a large message to a constrained crypto module, it may be tempting to separate the internal randomized digest from the post-quantum signature. Then the randomized digest could be computed locally, and only the short result would be sent to the crypto module for signing. More precisely, the signer could compute the following:

- μ=SHAKE256(tr||M) locally for Dilithium; send μ to the crypto module for signing.
- P=HashToPoint(r||M, q, n) locally for Falcon; send P to the crypto module for signing.
- d=H_msg(R||PK.seed||PK.root||M) locally for SPHINCS+; send d to the crypto module.

However, this paradigm changes the security model for these signatures by splitting the operation over two separate suboperations. Doing so will most likely be incompatible with cryptographic certifications like FIPS or Common Criteria. Moreover, in the case of Falcon and SPHINCS+, the possibility of malformed digests even introduces a mathematical flaw that makes the algorithms insecure. More details on the implications are given in Appendix D.

C.6 Conclusion

In this appendix, we evaluated the pros and cons of signing a message digest with a post-quantum signature scheme which can sign arbitrary messages. Some use-cases could prefer to digest the message before passing the digest to the signing algorithm for various reasons. We evaluated the alternatives and concluded that the pure post-quantum signatures without any sort of digest will probably be the choice for most use-cases. For some applications, digesting before signing may still make sense.

Appendix D Hash then Sign Previous Discussions

D.1 Internet Research Task Force (IRTF) Crypto Forum Research Group (CFRG)

The Internet Research Task Force (IRTF) Crypto Forum Research Group (CFRG) discussed the topic in a long thread. Various thoughts were expressed which mainly focused around the security concerns of digest-then-sign and alternatives. One message argued that the Internal Digest in Dilithium limits the usability of any found collision to a specific public key but does not frustrate a collision attack against a specific public key. Someone described current HSM use-cases that leverage digests before signing like firmware signing (PKC#11), trusted platform modules (TPMs), and Time Stamp Authorities (TSAs).

The collision resistance requirement of digest-then-sign approach was also discussed. Some argued that if the hash / digest function is broken in terms of collision, then we would have more problems with our post-quantum signatures and that we generally can trust the SHA-2 and SHA-3 families as collision resistant. Some proposed the digesting to be in the envelope, out of the signature. Another response made the point that we could hash and randomize but not move that out of the signature. It also argued that HSMs trusting the digest can be dangerous, and using the randomization in the signature allows for better security analysis.

One more argument for the advantage of binding the signature to the public key was also brought up in the thread. Someone also made the point that the EdDSA went through the exercise of defining two versions, Pure and Prehash, which did not lead to interoperability as only the former was predominantly implemented. There were concerns voiced regarding the size of the message input for HSMs and other use-cases. Different approaches to message streaming digesting were also discussed.

The topic was brought up in another thread in IETF's CFRG WG, where similar arguments were made. The collision resistance concern was discussed again in that thread. One response stressed the importance of using a conservative hash function like SHA2-512 or SHAKE256 for digest-then-sign, and others pointed out that hashes of today are secure and give us sufficient collision resistance. Another response stressed that taking the randomizing digest out of Falcon is a dangerous idea.

In summary of these discussions, signing without digesting first is a more secure approach which allows better security analysis of signature schemes that can take arbitrary size messages as input. Digest-then-sign comes with a collision resistance requirement for the digest function, which can generally be assumed for modern digest functions. So the collision requirement is not a strong one. Taking the randomized hashing out of the signature is probably a bad idea in terms of cryptographic risk. Digesting very large messages can be a concern for some use-cases that incrementally digest the message.

D.2 IETF LAMPS (Limited Additional Mechanisms for PKIX and SMIME) Working Group (LAMPS WG)

The topic was also discussed in two threads (PT7jTztNfI1K6DkS7bQ_SkljoVI, xchLLz0kdM1sUjlCBYNZaPj4jt4) in IETF's LAMPS WG, which deals with certificates and CMS. The former thread overlaps with the CFRG thread summarized in Appendix D.1. In the latter, a few participants expressed support for pre-hashing in principle but without laying out the actual mainstream use-cases.

D.3 NIST PQC Forum

The digest-then-sign discussion also took place in a long thread in NIST's PQC email list. The trigger for this discussion was the PKCS#11 incremental API incompatibility with big messages and the current arbitrary message signing approach of post-quantum signatures.

The initial message pointed out that PKCS#11 uses a one-part `C_Sign`, `C_Verify` and multi-part APIs `C_SignInit/Update/Final`, `C_VerifyInit/Update/Final` which traditionally assumed you can receive and hash the whole message before signing it with RSA and ECDSA. The multi-part API digests the message incrementally until it completes the hash and signs it. So, to make it work for quantum-safe signatures, you would need sign the hash of the message with the new signature. The thread pointed out the collision concern with digest-then-sign and proposed various approaches by changing the way randomized hashing takes place in PQC signatures, but that would affect their security. One could also randomize the hash of the message H(nonce, M) to improve the collision concern, but that would not work because the nonce is part of the signature which comes after the message. That means that the nonce will not be available along with the whole message as the verifier starts incremental verification of very big messages that can't be buffered. Reversing the order in randomized hashing of the message does not work because of collision concerns due to length extension attacks.

Another approach would be taking the randomized hashing of these signatures out of the signature and doing it independently. As it was pointed out, that could have detrimental effects on security, so it is not a good option. Another approach would be to digest the message only for the multi-part APIs and not the one-part ones. The challenge with that would be that there would be two different approaches for the incremental and one-part method. That seemed to be the case with ECDSA as well in PKCS#11 with the CKM_ECDSA and CKM_ECDSA_SHA256 mechanisms.

One more idea mentioned was for the incremental interface only to digest H(pk||m) as the PK will be available before starting the incremental verification and incrementally calculating the digest would be possible. The counterargument against that was the one-part and incremental interface should use the same signing method, and not one without pre-hashing and one with pre-hashing H(pk||m).

The NIST PQC alias saw three more threads (PLAkpoagAQAJ, BuZZpWLaAgAJ, 4MBurXr58Rs) on the topic which overlap with the aforementioned long thread in NIST's PQC email list and the IETF threads discussed in Appendix D above. One comment supported digesting in the signature because we don't have to expose the hash in the API, test vectors are comprehensive, and improper hash functions are not a concern.

D.4 Liboqs and OpenSSL 1.1.1 Signature Performance Platform Details

```
model name     : on Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz
Target platform:  x86_64-Linux-5.4.241-160.348-aws
Compiler:         gcc (7.3.1)
Compile options:  [-Wa,--noexecstack;-O3;-fomit-frame-pointer;-fdata-sections;-ffunction-
sections;-Wl,--gc-sections;-Wbad-function-cast]
OQS version:      0.8.0-rc1
Git commit:       unknown
OpenSSL enabled:  Yes (OpenSSL 1.1.1g FIPS  21 Apr 2020)
AES:             NI
SHA-2:            OpenSSL
```

```
SHA-3:          OpenSSL
OQS build flags:  OQS_DIST_BUILD OQS_OPT_TARGET=generic CMAKE_BUILD_TYPE=Release
CPU exts active:  ADX AES AVX AVX2 AVX512 BMI1 BMI2 PCLMULQDQ POPCNT SSE SSE2 SSE3

OpenSSL 1.1.1g 21 Apr 2020
built on: Mon May  8 16:50:49 2023 UTC
options:bn(64,64) md2(char) rc4(16x,int) des(int) aes(partial) idea(int) blowfish(ptr)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -O3 -O2 -g -pipe -Wall -Wp,-
D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector-strong --param=ssp-buffer-size=4 -grecord-
gcc-switches -m64 -mtune=generic -Wa,--noexecstack -DOPENSSL_USE_NODELETE -DL_ENDIAN -
DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -
DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -
DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DVPAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -
DX25519_ASM -DPOLY1305_ASM -DZLIB -DNDEBUG -DPURIFY -DDEVRANDOM="\"/dev/urandom\""
```

D.5 Security Issues when Externalizing the Internal Digest

Externalizing the Internal Digest out of the signature updates the security model, which increases the attack surface by allowing an attacker to send malicious specially crafted “digests” to the crypto module. Such an attack would mathematically break Falcon and SPHINCS+. For instance:

- Multiple Falcon signatures of the same point P=HashToPoint(r||M, q, n) may reveal information about the private key (hence the randomization of the digest). If the client is responsible for computing HashToPoint, an attacker could send the same point multiple times to obtain multiple valid signatures, and extract the private key.
- If the internal SPHINCS+ digest is an attacker-controlled string, instead of the output of a hash function, then an attacker would be able to choose which parts of the FORS tree it learns at each signature. Sending some specially crafted fake digests would be enough to forge a valid signature for a target message.

Additionally, delegating the randomized hash to the client application means the client must access resources it normally shouldn’t, such as parts of the private key or sampling randomness, which is not always a safe assumption. For instance:

- For Falcon, the Internal Digest is P=HashToPoint(r||M, q, n) where r is a random nonce. This means the client application has to sample its own randomness.
- For SPHINCS+, the Internal Digest is d=H_msg(R||PK.seed||PK.root||M). The issue is with R. Normally, it is generated by hashing a part of the private key, a random seed, and the message. The most obvious issue here is the access to the part of the private key. A single-pass SPHINCS+ variant where R would be sampled randomly would partially solve this, but it would mean that the client application has to sample its own randomness instead.

Such changes would most likely be forbidden in certified implementations.

For Dilithium, the Internal Digest is μ=H(tr||M) where tr=H(pk) is normally precomputed as part of the private key. This is non-sensitive information, so the client could read this field (or recompute it itself from the public key). Decoupling the calculation of μ and the rest of the signature would make implementations more complicated. It would also mean the cryptographic boundary is split between two entities, but the implications are not as serious as for Falcon and SPHINCS+.