WELLS FARGO

September 2025

Wells Fargo Technology Case Study

Transport Layer Security (TLS) Extensions for Post Quantum Cryptography (PQC) Quantum Security (QTLS)

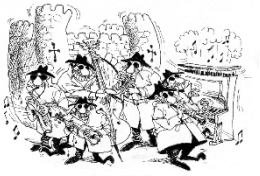
Wells Fargo Technology
Cybersecurity Data Science (CSDS)
Quantum (PQC) Security

Peter Bordow
Jeff Stapleton
Abhijit Rao
Sarah Chen

wolfSSL

Anthony Hu

KEYFACTOR



David Hook
Roy Basmacier

Table of Contents

List of Figures

List of Tables

Abstract

The inevitable cryptographic transition [1] from legacy asymmetric to Post Quantum Cryptography (PQC) algorithms [2] due to the evolution towards a *cryptographically relevant quantum computer* (CRQC) is well understood. For example, Shor's Algorithm [3] will break systems that rely on *legacy* asymmetric cryptography, including RSA algorithm (based on integer factorization), the Diffie Hellman key exchange (based on discrete logarithm), and elliptic curve cryptography (ECC) which uses the algebraic structures of elliptic curves over finite fields. However, enhancing the Transport Layer Security (TLS) protocol [4] which uses aforementioned asymmetric cryptography with PQC based asymmetric cryptography algorithms using various X.509 certificates [5] formats pose unique challenges. This Technology Case Study looks at an implementation of TLS enhanced using the draft X9.146 standard [6] to address those challenges.

The Accredited Standards Committee (ASC) X9 Financial Services[1] is accredited by the American National Standards Institute[2] (ANSI) for developing United States national standards by consensus for the financial services industry. ASC X9 is the USA technical advisory group (TAG) to ISO TC68 Financial Services[3] and the TC68 secretariat. The X9F5 Financial PKI work group is developing the draft X9.146 standard.

Challenges

There are five challenges regarding the transition of the TLS protocol to PQC algorithms for the financial services industry. The draft X9.146 standard defines new TLS 1.3 protocol extensions for negotiating public key certificate formats, public key algorithms, and digital signature algorithms between a TLS client and TLS server. Specifically, the Certificate Key Selection (CKS) extension is for negotiating the public key certificate format and public key selection between a TLS client and server.

The primary challenge is the transition from legacy to PQC algorithms. TLS uses ephemeral public keys, where the DH, ECDH, and ultimately ML-KEM public keys are signed per CertificateVerify messages. For hybrid key establishment, both the legacy and PQC public keys are similarly signed. The legacy RSA or ECDSA and ultimately ML-DSA digital signature algorithms used to (i) sign the CertificateVerify messages and (ii) sign the certificates are verified using the static public keys. See Table 1 Asymmetric Algorithms for an overview.

Table 1 Asymmetric Algorithms

Algorithms	Legacy	NIST PQC
Key Establishment	DH, ECDH	ML-KEM
Digital Signature	RSA, DSA, ECDSA	ML-DSA

The secondary challenge is whether to use hybrid cryptography. For hybrid key establishment a legacy algorithm and a PQC algorithm are both used to establish the symmetric session keys. For hybrid digital signatures, a legacy algorithm and a PQC algorithm are both used for dual signatures.

The third challenge is various certificate formats. There are various certificate formats under consideration, some standardized in X.509 while others are IETF draft proposals. For the purposes of this

[1] ASC X9 https://x9.org/
[2] ANSI https://www.ansi.org/
[3] ISO TC68 https://www.iso.org/committee/49650.html

Technology Case Study, the formats are called Classic, Chimera, Composite, and Chameleon certificates. See Table 4 Certificate Formats for details.

- Classic X.509 certificates contain one public key and one digital signature.
- Chimera X.509 certificates contain two public keys and two digital signatures.
- Composite certificates contain combined public keys and combined signatures.
- Chameleon certificates contain a delta extension to generate a delta certificate.

For example, when using hybrid signatures, a legacy private key and a PQC private key are used to generate dual signatures. Correspondingly, both the legacy public key and PQC public key are needed to verify the hybrid signatures. The two public keys might be in separate Classic certificates or combined in a Chimera, Composite, or Chameleon certificate.

In another example, when transitioning from a legacy algorithm to a PQC algorithm, the legacy public key and the PQC public key might be in separate Classic certificates or combined in a Chimera, Composite, or Chameleon certificate, but only one of the public keys is used at any given time. This might allow a smoother transition to (i) support relying parties who are not PQC-ready, (ii) enable relying parties who are PQC-ready, and (iii) reverse relying parties who encountered PQC issues.

The fourth challenge is key lengths. Generally, the bigger the key, the stronger the cryptography, but bigger keys need more time, more memory, and more storage. While a few extra milliseconds or an extra thousand bits might seem insignificant, handling thousands of TLS connections and managing millions of certificates becomes problematic. And to make things more interesting, NIST changed the descriptions for cryptographic security[4] when it released the Initial Public Draft (IPD). See Figure 1 Image NIST Table 4 NIST Security Strength Categories.

Table 4. NIST Security Strength Categories

Security Category	Corresponding Attack Type	Example
1	Key search on block cipher with 128-bit key	AES-128
2	Collision search on 256-bit hash function	SHA3-256
3	Key search on block cipher with 192-bit key	AES-192
4	Collision search on 384-bit hash function	SHA3-384
5	Key search on block cipher with 256-bit key	AES-256

Figure 1 Image NIST Table 4

Note that NIST removed the Security Strength Categories tables in the published FIPS documents but continues to reference the categories. However, NIST 800-57 Recommendation for Key Management [5] still refers to security strength (bits of security) as comparable key lengths expressed as the number of bits.

The fifth challenge is the dichotomy between public PKI and private PKI. An organization's private PKI is unknown and untrusted by others unless they have a specific contractual agreement and some amount of

[4] FIPS 203 IPD https://csrc.nist.gov/pubs/fips/203/ipd
[5] NIST 800-57 https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

transparency. Conversely, a third-party service provider public PKI requires a Webtrust CA audit[6] providing trust but falls under the CA Browser Forum[7] rules which have become problematic for the financial services industry. Note that the X9 Financial PKI[8] launched in 2015 addresses this issue.

Introduction

This section provides an overview of the Wells Fargo QTLS initiatives and an summary of the X9.146 draft standard under development by the X9F5 Financial PKI work group. For details of the TLS protocol, certificate formats, and the X9.146 certificate key selection (CKS) extension to the TLS protocol see the section *Technical Description*.

Wells Fargo Initiatives

Wells Fargo has been extremely active advancing Post Quantum Cryptography (PQC) solutions for Public Key Infrastructure (PKI) and Transport Layer Security (TLS) technologies. Publicly within the financial services industry, Wells Fargo has been an active participant in ASC X9 work since 2017, and privately within the enterprise, Wells Fargo began its own PQC and QTLS initiatives in 2020. See Table 2 QTLS Timeline for an overview.

Table 2 QTLS Timeline

	2017	2018	2019	2020	2021	2022	2023	2024	2025
X9 PKI Study Group	2017	2018	2019	2020	2021	2022	2023	2024	-
X9 Financial PKI	-	-	-	-	-	-	-	-	2025
X9F5 PKI Work Group	-	-	2019	2020	2021	2022	2023	2024	2025
X9.146 QTLS	-	-	-	2020	2021	2022	2023	2024	2025
WFC PQC Phase 1	-	-	-	2020	2021	-	-	-	-
WFC QTLS Phase 1	-	-	-	2020	-	-	-	-	-
WFC QTLS Phase 2	-	-	-	-	-	2022	2023	2024	2025

Wells Fargo was instrumental in establishing the X9 PKI Study Group in 2017 to determine the best solution for the financial services industry. The study group work concluded in 2025 with the launch of the X9 Financial PKI[9] supporting PQC certificates.

Wells Fargo participates on the X9F4 Financial PKI work group, initiated the QTLS new work item in 2020 , and is the current editor for the development of the X9.146 QTLS draft standard.

Wells Fargo completed its initial PQC Phase 1 performance evaluation of the NIST PQC algorithms in 2021 and its initial QTLS testing in 2020 using X.509 certificates.

Wells Fargo began its secondary QTLS testing of the X9.146 draft standard with wolfSSL in 2022 and Keyfactor in 2024. This Technology Case Study is the culmination of the work with wolfSSL and Keyfactor in 2025.

---

[6] Webtrust CA https://www.cpacanada.ca/en/business-and-accounting-resources/audit-and-assurance/overview-of-webtrust-services/principles-and-criteria

[7] CA Browser Forum https://cabforum.org/

[8] X9 Financial PKI https://x9.org/pki-industry-forum/

[9] X9 Financial PKI https://www.digicert.com/news/digicert-selected-by-asc-x9-to-provide-managed-pki-service-infrastructure

X9.146 Quantum TLS Draft Standard

The draft X9.146 standard defines new TLS 1.3 protocol extensions for negotiating public key certificate formats, public key usage, and digital signatures between a TLS client and server.

- Certificate Key Selection (CKS) extension
- Hybrid Scheme List (HSL) extension

The TLS 1.3 extension called Certificate Key Selection (CKS) is for negotiating the public key certificate format (Classic, Chimera, Composite, Chameleon) and use of the public keys between a TLS client and server. See Table 4 Certificate Formats for more information.

Note that the Chimera, Composite, and Chameleon certificates called "hybrid certificates" contain two public keys and two signatures, but should not be confused with hybrid digital signatures or hybrid key establishment methods.

- Hybrid certificates contain two public keys and two signatures.
- "Hybrid signatures" refers to generating and validating two signatures using different algorithms.
- "Hybrid key establishment" refers to generating two shared secrets using different algorithms.

The TLS 1.3 protocol extension called Hybrid Scheme List (HSL) is for negotiating hybrid signatures between a TLS client and server.

- Hybrid signatures using hybrid certificates
- Hybrid signatures using Classic certificates

The draft X9.146 standard aligns with TLS 1.3 protocol RFC 7250 [7] and RFC 8446 [4] as it applies to signature algorithms and certificate formats.

- The signature_algorithms extension as it applies to signatures in CertificateVerify messages
- The signature_algorithms_cert extension as it applies to signatures in certificates
- The client_certificate_type extension as it applies to client certificate formats.
- The server_certificate_type extension as it applies to server certificate formats.

The TLS protocol with X.509 [5] certificates is widely used in financial services to authenticate financial service providers and protect financial transactions. The ASC X9 Financial PKI Study Group has identified numerous PKI Use Cases relying on TLS certificates including browsers, mobile apps, Automated Teller Machines (ATM), point-of-sale (POS) terminals, aggregators, host systems, and others that interact with financial services.

Note that the scope of this Technology Case Study is limited to the CKS extension. The HSL extension is out of scope.

Financial service providers, represented by the TLS server, operate financial applications that interact with either (i) financial customers or (ii) other financial applications. These financial customers and other financial applications are represented by the TLS client. **From a financial services perspective, it is the TLS server that manages risk and controls the authentication and authorization of the TLS client.** This differs from the browser approach to verify an unknown server so the end-user can trust the data content.

Technical Description

This section provides an overview of the TLS v1.3 protocol, a comparison of the various certificate formats, specification of the X9.146 CKS extension, information on Classic certificate versus Chimera certificates, and guidance from X.509 on using the Chimera alternative extensions for a relying party (RP) and a certificate authority (CA).

Transport Layer Security (TLS)

Table 3 TLS Handshake provides information from the RFC 8446 [4] specification. The TLS 1.3 protocol is significantly different than its predecessors. TLS extensions were allocated, the Encrypted Extension message was added, and all messages after the Client Hello and Server Hello messages are encrypted.

Table 3 TLS Handshake

Client Messages	Network	Server Messages
Client Hello message ⇒	Cleartext	-
-	Cleartext	⇐ Server Hello message
-	Encrypted	⇐ Encrypted Extension message
-	Encrypted	⇐ Certificate Request message
-	Encrypted	⇐ Certificate message
-	Encrypted	⇐ Certificate Verify message
-	Encrypted	⇐ Finished message
Certificate message ⇒	Encrypted	-
Certificate Verify message ⇒	Encrypted	-
Finished message ⇒	Encrypted	-
Send / Receive message ⇔	Encrypted	⇔ Send / Receive message

The CKS extension is used primarily in the Client Hello and Server Hello messages.

Certificate Formats

Table 4 Certificate Formats contrasts the basic differences between Classic [5], Chimera [9], Composite [10], and Chameleon [11] certificates. Classic certificates contain a single public key and a single certificate signature, where the public key and signature are either legacy algorithms or PQC algorithms. The other certificate formats (Chimera, Composite, and Chameleon) contain two public keys, a legacy algorithm and a PQC algorithm, and two signatures, a legacy algorithm and a PQC algorithm.

Table 4 Certificate Formats

Certificate Format	Native	Extensions
Classic	Legacy public key Legacy signature	N/A
Classic	PQC public key PQC signature	N/A
Chimera	Legacy public key Legacy signature	PQC public key PQC signature
Composite	Legacy public key PQC public key	N/A
	Legacy signature PQC signature	N/A

Certificate Format	Native	Extensions
Chameleon	Legacy public key Legacy signature	Delta Certificate
Delta Certificate	PQC public key PQC signature	N/A

Chimera certificates (sometimes called Catalyst or dual-key certificates) contain a public key in the Subject Public Key native field and a certificate signature in the SIGNED native field, and another public key and certificate signature in the alternate extensions, as defined in the X.509 [5] standard. See Table 8 Certificate Structure.

Composite certificates contain two public keys in the Subject Public Key native field and two certificate signatures in the SIGNED native field, as defined in the IETF draft [10] specification.

Chameleon certificates contain a public key in the Subject Public Key native field and a certificate signature in the SIGNED native field, and another public key and certificate signature in the Delta Certificate extension. The relying party dynamically composes a complete Delta Certificate using the X.509 native fields, relevant extensions, and the other public key and certificate signature, as defined in the IETF draft [11] specification.

Note that the scope of this Technology Case Study is limited to Classic and Chimera certificates. Table 5 Extension Identifier Names and Components defines the three new TLS extensions introduced in the X.509:2019 [5] version for the Chimera certificates.

Table 5 Extension Identifier Names and Components

Extension Name	1st Component	2nd Component
subjectAltPublicKeyInfo	{ 2.5.29.72 }	SubjectAltPublicKeyInfo
altSignatureAlgorithm	{ 2.5.29.73 }	AltSignatureAlgorithm
altSignatureValue	{ 2.5.29.74 }	AltSignatureValue

Certificate Key Selection (CKS) Extension

Table 6 CKS Codes provides definitions for Classic, Chimera, Composite, and Chameleon certificate usage, with higher numbers reserved for future use. The External (255) code is reserved for certificate management systems separate from the TLS 1.3 protocol.

Table 6 CKS Codes

CKS Codes		Fields	Extensions	Description
Default	(0)	Native	N/A	Classic: Native only – Alternate not present
Native	(1)	Native	Alternate	Chimera: Native default – ignore Alternate
Alternate	(2)	Native	Alternate	Chimera: Alternate only – ignore Native
Both	(3)	Native	Alternate	Chimera: Native and Alternate
Composite	(4)	Native	N/A	Composite:
Chameleon	(5)	Native	Delta	Chameleon:
Classic	(6)	Native	N/A	Classic: certificate pair
Reserved	(7)	N/A	N/A	Reserved for future use
…	…	…	…	Reserved for future use
Reserved	(254)	N/A	N/A	Reserved for future use

CKS Codes	Fields	Extensions	Description
External (255)	N/A	N/A	Codes are external to TLS protocol

Default (0) is a Classic certificate with only the native subjectPublicKey and caSignatureValue fields.

Native (1) is a Chimera certificate with both native subjectPublicKey and caSignatureValue fields and alternate subjectAltPublicKey and altSignatureValue extensions, but only the native fields are used and the alternate extensions are ignored. This allows backwards compatibility.

Alternate (2) is a Chimera certificate with both native subjectPublicKey and caSignatureValue fields and alternate subjectAltPublicKey and altSignatureValue extensions, but only the alternate extensions are used and the native fields are ignored. This allows forwards interoperability.

Both (3) is a Chimera certificate with native subjectPublicKey and caSignatureValue fields and alternate subjectAltPublicKey and altSignatureValue extensions, such that both signatures and both public keys are used.

Composite (4) is a Composite certificate with a native CompositePublicKey and CompositeSignatureValue such that the composite signature and the composite public key is used.

Chameleon (5) is a Chameleon certificate with native subjectPublicKey and caSignatureValue fields and a DeltaCertificateDescriptor extension from which a Delta Certificate can be created, such that both public keys and both signatures are used.

Classic (6) is two Classic certificates with matching Subject fields, such that both subjectPublicKey fields and caSignatureValue fields are used.

Reserved (7 – 254) are reserved for X9 future use.

External (255) indicates that the CKS codes are managed using some system policy controls external to the TLS protocol.

Note that the scope of this Technology Case Study is limited to the *Default*, *Native*, *Alternate*, and *Both* CKS codes. The CKS codes *Classic* and *External* were out of scope.

X.509 Classic Certificates

Table 7 Certificate Example provides an example certificate in the first column, the native fields noted in the second column, and typical V3 extensions in the third column. The certificate consists of (i) native fields and (ii) extensions, together denoted as TBSCertificate[10] meaning the data to-be-signed, and (iii) the digital signature, generated by the certificate authority (CA) that issued the certificate.

[10] TBSCertificate (To Be Signed Certificate) is the part of an X.509 certificate that contains all the information about the certificate—such as the subject, issuer, validity period, and public key—excluding the signature, and is the portion that is actually signed by the certificate authority.

Table 7 Certificate Example

Certificate fields			Native	Extensions
TBSCertificate	Version		V3	
	Certificate Serial Number		SN	
	Signature Algorithm Identifier		AID	
	Issuer Name		Issuer info	
	Validity		Dates	
	Subject Name		Subject info	
	Subject Public Key Info		Public Key	
	Extensions [3]			
		Authority Key Identifier (AKI)		Hash (Issuer Public Key)
		Subject Key Identifier (SKI)		Hash (Subject Public Key)
		Key Usage		Key Usage flags
		Extended Key Usage		Key Usage OID
		Certificate Policies		Certificate Policy URL
		Subject Alternate Name (SAN)		Subject info
		Basic Constraints		CA flag
		Authority Information Access (AIA)		OCSP URL
	SIGNED {TBSCertificate}		Signature	

Note that the actual extensions will vary by certificate authority (CA), but these are typical extensions included for certificates used by TLS servers and clients. From a PKI perspective, the TLS client is the relying party when it receives the server certificate, and the TLS sever is the relying party when it receives the client certificate. Relying parties first validate the certificate chain from the subject certificate to the Root CA certificate, and then trusting the subject certificate, uses the subject's public key to verify the signature on the ephemeral public key exchanged during the TLS handshake.

Figure 2 Classic Certificate Validation shows the relying party process flow for verifying the signature on the ephemeral key sent by the other party. The ephemeral public key is signed by the subject's static private key and verified using the subject's static public key. However, the relying party validates the certificate chain before it trusts the subject's static public key certificate.

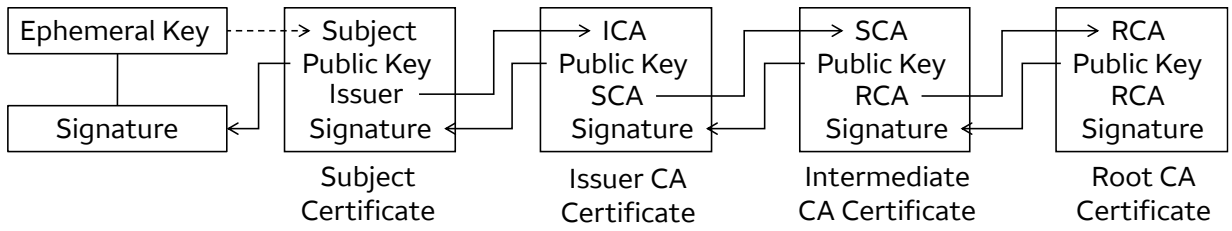


Figure 2 Classic Certificate Validation

The relying party uses the subject's public key certificate to determine the Issuer CA certificate (ICA), uses the ICA certificate to determine the subordinate CA certificate (SCA), and uses the SCA certificate to determine the Root CA (RCA) certificate.

The relying party then uses the RCA public key to verify the RCA certificate and the SCA certificate, uses the SCA public key to verify the ICA certificate, uses the ICA public key to verify the subject's certificate, and uses the subject's public key to verify the ephemeral key signature.

X.509 Chimera Certificates

Table 8 Certificate Structure shows the three certificate extensions defined in X.509:2019 allowing a Certification Authority (CA) to issue certificates containing two public keys, two signature algorithms, and two signatures. The native public key, signature algorithm, and signature can be augmented with a set of alternative values carried in certificate extensions.

Table 8 Certificate Structure

Certificate			Native	Extensions
TBSCertificate	Version		V3	
	Certificate Serial Number		SN	
	Signature Algorithm Identifier		AID	
	Issuer Name		Issuer info	
	Validity		Dates	
	Subject Name		Subject info	
	Subject Public Key Info		Public Key	
	Extensions [3]			
		Alternate Subject Public Key Info		subjectAltPublicKeyInfo
		Alternate Signature Algorithm		altSignatureValue
		Alternate Signature Value		altSignatureAlgorithm
	SIGNED {TBSCertificate}		Signature	

X.509 [5] provides guidance for a relying party to use alternative extensions.

> It is possible for a public key certificate to hold:
> - a subject's alternative public key information to be used instead of the information provided in the subjectPublicKeyInfo component;
> - an alternative signature algorithm to be used instead of the algorithm specified in the signature component; and
> - an alternative digital signature to be checked instead of the native digital signature.
>
> Such alternative specifications are provided as public key certificate extensions.
>
> A relying party that has not migrated to support alternative cryptographic algorithms and alternative digital signatures shall verify the native digital signature.
>
> A relying party that has migrated to support alternative cryptographic algorithms and alternative digital signatures shall verify the alternative digital signature.

X.509 [5] also provides guidance to the CA for signing certificates with alternative extensions.

> A CA generating a public key certificate with the alternative cryptographic algorithms and alternative digital signature shall:
> - when generating the value in the altSignatureValue extension, exclude the signature component and the altSignatureValue extension from the public key certificate, and generate the digital signature over the remaining DER encoded public key certificate using the algorithm specified in the altSignatureAlgorithm extension; and
> - when generating the value in the signature component, the subjectAltPublicKeyInfo, the altSignatureAlgorithm and the altSignatureValue extensions shall be included in the DER encoding of the public key certificate.

The X.509 objective is for a relying party to use either the native public key and the certificate signature, or the alternative public key and certificate signature, but not both. In comparison, the IETF inclination towards hybrid cryptography has promoted the Composite and Chameleon certificates.

Figure 3 Chimera Certificate Validation shows the relying party process flow for verifying the signature on the ephemeral key sent by the other party. The ephemeral public key is signed by the subject's static native or alternative private key and verified using the subject's static native or alternative public key. Likewise, the relying party validates the certificate chain before it trusts the subject's static public key certificate.

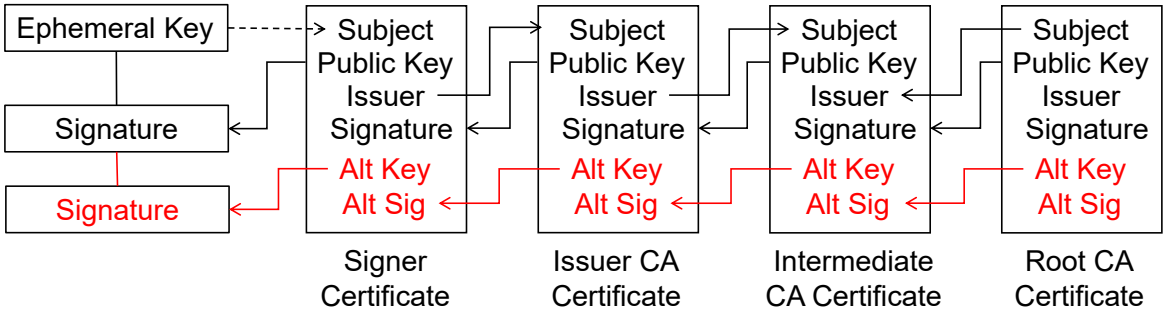


Figure 3 Chimera Certificate Validation

The relying party uses the subject's public key certificate to determine the Issuer CA certificate (ICA), uses the ICA certificate to determine the subordinate CA certificate (SCA), and uses the SCA certificate to determine the Root CA (RCA) certificate.

The relying party then uses the RCA native (or alternative) public key to verify the RCA certificate and the SCA certificate, uses the SCA native (or alternative) public key to verify the ICA certificate, uses the ICA native (or alternative) public key to verify the subject's certificate, and uses the subject's native (or alternative) public key to verify the ephemeral key signature.

Note that the CKS code *Both* (3) was added[11] to support using Chimera certificates for hybrid signatures.

[11] Anthony Hu (wolfSSL) and Mike Ounsworth (Entrust) both discussed hybrid signatures with Jeff Stapleton (Wells Fargo) at ICMC23 on September 22, 2023 in Ottawa Canada.

Lab Environments

This section provides an overview of the wolfSSL, Wells Fargo, and Keyfactor lab environments. wolfSSL developed the X9.146 software and posted beta wolfCrypt software on GitHub [13]. Wells Fargo downloaded the wolfCrypt software and tested the X9.146 CKS codes. Keyfactor downloaded the wolfSSL software, implemented the X9.146 beta software, and posted beta Bouncy Castle software on GitHub [14]. Keyfactor also tested the X9.146 CKS codes using both wolfCrypt and Bouncy Castle.

wolfSSL Lab Environment

wolfSSL does a plethora of testing as can be seen by their "wolfSSL Rainbow of Testing!" – see Figure 4 wolfSSL Rainbow of Testing.

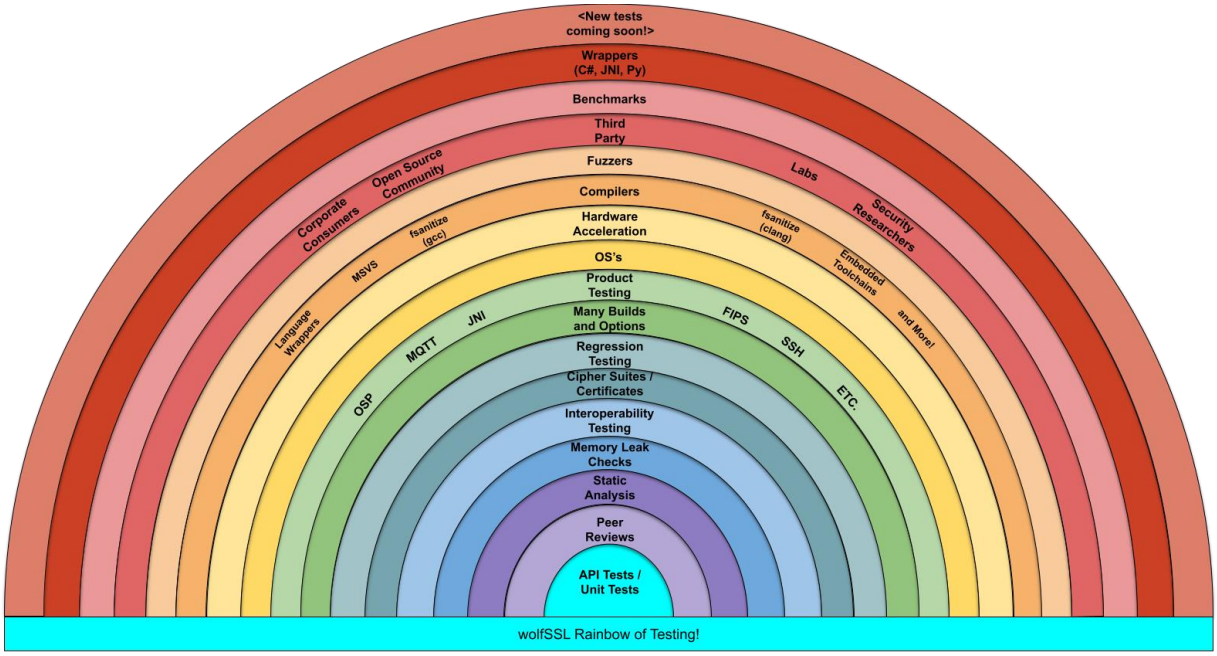


Figure 4 wolfSSL Rainbow of Testing

Once a feature or *fixup* is completed, testing begins. The testing primarily occurred in in two different environments. The first is a physical testing site based in Bozeman, Montana composed of multiple hardware platforms for multi-architecture testing, orchestrated by an instance of Jenkins. The second environment is the wolfSSL GitHub test suite. Combined, these enable thousands of runs of tens of thousands of test suites. Finally, each individual engineering member of the wolfSSL team adds their own tests to the test suite with ensuring every feature is working as expected and no future changes will break them.

Wells Fargo Lab Environment

The Wells Fargo lab was configured to evaluate the *Default* (0), *Native* (1), *Alternate* (2), and *Both* (3) CKS codes with Classic and Chimera certificates using wolfSSL beta software. Two servers running wolfSSL were used for Stage 1 and Stage 2 testing along with a third sever for monitoring and packet capture. The first sever (csl01) was used for Stage 1 testing with a TLS client and TLS server on the same machine. The second server (csl02) was used for Stage 2 testing with a TLS client on the second machine and the TLS

sever on the first machine. All three servers were remotely accessed over an internal connection such as SSH or RDP. See Figure 5 Wells Fargo Lab for an overview and Table 9 WFC Specifications for details.

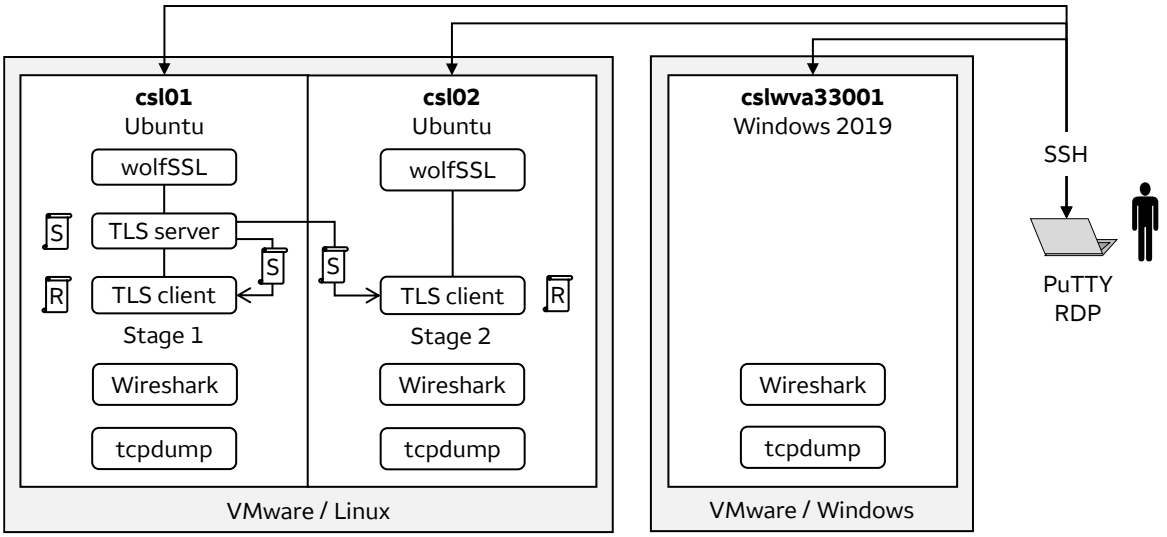


Figure 5 Wells Fargo Lab

Table 9 WFC Specifications

Specifications	csl01	csl02	cslwva33001
Processor Model:	Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz, 4 Core(s)	Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz, 4 Core(s)	Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz, 2300 Mhz, 2 Core(s), 2 Logical Processor(s)
Hypervisor:	VMware (full)	VMware (full)	VMware (full)
Operating System:	Ubuntu 22.04.5 LTS	Ubuntu 22.04.5 LTS	Microsoft Windows Server 2019 Standard
Kernel:	Linux 5.15.0-125-generic	Linux 5.15.0-125-generic	N/A
Architecture:	X86-64	X86-64	x64
Memory:	16GB	16GB	10GB

Server csl01 contained a TLS client and TLS server using Classic and Chimera certificates. The TLS client was provided a Classic root CA certificate, denoted {R} for testing the *Default* (0) CKS code, and a Chimera root CA certificate [R] for testing *Native* (1), *Alternate* (2), and *Both* (3) CKS codes. The TLS server was provided a Classic server certificate, denoted {S} for testing the *Default* (0) CKS code, and a Chimera sever certificate, demoted [S] for testing *Native* (1), *Alternate* (2), and *Both* (3) CKS codes. Client authentication was not used so TLS Client certificates were not generated. The TLS Server sends its certificates {S} and [S] in the Certificate message to the TLS Client. Wireshark with a command line interface (CLI) and tcpdump were used to capture information.

Server csl02 contained only a TLS client using Classic and Chimera certificates. The TLS client was provided a Classic root CA certificate, denoted {R} for testing the *Default* (0) CKS code, and a Chimera root CA certificate, denoted [R] for testing *Native* (1), Alternate (2), and Both (3) CKS codes. Client authentication was not used so TLS Client certificates were not generated. The TLS Server csl01 sends its certificates {S} and [S] in the Certificate message to the TLS Client csl02. The team used Wireshark with a command line interface (CLI) and tcpdump to capture information.

Server cskwva33001 contained Wireshark with a graphical user interface (GUI) and tcpdump was used to view the data captured on csl01 and csl02. Python scripts were used (i) to generate the certificate chain together with wolfSSL utilities and (ii) to manually initiate TLS client hello message to the TLS server. Access to the lab was done remotely over SSH connections using PuTTY for the Linux server and RDP for the Window server.

Steps to Configure the Wells Fargo Lab

To support CKS flag functionality, the wolfSSL GitHub repository was pulled and the following changes were made to the source code:

1. In wolfssl/examples/client/client.c - code was implemented to include input of CKS flag value of 0, 1, 2 or 3 and map these values to the CKS enumeration in wolfSSL for the TLS handshake. No input for this variable is treated as WOLFSSL_CKS_SIGSPEC_BOTH as the default value

2. When initiating the TLS connection on the client side, an additional command line option to accommodate the CKS flag --cks <num> is expected where:
   --cks 0 maps to *Default* (0)
   --cks 1 maps to *Native* (1)
   --cks 2 maps to *Alternate* (2)
   --cks 3 maps to *Both* (3)

The mapping is required because the WolfSSL code and X9.146 specification enumerate the CKS property with different values.

3. In wolfssl/config.c - A flag was added to include CKS support – “softcode-cks-flag”.

4. In wolfssl/wolfssl/ssl.h, a WOLFSSL_CKS_SIGSPEC_DEFAULT was defined to support CKS 0.

5. In wolfssl/src/tls.c, the code was modified to support CKS *Default* (0) functionality.

The following command lines were entered in the terminal to build from the local GitHub directory of wolfSSL:

- ./autogen.sh
- ./configure --enable-experimental --enable-dual-alg-certs --enable-dilithium --enable-debug **--** enable-softcode-cks-flag --disable-shared --enable-keylog-export
- make
- sudo make install
- sudo ldconfig

The following artifacts are required for the generation of valid X9.146 Chimera certificates:

- CA and Server RSA-2048 public/private key-pairs
- CA and Server ML-DSA-44 public/private key-pairs
- CA signed chimera certificates
- Server chimera certificate signed by CA

Pre-canned X9.146 certificates can be found in the GitHub wolfssl-examples/certs/ directory. The code to generate the Chimera certificates are found in the wolfssl-examples/X9.146 directory, which accepts a

pre-canned PQC certificate and PQC server key from wolfssl-examples/certs/ along with a key-pair generated using a legacy algorithm.

Wells Fargo used the wolfssl-examples/X9.146/gen_rsa_mldsa_dual_keysig_cert.c to generate the RSA/MLDSA Chimera certificates using the RSA-2048 and MLDSA-44 algorithms.

Steps to Generate Keys and Certificates
To generate the RSA-2048 keys, the openssl genpkey command is invoked via the terminal, producing keys with DER encoding. The RSA-2048 public key for the CA is outputted, named ca-key-rsa2048.der while the Server key is named server-key-rsa2048.der:

- openssl genpkey -algorithm rsa -pkeyopt rsa_keygen_bits:2048 -out ca-key-rsa2048.der -outform der
- openssl genpkey -algorithm rsa -pkeyopt rsa_keygen_bits:2048 -out server-key-rsa2048.der -outform der

To generate the Chimera certificates for both the root CA and the TLS server, wolfSSL includes C code-based executables gen_rsa_mldsa_dual_keysig_root_cert and gen_rsa_mldsa_dual_keysig_server_cert producing DER encoded root CA and the TLS server certificates. Changes were made to the source code of gen_rsa_mldsa_dual_keysig_cert.c to configure ML-DSA security category levels (see Figure 1 Image NIST Table 4) and certificate output file names taking in as input the test case id. This allows for customizable selection of the ML-DSA key strengths used in the certificates based on user input.

The wolfssl-examples/certs/ directory contains the pre-canned Classic PQC certificate artifacts to be used for certificate generation. The ML-DSA-44 alternative private signature key was copied from this directory into the directories containing the artifacts for each test case. This was done by executing the following command on the terminal while residing in the target test case directory:

- cp {path to}/wolfssl-examples/certs/mldsa44_server_key.pem server-mldsa44-key-pq.pem

To generate the Chimera certificate for the root CA, the command line option would adhere to the following format:

{path to}/gen_rsa_mldsa_dual_keysig_root_cert {dilithium level} {test oid} {path to}/certs

Note: NIST Security Strength Categories 2/3/5 maps to ML-DSA 44/65/87 parameter sets. For instance, if you want to use ML-DSA-44, you would replace the {dilithium level} placeholder with 2. By supplying the test case id, output certificate file names reflect the signature algorithms, their key strengths and the CKS value selected during certificate generation. The output file names follow the following format:

[client/server]-pq-[classic algorithm]-[pqc algorithm]-[cks].der

To generate the Chimera certificate, the following command line option needs to be invoked.

For the signed server certificate:

{path to}/gen_rsa_mldsa_dual_keysig_server_cert 2 {test case id} {path to}/certs

For the signed ca certificates:

{path to}/gen_rsa_mldsa_dual_keysig_root_cert 2 {test case id} {path to}/certs

This results in two X.509 Chimera certificates: “ca-cert-pq-rsa2048-mldas44-nat.der” and “server-cert-pq-rsa2048-mldsa44-nat.der”.

OpenSSL’s x509 command is then called to convert the root cert, the server cert, and the server key from DER encoding to PEM encoding:

- openssl x509 -in ca-cert-pq-rsa2048-mldsa44-nat.der -inform der -out ca-cert-pq-rsa2048-mldsa44-nat.pem -outform pem
- openssl x509 -in server-cert-pq-rsa2048-mldsa44-nat.der -inform der -out server-cert-pq-rsa2048-mldsa44-nat.pem -outform pem
- openssl pkey -in server-key-rsa2048.der -inform der -out server-key-rsa2048.pem -outform pem

This completes the Wells Fargo lab configuration.

Keyfactor Lab Environment

Development was carried out on a branch of the bc-java repository on GitHub as found at:

https://github.com/bcgit/bc-java

The backend development system for Bouncy Castle runs under GitLab with regression and continuous integration builds being carried out on independent Linux boxes configured as GitLab runners. The current runnable branch code, for testing purposes on compatible environments, is available on GitHub at:

https://github.com/bcgit/bc-java-x9.146

For operational testing, the Keyfactor Lab Environment was configured to evaluate the *Default* (0), *Native* (1), *Alternate* (2), and *Both* (3) CKS codes with Chimera certificates using Bouncy Castle. Interoperability testing was performed across all permutations of Bouncy Castle (BC) TLS server, BC TLS client, wolfSSL TLS server, and wolfSSL TLS client. Conventional/post-quantum keys were generated and converted to PEM encoding using OpenSSL. Certificate chains were generated using wolfSSL utilities. See Table 10 Keyfactor Specifications.

The following algorithm combinations were tested for interoperability:

- P-256/MLDSA44,
- P-384/MLDSA65,
- P-521/MLDSA8,
- RSA-3072/MLDSA44.

Client authentication was not used, so TLS Client certificates were not generated.

Table 10 Keyfactor Specifications

Specifications	Gitlab Runner 1	Gitlab Runner 2	Operational
Processor Model:	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 8 Cores	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 8 Cores	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 6 Cores
Operating System:	Ubuntu 22.04.4 LTS	Ubuntu 22.04.4 LTS	Pop!_OS 22.04 LTS
Kernel:	Linux 5.15.0-125-generic	Linux 5.15.0-125-generic	Linux version 6.9.3-76060903-generic
Architecture:	X86-64	X86-64	x64
Memory:	32GB	32GB	32GB

Lab Experiments

This section summarizes the X9.146 CKS codes tested in the labs. A list of Test Case IDs were prepared to identify and prioritize the tests to be performed. The CKS test case scenarios are enumerated as an ordered list of conditions to track which tests are included and which are excluded for this testing. The enumerated conditions are organized as follows.

Client **.** Server **.** CKS **.** Native **.** ALT **.** KM **.** TLS

Client values are (1) wolfSSL, (2) OpenSSL, and (3) Keyfactor Bouncy Castle (BC)

Server values are (1) wolfSSL, (2) OpenSSL, and (3) Keyfactor Bouncy Castle (BC)

CKS values are (0) *Default*, (1) *Native*, (2) *Alternate*, (3) *Both*, and (4) *External*

Native values are (1) RSA and (2) ECDSA

a) Note the X.509 Native fields are the static public key used to verify the signature on the ephemeral public key used to negotiate the shared secret between the TLS client and server
   - The client derives the symmetric keys from the shared secret
   - The server derives the symmetric keys from the shared secret

b) Note that FIPS 186-5 February 2023 disapproved the Digital Signature Algorithm (DSA)

c) Note that FIPS 186-5 February 2023 approved the Edwards Curve Digital Signature Algorithm (EdDSA) specified in IETF RFC 8032 January 2017

ALT values are (3) ML-DSA and (4) SLH-DSA

a) Note the X.509 ALT fields are the static public key used to verify the signature on the ephemeral encapsulation public key for key encapsulation to exchange the encapsulated symmetric key

b) Note draft FIPS 204 August 2023 defines the Module-Lattice-Based Digital Signature Standard (ML-DSA) which is denoted (3) to distinguish from the Native algorithms.

c) Note draft FIPS 205 August 2023 defines the Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) which is denoted (4) to distinguish from the Native algorithms

KM values are (1) ECDHE and (2) ML-KEM

a) Note that for ECDHE the client and server generate ephemeral keys and exchange ephemeral public keys to establish the shared secret, but only the server signs its ephemeral public key.
   - For Classic certificates the ECDHE ephemeral public keys are signed using Native field public key algorithms.
   - For Chimera certificates the ECDHE ephemeral public keys are signed using Native field public key algorithms.

b) Note that for ML-KEM the client generates (KeyGen) the public (encapsulation) and private (decapsulation) keys and sends the public key to the server, the server generates (Encaps) and sends the ciphertext the client, and the client recovers (Decaps) the shared secret. [12] [15]

c) server generates ephemeral keys and sends its signed ephemeral encapsulation public key to the client for key encapsulation.
   - For Chimera certificates the ML-KEM ephemeral encapsulation public key is signed using the ALT extension public key algorithms

TLS values refer to the TLS 1.3 cipher suites defined in RFC 8446

(1) TLS_AES_128_GCM_SHA256
(2) TLS_AES_256_GCM_SHA384
(3) TLS_CHACHA20_POLY1305_SHA256
(4) TLS_AES_128_CCM_SHA256
(5) TLS_AES_128_CCM_8_SHA256

There are 480 possible test cases. However, the QTLS proof-of-technology is designed to determine the feasibility of a quantum-resistant TLS using the Classic and Chimera certificates that will provide the financial service industry a migration path from today's legacy cryptographic algorithms to the NIST post quantum cryptographic (PQC) algorithms. This initiative is not intended to provide a minimum viable product (MVP) or a generally available (GA) product. Consequently, the PQC Lab test choices reduce the totals to 16 test cases.

- **Client** is (1) wolfSSL or (3) Bouncy Castle (BC)
- **Server** is (1) wolfSSL or (3) Bouncy Castle (BC)
- **CKS** is (0) *Default*, (1) *Native*, (2) *Alternate*, or (3) *Both*
- **Native** is (1) RSA-2048 with SHA2-256 only
- **ALT** is (2) ML-DSA-44 with SHA2-256 only
- **KM** is (1) ECDHE-256 or (2) ML-KEM-512
- **TLS** cipher suite is (2) TLS_AES_256_GCM_SHA384 only

Table 11 CKS Test Cases

Test Case	Client	Server	CKS	Native	Alt	KM	TLS Cipher Suite
1.1.0.1.0.1.2	wolfSSL (1)	wolfSSL (1)	CKS (0)	RSA (1)	N/A (0)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.1.1.1.3.1.2	wolfSSL (1)	wolfSSL (1)	CKS (1)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.1.2.1.3.1.2	wolfSSL (1)	wolfSSL (1)	CKS (2)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.1.3.1.3.1.2	wolfSSL (1)	wolfSSL (1)	CKS (3)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.3.0.1.0.1.2	wolfSSL (1)	BC (3)	CKS (0)	RSA (1)	N/A (0)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.3.1.1.3.1.2	wolfSSL (1)	BC (3)	CKS (1)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.3.2.1.3.1.2	wolfSSL (1)	BC (3)	CKS (2)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
1.3.3.1.3.1.2	wolfSSL (1)	BC (3)	CKS (3)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.1.0.1.0.1.2	BC (3)	wolfSSL (1)	CKS (0)	RSA (1)	N/A (0)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.1.1.1.3.1.2	BC (3)	wolfSSL (1)	CKS (1)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.1.2.1.3.1.2	BC (3)	wolfSSL (1)	CKS (2)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.1.3.1.3.1.2	BC (3)	wolfSSL (1)	CKS (3)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.3.0.1.0.1.2	BC (3)	BC (3)	CKS (0)	RSA (1)	N/A (0)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.3.1.1.3.1.2	BC (3)	BC (3)	CKS (1)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.3.2.1.3.1.2	BC (3)	BC (3)	CKS (2)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)
3.3.3.1.3.1.2	BC (3)	BC (3)	CKS (3)	RSA (1)	ML-DSA (3)	ECDHE (1)	TLS_AES_256_GCM_SHA384 (2)

wolfSSL Lab Experiments

The X9.146 README in the wolfssl-examples GitHub repo explains how to demonstrate Chimera certificates in the context of a TLS 1.3 connection as specified by X9.146. Chimera certificates have dual public keys and signatures in a single certificate. The README provides detailed setup and execution instructions for post-quantum cryptographic demos hybridizing conventional algorithms (ECDSA/RSA) with ML-DSA signatures. The steps include certificate creation and the execution of a TLs 1.3 server and client that perform the TLS handshake with hybrid authentication as specified by X9.146. The document

can be obtained by cloning the wolfssl-examples repo (see https://github.com/wolfssl/wolfssl-examples) or can be viewed by going to the following link (see https://github.com/wolfSSL/wolfssl-examples/blob/master/X9.146/README.md).

Wells Fargo used the demonstration provided by wolfSSL as a baseline to replicate the process of establishing a TLS 1.3 connection with X9.146 specifications. Test cases were handled per the instructions provided in the wolfSSL README to generate and run RSA-ML-DSA certificates.

Wells Fargo made the following adjustments to some of the source material (code and configurations) to refine requirements for the selected test cases and o further the established steps to generate Chimera certificates and run a TLS connection:

- added support in the configurations to account for CKS codes
- added support to include native certificates (CKS 0)
- added support to customize RSA-MLDSA key strengths for Chimera certificate generation
- added support to map generated Chimera certificate file names to reflect signature algorithms

While wolfSSL specifies using RSA-3072 in their RSA Demos, Wells Fargo opted to use RSA-2048 as the legacy signature algorithm for certificate generation. Additionally, as per a previous version of the wolfSSL README file, OpenSSL commands were used to generate the RSA key-pairs instead of the wolfssl commands used under their RSA Demo. The commands for generating keys using OpenSSL are listed in their "Generating a Certificate Chain and Adding Alterative keys and Signatures" portion of their README file. For details, refer to section *Steps to Configure the Wells Fargo Lab*.

Wells Fargo Lab Experiments

The following test cases in Table 12 Selected CKS Test Cases were selected from Table 11 CKS Test Cases to demonstrate the operability of the X9.146 Classic and Chimera certificates.

Table 12 Selected CKS Test Cases

Test Case ID	Native Field	Alternative Field	CKS X9.146	CKS Enumeration in wolfSSL
1.1.0.1.0.1.2	RSA-2048	N/A	0	WOLFSSL_CKS_SIGSPEC_NATIVE
1.1.1.1.3.1.2	RSA-2048	ML-DSA-44	1	WOLFSSL_CKS_SIGSPEC_NATIVE
1.1.2.1.3.1.2	RSA-2048	ML-DSA-44	2	WOLFSSL_CKS_SIGSPEC_ALTERNATIVE
1.1.3.1.3.1.2	RSA-2048	ML-DSA-44	3	WOLFSSL_CKS_SIGSPEC_BOTH

Using the command line options from the *Wells Fargo Lab Configuration* section, artifacts were generated for each test case in the table. For the purposes of demonstrating the steps for the TLS connection, test case 1.1.0.1.0.1.2 is used as an example for this experiment writeup. Below is a screenshot that shows the contents of the directory containing the artifacts for 1.1.0.1.0.1.2 after finishing all the generation, along with each artifact's respective file size in KB – see Table 13 QTLS Certificates and Keys.

Table 13 QTLS Certificates and Keys

File Name	Size	Description	Algorithm	Notes
ca-cert-pq-rsa2048-mldsa44-nat.der	4.7K	DER encoded CA Chimera certificate	RSA-2048 MLDSA-44	Direct output from code

File Name	Size	Description	Algorithm	Notes
ca-cert-pq-rsa2048-mldsa44-nat.pem	6.5K	PEM encoded CA Chimera certificate	RSA-2048 MLDSA-44	Converted from DER file – size is significantly larger than the DER file
ca-key-rsa2048.der	1.2K	DER encoded legacy key	RSA-2048	Used for CA Chimera certificate generation
server-cert-pq-rsa2048-mldsa44-nat.der	4.7K	DER encoded server Chimera certificate	RSA-2048 MLDSA-44	Direct output from code
server-cert-pq-rsa2048-mldsa44-nat.pem	6.4K	PEM encoded server Chimera certificate	RSA-2048 MLDSA-44	Converted from DER file – size is significantly larger than the DER file
server-key-rsa2048.der	1.2K	DER encoded legacy server private key	RSA-2048	Used for server Chimera certificate generation
server-key-rsa2048.pem	1.7K	PEM encoded legacy server private key	RSA-2048	Used to establish the TLS 1.3 connection – size is larger than the DER file
server-mldsa44-key-pq.pem	5.3K	PEM encoded PQC private key	MLDSA-44	Used to generate CA/server Chimera certificate and used to establish the TLS 1.3 connection

Steps to Run the TLS Connection using tcpdump

A folder with scripts encoded in Python to automate the TLS connection can be found within the directory containing the folders with artifacts from the different test cases. The TLS connection is run with tcpdump listening inand outputting a PCAP file containing network data that was captured during the connection. The PCAP file follows the naming convention of {test_case_id}_{cks}.pcap. This file is stored in the respective folder of the test case. Note that tcpdump must be initialized before calling the automation script in the separate terminals. Assuming that we are already in the directory containing the artifacts for 1.1.1.1.3.1.2 execute the following command to initiate the listener:

- tcpdump --interface lo -X -w 1.1.1.1.3.1.2_cks1.pcap

Once the certs and keys are ready, a prepared python script “automate_tls.py” launches the client server based on the Test-Case ID and the tools to capture network traffic and resource usage. The script takes an input of the appropriate Test Case ID and an input to initiate the terminal as a server or client. By initiating the “automate_tls.py” script on the two remaining free terminals for the Ubuntu machine, the client and server are launched respectively using the commands found on the wolfssl-examples/README.md.

Execute the following command line to run the “automate_tls.py” python script.

- python3 automate_tls.py

Once the script has been initiated, it will prompt for the Test Case ID as an input. If the entered test case ID is valid, it then prompts the user to select whether to launch the client (1) or the server (2) for wolfSSL. Below shows 1.1.1.1.3.1.2 being selected as the test case with the option to launch server-side – see Figure 6 Prompts.

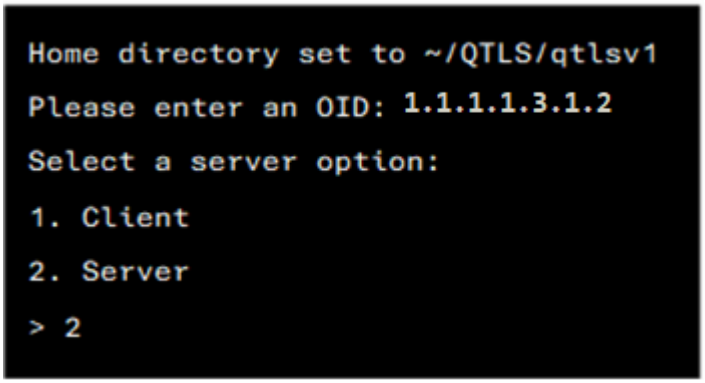


Figure 6 Prompts

To initiate the client, run the script on a separate client-side terminal and select option 1 after entering in the same Test Case ID. Below is a screenshot of the TLS connection after using the script to launch both client(right) and server(left) – see Figure 7 Client Script.

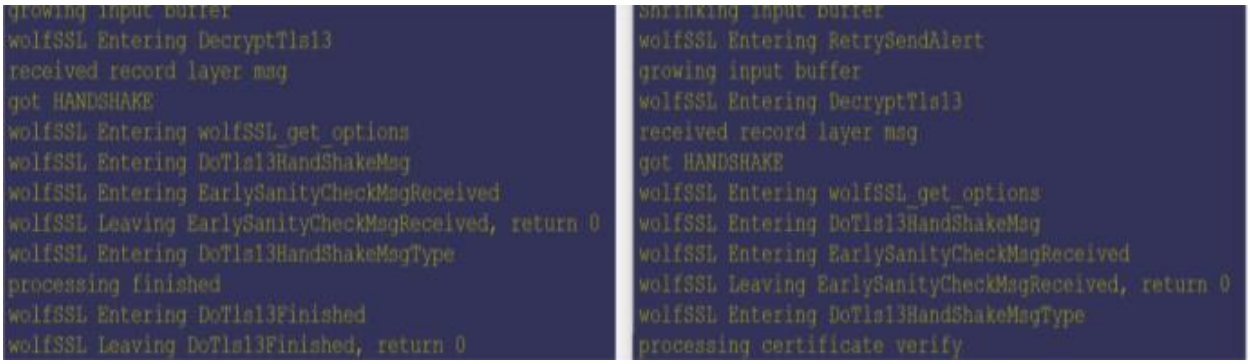


Figure 7 Client Script

Here is a screenshot of the termination of the handshake – see Figure 8 Handshake Termination.

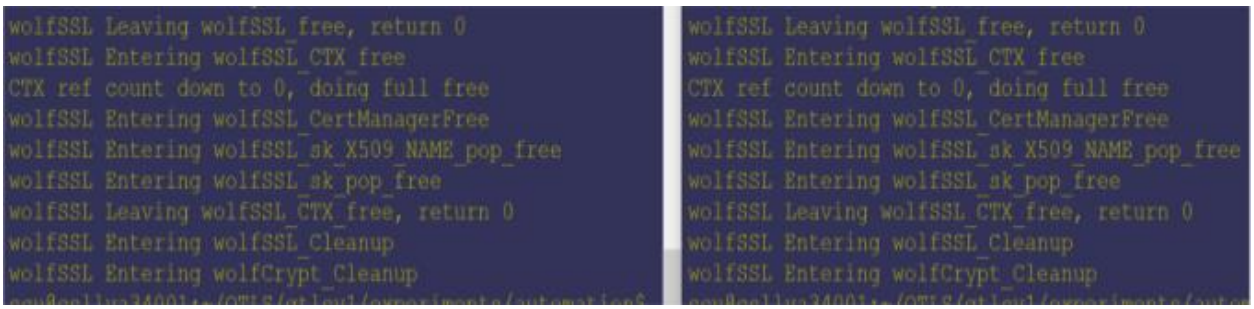


Figure 8 Handshake Termination

After the handshake is complete, the tcpdump listener needs to be terminated. The directory should then contain the newly generated PCAP file from tcpdump – see Figure 9 TCP DUMP.

Figure 9 TCP DUMP

Note that every handshake executed gets logged in a log file under the wolfssl directory, “sslkeylog.log” where the client and server traffic secret is recorded. Below is a screenshot of what the log file looks like – see Figure 10 SSL Key Log.

Figure 10 SSL Key Log

For more examples of keylog files generated by wolfSSL, please refer to the wolfSSL directory wolfssl/scripts.

Steps to Repeat TLS Connection to Capture Metrics with perf stat and time

The “automate_tls.py” script also supports options to run the connection with the “perf stat” or “time” command to capture metrics on the client side. Note that when using these options to collect metrics, it is unnecessary to run tcpdump. To run the script with metrics collection, the script will prompt for the option to enable metric collection and accept an additional input after selecting to run as the client. Depending on user selection, the script will then execute perf stat (1) or time (2) to capture metrics for the selected test case. Below shows 1.1.1.1.3.1.2 being selected with metrics collection enabled – see Figure 11 Metrics.


```
Home directory set to ~/QTLS/qtlsv1
Please enter an OID: 1.1.1.1.3.1.2
Select a server option:
1. Client
2. Server
> 1
Would you like to record metrics? (y/n)y
1. perf stat
2. time
>
```

Figure 11 Metrics

After selecting *perf stat* or *time*, the client side will be set to run with that option. After the client-server TLS connection is terminated, *perf stat* or *time* will output the performance metrics it collected.

Steps to Reviewing PCAP Files from tcpdump in Wireshark

The PCAP files generated from the tcpdump sessions, along with the sslkeylog.log are then transferred to a Windows 2019 lab machine using scp. This Windows machine is used to open up the PCAP files via Wireshark, where the following data for each PCAP file is collected:

- Time taken to complete handshake
- Time taken to tear down client
- Client hello CKS data
- Server hello CKS data

Below is a screenshot of the PCAP file for 1.1.1.1.3.1.2 as viewed in Wireshark – see Figure 12 Packet Capture.

Time	Source	Destination	Protocol	Length	Info
1 0.000000	127.0.0.1	127.0.0.1	TCP	74	35640 → 11111 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=1666479247 TSecr=0 WS=128
2 0.000016	127.0.0.1	127.0.0.1	TCP	74	11111 → 35640 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=1666479247 TSecr
3 0.000028	127.0.0.1	127.0.0.1	TCP	66	35640 → 11111 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1666479247 TSecr=1666479247
4 0.000574	127.0.0.1	127.0.0.1	TLSv1.3	577	Client Hello
5 0.000587	127.0.0.1	127.0.0.1	TCP	66	11111 → 35640 [ACK] Seq=1 Ack=512 Win=65024 Len=0 TSval=1666479248 TSecr=1666479248
6 0.003139	127.0.0.1	127.0.0.1	TLSv1.3	194	Server Hello
7 0.003348	127.0.0.1	127.0.0.1	TCP	66	35640 → 11111 [ACK] Seq=512 Ack=129 Win=65408 Len=0 TSval=1666479250 TSecr=1666479250
8 0.005767	127.0.0.1	127.0.0.1	TLSv1.3	99	Encrypted Extensions
9 0.005774	127.0.0.1	127.0.0.1	TCP	66	35640 → 11111 [ACK] Seq=512 Ack=162 Win=65408 Len=0 TSval=1666479253 TSecr=1666479253
10 0.006479	127.0.0.1	127.0.0.1	TLSv1.3	4896	Certificate
11 0.006493	127.0.0.1	127.0.0.1	TCP	66	35640 → 11111 [ACK] Seq=512 Ack=4992 Win=62592 Len=0 TSval=1666479253 TSecr=1666479253
12 0.018296	127.0.0.1	127.0.0.1	TLSv1.3	352	Certificate Verify
13 0.018383	127.0.0.1	127.0.0.1	TCP	66	35640 → 11111 [ACK] Seq=512 Ack=5278 Win=64768 Len=0 TSval=1666479265 TSecr=1666479265
14 0.018679	127.0.0.1	127.0.0.1	TLSv1.3	124	Finished
15 0.018685	127.0.0.1	127.0.0.1	TCP	66	35640 → 11111 [ACK] Seq=512 Ack=5336 Win=64768 Len=0 TSval=1666479266 TSecr=1666479266
16 0.021960	127.0.0.1	127.0.0.1	TLSv1.3	124	Finished
17 0.025615	127.0.0.1	127.0.0.1	TLSv1.3	102	Application Data
18 0.025662	127.0.0.1	127.0.0.1	TCP	66	11111 → 35640 [ACK] Seq=5336 Ack=606 Win=65536 Len=0 TSval=1666479273 TSecr=1666479269
19 0.025805	127.0.0.1	127.0.0.1	TLSv1.3	110	Application Data
20 0.025891	127.0.0.1	127.0.0.1	TLSv1.3	90	Alert (Level: Warning, Description: Close Notify)
21 0.026017	127.0.0.1	127.0.0.1	TCP	66	11111 → 35640 [FIN, ACK] Seq=5404 Ack=606 Win=65536 Len=0 TSval=1666479273 TSecr=1666479269
22 0.026753	127.0.0.1	127.0.0.1	TLSv1.3	90	Alert (Level: Warning, Description: Close Notify)
23 0.026785	127.0.0.1	127.0.0.1	TCP	54	11111 → 35640 [RST] Seq=5405 Win=0 Len=0

Figure 12 Packet Capture

Table 14 Wireshark shows the data captured by tcpdump as viewed in Wireshark. The data in the CKS Client Hello column shows the order of the bytes array representing the CKS code sent over to the server, where the first value will be the CKS being requested. The CKS Server Hello column displays the CKS code accepted by the server. The bytes map to the CKS where:

- 00 represents CKS *Default* (0)
- 01 represents CKS *Native* (1)

- 02 represents CKS *Alternate* (2)
- 03 represents CKS *Both* (3)

Table 14 Wireshark

Test Case ID	CKS Client Command line (X9.146 Enumeration)	CKS Client Hello (WolfSSL Enum.)	CKS Server Hello (WolfSSL Enum.)	Notes
1.1.0.1.0.1.2	*Default* single-signature certificate	Data: 000000	Data: N/A	Successful connection 1.1.0.1.0.1.2_cks0_6-9.pcap
1.1.1.1.3.1.2	use *Native* - ignore *Alternate*	Data: 010302	Data: 01	Successful connection 1.1.1.1.3.1.2_cks1_6-9.pcap
1.1.2.1.3.1.2	Use *Alternative* – Ignore *Native*	Data: 020301	Data: 02	Successful connection 1.1.2.1.3.1.2_cks4_6-9.pcap
1.1.3.1.3.1.2	Use *Both* – sign with both signatures	Data: 030102	Data: 03	Successful connection 1.1.3.1.3.1.2_cks5_6-9.pcap

Table 15 TLS Connections shows the time taken for the TLS connection for the four test cases.

Table 15 TLS Connections

Test case ID	Calculated in Wireshark (Finished message)	Calculated in Wireshark (client teardown)	@client – using time	@client – using perf stat -r 1 -B
1.1.0.1.0.1.2	0.018792s 18.792ms	0.020150s 20.150ms	0.02 user 0.01 system 0:00.04 elapsed	0.054641221s time elapsed 0.025463000s user 0.018535000s sys
1.1.1.1.3.1.2	0.026057s 26.057ms	0.031857s 31.857ms	0.02 user 0.01 system 0:00.06 elapsed	0.048904011s time elapsed 0.024033000s user 0.017489000s sys
1.1.2.1.3.1.2	0.013141s 13.141ms	0.015010s 15.010ms	0.01 user 0.02 system 0:00.04 elapsed	0.040464611s time elapsed 0.018987000s user 0.021572000s sys
1.1.3.1.3.1.2	0.029754s 29.754ms	0.036832s 36.832ms	0.02 user 0.01 system 0:00.05 elapsed	0.064142386s time elapsed 0.026666000s user 0.016162000s sys

Using the table, a graph is created to visualize the data points for each respective test case. The X-axis represents each CKS test case, starting from CKS 0 on the right to CKS 3 on the far left. The Y-axis represents the time, in seconds, it took to run the TLS connection. Note that for the "perf stat" and "time" data in the graph, the total elapsed time is used to plot the points – see Figure 13 TLS Connections.

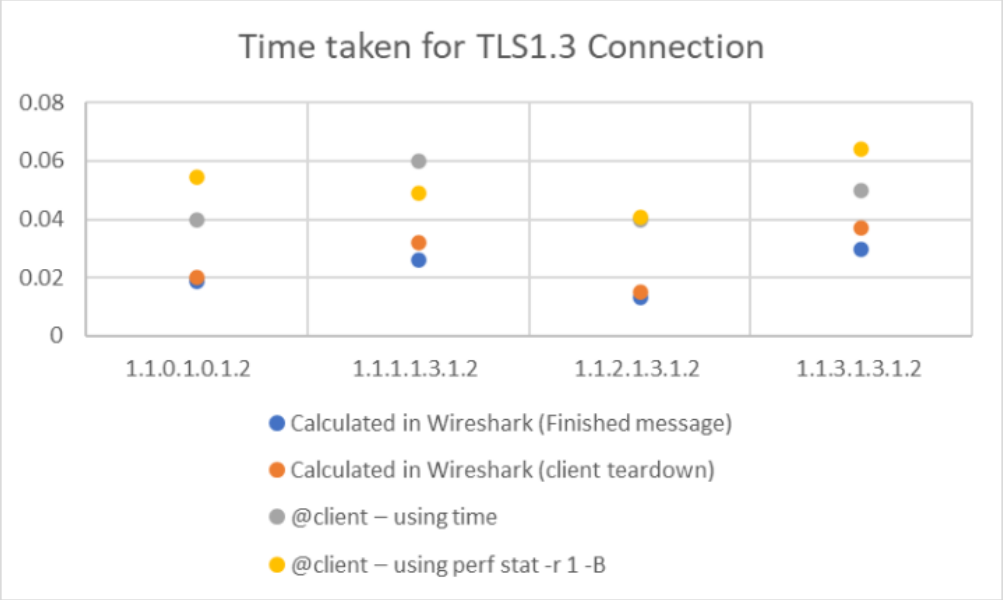


Figure 13 TLS Connections

Using the Classic certificate for CKS *Default* (0) as the baseline, data from the experiment indicate that the fastest performance came from CKS Alternate (2), while the slowest performance came from CKS *Both* (3).

Using the "finished message" times from Wireshark to calculate performance differences between CKS *Default* (0) and the others, CKS *Alternate* (2) came in first being 5.651 ms faster than CKS *Default* (0), then CKS *Native* (1) at 7.265 ms slower than CKS *Default* (0), and finally CKS *Both* (3) having the worst performance at 10.962 ms slower than CKS *Default* (0). Further analysis may be needed to investigate the differences in metrics.

Table 16 Performance shows the calculations of the CKS performance differences using CKS *Default* (0) as the baseline:

Table 16 Performance

CKS	Calculated in Wireshark (Finished message)	Time in Milliseconds	Delta (RSA)
			***Default* (0)**
Default (0)	0.018792	18.792	0
Native (1)	0.026057	26.057	7.265
Alternate (2)	0.013141	13.141	-5.651
Both (3)	0.029754	29.754	10.962

Keyfactor Lab Experiments

Keyfactor independently completed performance testing using Bouncy Castle and interoperability testing between Bouncy Castle and wolfSSL. Performance testing used ECDSA and ML-DSA algorithms and interoperability testing used RSA, ECDSA and ML-DSA. See section *Keyfactor Lab Environment* for details.

Performance

Each test case ran 10,000 times and the average time (in ms) for the TLS handshake was recorded and used to generate the graph shown in Figure 14 Bouncy Castle Performance. ECDSA curves p256, p384 and p521 were used with corresponding SHA 256, 384 and 512 hash algorithms.

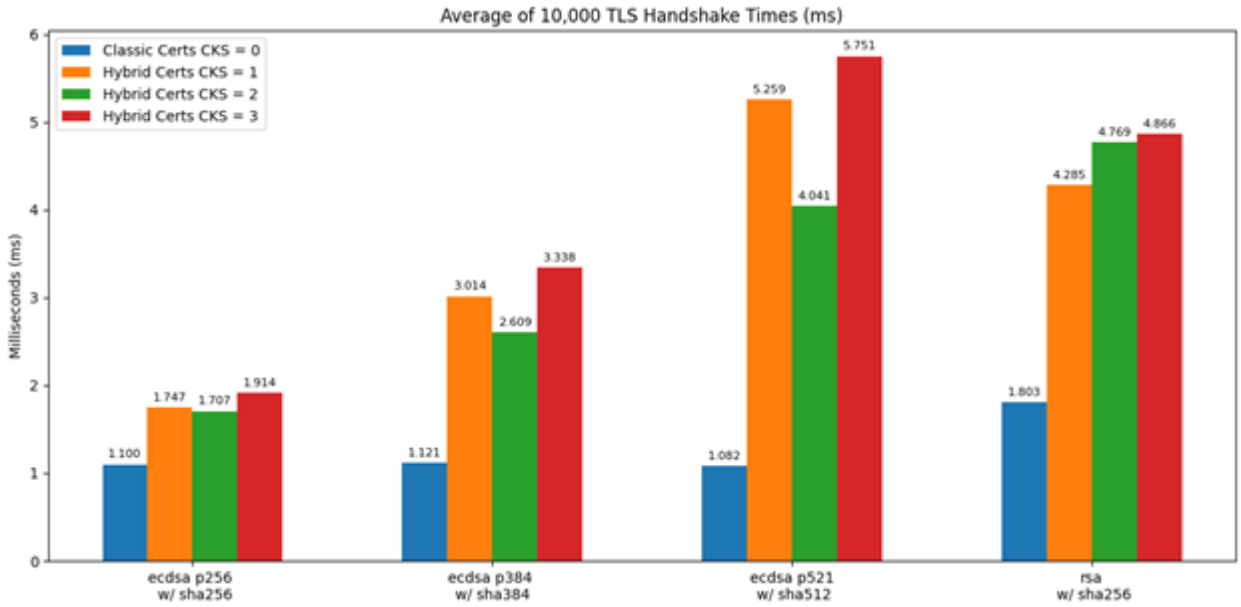


Figure 14 Bouncy Castle Performance

As expected, Classic certificates with *Native* fields perform faster than Chimera (hybrid) certificates with *Native* fields and *Alternate* extensions. Notable CKS *Native* (1) with Chimera (hybrid) certificates using the Native fields and ignoring the Alternate extensions is slower than CKS *Default* (0) with Classic certificates, even though they contain the same algorithm in their *Native* fields. An isolated graph showing Classic and Chimera certificates is shown in Figure 15 Bouncy Castle Hybrid Performance.

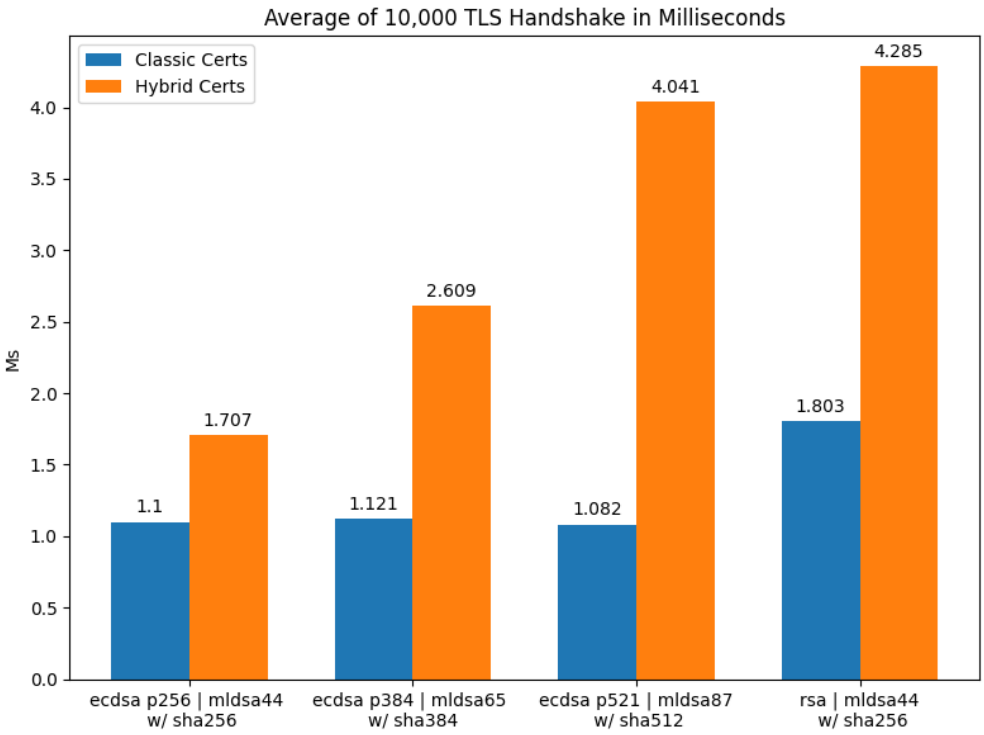


Figure 15 Bouncy Castle Hybrid Performance

Table 17 Bouncy Castle Benchmark contains the test cases selected to benchmark the TLS handshake.

Table 17 Bouncy Castle Benchmark

Test Case ID	Native Field	Alternative Field	X9.146 CKS
3.3.0.1.2.1.2	RSA-3072	N/A	0
3.3.1.1.2.1.2	RSA-3072	ML-DSA-44	1
3.3.2.1.2.1.2	RSA-3072	ML-DSA-44	2
3.3.3.1.1.1.2	RSA-3072	ML-DSA-44	3
3.3.0.3.1.1.2	ECDSA-P256	N/A	0
3.3.1.3.1.1.2	ECDSA-P256	ML-DSA-44	1
3.3.2.3.1.1.2	ECDSA-P256	ML-DSA-44	2
3.3.3.3.1.1.2	ECDSA-P256	ML-DSA-44	3
3.3.0.4.3.1.2	ECDSA-P383	N/A	0
3.3.1.4.3.1.2	ECDSA-P383	ML-DSA-65	1
3.3.2.4.3.1.2	ECDSA-P383	ML-DSA-65	2
3.3.3.4.3.1.2	ECDSA-P383	ML-DSA-65	3
3.3.0.5.4.1.2	ECDSA-P521	N/A	0
3.3.1.5.4.1.2	ECDSA-P521	ML-DSA-87	1
3.3.2.5.4.1.2	ECDSA-P521	ML-DSA-87	2
3.3.3.5.4.1.2	ECDSA-P521	ML-DSA-87	3

Interoperability

The following test cases were selected to demonstrate the interoperability with wolfSSL using the X9.146 Chimera certificates. The steps to reproduce the tests can be found on GitHub Table 18 Bouncy Castle Interoperability shows the permutation of tests performed. The following choices were added:

- *Native* is (3) ECDSA-P256 with SHA2-256, (4) ECDSA-P384 with SHA2-384, (5) ECDSA-P521 with SHA2-512

- *Alternative* is (2) ML-DSA-44 with SHA2-256, (3) MLDSA-65 with SHA2-384, (4) MLDSA-87 with SHA2-521

Table 18 Bouncy Castle Interoperability

Test Case ID	Native Field	Alternative Field	X9.146 CKS
3.1.0.1.2.1.2	RSA-3072	N/A	0
3.1.1.1.2.1.2	RSA-3072	ML-DSA-44	1
3.1.2.1.2.1.2	RSA-3072	ML-DSA-44	2
3.1.3.1.1.1.2	RSA-3072	ML-DSA-44	3
3.1.0.3.1.1.2	ECDSA-P256	N/A	0
3.1.1.3.1.1.2	ECDSA-P256	ML-DSA-44	1
3.1.2.3.1.1.2	ECDSA-P256	ML-DSA-44	2
3.1.3.3.1.1.2	ECDSA-P256	ML-DSA-44	3
3.1.0.4.3.1.2	ECDSA-P383	N/A	0
3.1.1.4.3.1.2	ECDSA-P383	ML-DSA-65	1
3.1.2.4.3.1.2	ECDSA-P383	ML-DSA-65	2
3.1.3.4.3.1.2	ECDSA-P383	ML-DSA-65	3
3.1.0.5.4.1.2	ECDSA-P521	N/A	0
3.1.1.5.4.1.2	ECDSA-P521	ML-DSA-87	1
3.1.2.5.4.1.2	ECDSA-P521	ML-DSA-87	2
3.1.3.5.4.1.2	ECDSA-P521	ML-DSA-87	3
1.3.0.1.2.1.2	RSA-3072	N/A	0
1.3.1.1.2.1.2	RSA-3072	ML-DSA-44	1
1.3.2.1.2.1.2	RSA-3072	ML-DSA-44	2
1.3.3.1.1.1.2	RSA-3072	ML-DSA-44	3
1.3.0.3.1.1.2	ECDSA-P256	N/A	0
1.3.1.3.1.1.2	ECDSA-P256	ML-DSA-44	1
1.3.2.3.1.1.2	ECDSA-P256	ML-DSA-44	2
1.3.3.3.1.1.2	ECDSA-P256	ML-DSA-44	3
1.3.0.4.3.1.2	ECDSA-P383	N/A	0
1.3.1.4.3.1.2	ECDSA-P383	ML-DSA-65	1
1.3.2.4.3.1.2	ECDSA-P383	ML-DSA-65	2
1.3.3.4.3.1.2	ECDSA-P383	ML-DSA-65	3
1.3.0.5.4.1.2	ECDSA-P521	N/A	0

Test Case ID	Native Field	Alternative Field	X9.146 CKS
1.3.1.5.4.1.2	ECDSA-P521	ML-DSA-87	1
1.3.2.5.4.1.2	ECDSA-P521	ML-DSA-87	2
1.3.3.5.4.1.2	ECDSA-P521	ML-DSA-87	3

Results

Wells Fargo experiments consisted of four test cases using a Classic certificate for CKS *Default* (0) with RSA-2048, a Chimera certificate for CKS *Native* (1), *Alternate* (2) and *Both* (3) with RSA-2048 and ML-DSA-44 and wolfSSL for both the TLS client and the TLS server.

Assumption 1: Larger certificates take longer to transmit and parse
Both CKS *Default* (0) and CKS *Native* (1) used RSA-2048, but CKS *Native* (1) is 38.6% slower than CKS *Default* (0). Since CKS *Default* (0) is a Classic certificate and CKS *Native* (1) is a Chimera certificate, the assumption is that the larger certificate took longer to transmit and possibly longer to parse.

Keyfactor experiments showed similar results, where CKS *Default* (0) is faster than CKS *Native* (1) time.

Assumption 2: ML-DSA-44 is faster than RSA-2048
CKS *Alternate* (2) used RSA-2048 and ML-DSA-44 with a Chimera certificate was 30% faster than CKS *Default* (0) used RSA-2048 with a Classic certificate, despite the differences in the certificate sizes. Thus, the assumption is that ML-DSA-44 is much faster than RSA-2048 algorithm which is not unexpected.

Keyfactor experiments showed dissimilar results, where ML-DSA is slower than ECDSA.

Assumption 3: Processing both RSA-2048 and ML-DSA-44 has resource conflict
CKS *Default* (0) used RSA-2048 with a Classic certificate, and CKS *Alternate* (2) used RSA-2048 and ML-DSA-44 a Chimera certificate. Hypothetically adding the CKS *Default* (0) time and the CKS *Alternate* (2) time should give an approximation for the CKS *Both* (3) time. However, CKS *Both* (3) is 93% slower than RSA-2048 and ML-DSA-44 which was a surprise. Another assumption is that wolfSSL incurred a resource conflict that resulted in slower performance.

Keyfactor experiments showed dissimilar results, where CKS *Default* (0) and CKS *Alternate* (2) times added together are on par with the CKS *Both* (3) times. Further analysis may be needed to investigate the differences in metrics.

Conclusion

This Technology Case Study supports the development of the draft X9.146 standard. Historically, an industry standard is developed to address an issue within a given industry segment with the hope that the target industry implements and adopts the standard. Conversely, this Technology Case Study demonstrates the feasibility of the X9.146 standard during its development.

The implementation of the Certificate Key Selection (CKS) extension to the TLS protocol in wolfSSL and Bouncy Castle demonstrates its feasibility. The beta versions for both products are available on GitHub.

The validation of the Certificate Key Selection (CKS) extension to the TLS protocol between wolfSSL and Bouncy Castle demonstrates its interoperability.

Regardless of the results and assumptions, the CKS testing was successful, indicating that that the CKS extension to the TLS protocol is valid and worth implementing. The X9F5 Financial PKI[12] workgroup will continue developing the X9.146 standard.

[12] X9F5 https://x9.org/x9-boardsubcommittees/

This concludes the Technology Case Study, but the X9F5 work continues. Organizations that are X9 or X9F members can join the X9F5 work on X9.146 efforts and others who wish to participate can join ASC X9 Financial Services.

References

[1] J. Stapleton and R. Poore, "Cryptographic Transitions," 2006 IEEE Region 5 Conference, San Antonio, TX, USA, 2006, pp. 22-30, doi: 10.1109/TPSD.2006.5507465. https://ieeexplore.ieee.org/document/5507465

[2] NIST Post-Quantum Cryptography (PQC) Project https://csrc.nist.gov/projects/post-quantum-cryptography

[3] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," Proceedings 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 1994, pp. 124-134, doi: 10.1109/SFCS.1994.365700. https://ieeexplore.ieee.org/document/365700

[4] RFC 8446 The Transport Layer Security (TLS) Protocol Version 1.3, August 2018 https://datatracker.ietf.org/doc/rfc8446/

[5] ITU-T X.509 Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks, October 2019 https://www.itu.int/rec/T-REC-X.509/en

[6] Draft X9.146 Public Key Infrastructure (PKI) Certificate Key Selection (CKS) for Transport Layer Security (TLS) https://x9.org/

[7] IETF RFC 7250 Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) – see https://datatracker.ietf.org/doc/rfc7250/

[8] IETF RFC 8422 Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier, August 2018 – see https://datatracker.ietf.org/doc/rfc8422/

[9] Multiple Public-Key Algorithm X.509 Certificates https://datatracker.ietf.org/doc/draft-truskovsky-lamps-pq-hybrid-x509/

[10] Composite Public and Private Keys For Use In Internet PKI https://datatracker.ietf.org/doc/draft-ounsworth-pq-composite-keys/

[11] A Mechanism for Encoding Differences in Paired Certificates https://datatracker.ietf.org/doc/draft-bonnell-lamps-chameleon-certs/

[12] FIPS 203 Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) https://csrc.nist.gov/pubs/fips/203/final

[13] wolfSSL Open Source Cybersecurity https://github.com/wolfSSL/

[14] Bouncy Castle https://github.com/bcgit/bc-java

[15] ML-KEM Post-Quantum Key Agreement for TLS 1.3 https://datatracker.ietf.org/doc/draft-connolly-tls-mlkem-key-agreement/

