Draft Document for Review October 27, 2025 1:54 pm SG24-8525-01

Redbooks
ibm.com/redbooks

# Transitioning to Quantum-Safe Cryptography on IBM Z

Bill White
Gregg Arquero
Ritu Bajaj
Anne Dames
Richard Kisley
Henrik Lyksborg
Charu Tejwani
Navya Ramanjulu

Security

IBM Z

IBM

Redbooks

IBM Redbooks

# Transitioning to Quantum-Safe Cryptography on IBM Z

October 2025

**Note:** Before using this information and the product it supports, read the information in “Notices” on page vii.

**Second Edition (October 2025)**

This edition applies to the quantum-safe standardized algorithms and the capabilities available with the IBM z17, IBM z16, and IBM z15.

This document was created or updated on October 24, 2025.

# Contents



# Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
*IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

# Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

| | | |
|---|---|---|
| DB2® | IBM Z® | Think® |
| Db2® | PIN® | z/OS® |
| IBM® | RACF® | z/VM® |
| IBM Research® | Redbooks® | z15™ |
| IBM Security® | Redbooks (logo)  ® | z16™ |

The following terms are trademarks of other companies:

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Other company, product, or service names may be trademarks or service marks of others.

# Preface

As cyberattacks continue to increase, the cost and reputation impacts of data breaches remain a top concern across all enterprises. Even if sensitive data is encrypted and is of no use now, cybercriminals are harvesting that data because they might gain access to a quantum computer that can break classical cryptographic algorithms sometime in the future. Therefore, organizations must start protecting their sensitive data today by using quantum-safe cryptography.

This IBM® Redbooks® publication reviews some potential threats to classical cryptography by way of quantum computers and how to make best use of today's quantum-safe capabilities on the IBM Z platform. This book also provides guidance about how to get started on a quantum-safe journey and step-by-step examples for deploying IBM Z® quantum-safe capabilities.

This publication is intended for IT managers, IT architects, system programmers, security administrators, and anyone who needs to plan for, deploy, and manage quantum-safe cryptography on the IBM Z platform. The reader is expected to have a basic understanding of IBM Z security concepts.

## Authors

This book was produced by a team of specialists from around the world working at IBM Redbooks, Poughkeepsie Center.

**Bill White** is an IBM Redbooks Project Leader and Senior IT Infrastructure Specialist at IBM Poughkeepsie, New York.

**Gregg Arquero** is a Senior Software Engineer at IBM. He joined IBM in 2015 working with the IBM z/OS® ICSF team. During his time on the ICSF team, he designed and developed several key crypto solutions on IBM Z, such as early ICSF availability at IPL, AES-DUKPT support, and quantum-safe algorithm support. He is also an avid innovator with over a dozen granted patents by the USPTO. He received his bachelor's degree in Computer Science from Binghamton University.

**Ritu Bajaj** is a Senior Design Researcher at IBM Z. As an empathetic designer and business professional, she uses human-centered design and user experience (UX) research to create value-add strategies for product innovation and sustainable business growth. She joined IBM in 2020 and collaborated with the IBM Z Security Hardware teams. She led the sponsor user program and client engagements for quantum-safe cryptography to gather feedback and synthesize clients' insights for starting IBM z16™ and enhancing ADDI with cryptographic discovery. She conducted UX heuristic evaluations for the Fully Homomorphic Encryption (FHE) toolkit and the IBM Z Security and Compliance Center dashboard. With dual degrees in M.Design from Illinois Institute of Technology and an executive MBA from Michigan State University, she applies her diverse knowledge in strategic and design thinking methodologies to solve customers' problems and explore use cases.

**Anne Dames** is a Distinguished Engineer in the IBM Z Cryptographic Technology area. She received a B.S. degree in Mathematics from Johnson C. Smith and a M.S. degree in Computer Science from the University of North Carolina at Charlotte. She has held various positions in firmware development, software, and product engineering. She has years of

experience as a development leader for the IBM Hardware Security Module product line and the IBM Common Cryptographic Architecture (CCA). She is currently leading a cross platform development effort to use quantum-safe cryptography.

**Richard Kisley** is a Senior Technical Staff Member in Crypto Card Development and payment security standards, at the Durham, North Carolina development lab in the United States. He has 20 years of experience in the Hardware Security Module field. He holds a Masters degree in Computer Science from Duke University. His areas of expertise include software development, cryptographic modules and payment security standards with the Payment Card Industry Security Standards Council (PCI SSC), X9.org, and the International Organization for Standardization (ISO).

**Henrik Lyksborg** is a Senior IT Specialist working in the IBM Crypto Competence Center based in Copenhagen, Denmark. He is a product owner for cryptographic appliances and APIs, such as IBM Enterprise Key Management Foundation (EKMF), IBM Advanced Cryptographic Service Provider (ACSP), and IBM Crypto Analytics Tool (CAT). Henrik studied Electrical Engineering at the Technical University of Denmark. He joined IBM in 1991 with a B.Sc. in Electrical Engineering and worked with IBM mainframe software licensing and client software maintenance delivery processes. In 2017, he took on an IBM internet payment platform job, dealing with Payment Card Industry (PCI) controlled business applications. Currently, Henrik works with cryptographic key management, cryptographic services, and cryptographic compliance controls to ensure that keys, certificates, and cryptographic material are well managed and documented.

**Charu Tejwani** is a Senior Product Manager in the IBM Z Security Product Management group. She has been with IBM for 16 years with over half of those years spent in Product Management in IBM Z, driving several solutions to market. Charu has experience with wide range of security capabilities on the IBM Z platform but she particularly feels passionate about cryptography. She has dedicated past two years understanding the Quantum Cryptography threat and how best to serve IBM Z clients with new solutions to help them prepare for the cyber attacks of the future. Charu holds a MS degree in Electrical Engineering from University of South Florida.

**Navya Ramanjulu** is a Senior Software Engineer in the IBM Enterprise Networking Software group, IBM Z. She received a Master's degree in Computer Science from North Carolina State University. Navya is the security architect for the z/OS Communications Server. Her areas of expertise include z/OS Encryption Readiness Technology, AT-TLS, z/OS Container Platform, z/OS FTP and System Resolver. Navya has 11 years of experience in the z/OS domain.

Thanks to the following people for their contributions to this project:

Didier Andre
IBM Security Expert & Delivery Consultant

Jacob Ruwald
IBM Crypto Competence Center

Garry Sullivan
IBM Trusted Key Entry Architect

A special thanks to the authors of the first edition of this book for their ground breaking work.

Didier Andre, Gregg Arquero, Ritu Bajaj, Joe Cronin, Anne Dames, Henrik Lyksborg, Alexandra Miranda, and Maxwell Weiss

# Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an IBM Redbooks residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

**ibm.com**/redbooks/residencies.html

# Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  **ibm.com**/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, IBM Redbooks
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

# Stay connected to IBM Redbooks

- Find us on LinkedIn:

  https://www.linkedin.com/groups/2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/subscribe

- Stay current on recent Redbooks publications with RSS Feeds:

  https://www.redbooks.ibm.com/rss.html

1

# Cryptography in the quantum computing era

For decades, organizations used cryptographic algorithms to protect their most sensitive data and communications in computer systems, networks, and storage devices. Imagine if an adversary or cybercriminal can break those cryptographic algorithms that you relied on for many years. What are the potential impacts on your business?

Once considered impossible, attacks that can compromise today's cryptographic algorithms can become possible with a powerful quantum computer. Your protected data can be stolen, exposed, altered, disabled, or destroyed through new attack vectors that increased in the quantum computing era.

Although quantum computers are still in their early stages of adoption, their use soon will be more widespread. A single quantum computer can be capable of performing millions of computations simultaneously.

Because quantum computers deal with probabilities, the problems they are good at solving are exponential in nature. That is, today's cryptographic algorithms might be threatened by quantum computers, potentially exposing sensitive data.

Attackers are already harvesting protected data in anticipation of cracking the protection algorithms sometime in the future. Therefore, it is important to take action now: assess the cryptography methods that are used today to protect your data, applications, and systems; understand the vulnerabilities in the quantum computing era; and evaluate the quantum-safe capabilities that are offered with the IBM Z platform.

This chapter includes the following topics:



## 1.1 When will quantum computers break cryptography

The question of when quantum computers might break cryptography is often asked, but unfortunately presents the threat as being in the future. Consider the following point from the World Economic Forum[1]:

> *For data that will require protection for decades, the threat is today. The impact is in the future.*

Many IT decision-makers plan to retain their data 11+ years into the quantum era as per an IBM MD&I survey. Their data includes personal identifiable information (PII), trade secrets, intellectual property, and other sensitive digital assets. This information already is at risk because they need to store it and keep it confidential for decades.

Although cybercriminals cannot easily break most encrypted data today, they might be able to decrypt that data in the future by using a large quantum computer, also known as a *cryptographically relevant quantum computer* (CRQC)[2]. Because it is unknown when Y2Q (Year to Quantum) will happen, it is best to start looking at ways to protect your data now:

> *Act now—it will be less expensive, less disruptive, and less likely to have mistakes caused by rushing and scrambling.*[3]

### 1.1.1 Business risks

The cost of data breaches continues to increase. As per the *Cost of a Data Breach Report*, the global average cost of has jumped to $4.88 million in 2024, a 10% spike and the highest increase since 2020.

Businesses have expressed shared concerns for data, system integrity, and software verification at risk with the potential threats of CRQC attacks and it is reflected in the increased security investment, which is up by 12% in 2024 over the previous year.

Additionally, many organizations operate within highly regulated industries and must comply with the latest standards or pay hefty fines. BSI, a German federal agency, requires the use of hybrid schemes in which both classical and quantum-safe algorithms are used for protection in high-security applications.

IT executives can proactively mitigate the risk of business disruptions that are caused by CRQC attacks. Currently, some organizations have decentralized small-size cryptography groups that are scattered across the IT business units. This causes disconnection and inertia against cryptographic agility[4]. Many IT organizations still do not have a comprehensive view of the cryptography in use because they lack cryptographic inventory tools and skills for the broad cryptography landscape.

A few cryptographic services groups have implemented proprietary cryptography libraries and abstraction layers that simplifies managing the updates and prepare them for cryptographic agility. Businesses, such as banks, telcos, and automotive manufacturers have expressed concerns about the effect of quantum-safe algorithms on system performance and latency.

---

[1] Reference *Is your cybersecurity ready to take the quantum leap?* (2021)
[2] CRQC is used to specifically describe quantum computers that are capable of breaking cryptographic algorithms used on conventional computers.
[3] Reference *NIST PQC Standardization Update-Round 2 and Beyond* (2020)
[4] Cryptographic agility is about an information security system rapidly switching to alternative cryptographic primitives and algorithms without making significant changes to the system's infrastructure.

Organizations, such as automobile manufacturers, must deal with programming resource constraints, such as key length size and RAM, to accommodate the use of public key quantum-safe algorithms and schemes in their new application development process.

Organizations also are interested in quantum-safe encryption to protect their long-lasting sensitive data and future-proof it. Quantum-safe readiness is on most organizations' roadmap. Some have already benchmarked quantum-safe algorithms and are exploring cryptographic inventory tools for application modernization.

### 1.1.2 Quantum threats and implications on data and identity

What will a cybercriminal be able to do with a quantum computer? Why do organizations need to act now? Why is the data at risk[5] today?

A cybercriminal with access to a sufficiently strong quantum computer can break the current cryptography. Here are some examples of the threats[6] and implications:

- Passive attacks on confidentiality

  Cybercriminals might harvest data communications, recover session keys from encrypted channel negotiation, and decrypt communication transmissions. They can steal snapshots of encrypted cloud data, extract keys that are protected by using public keys, and conduct retrospective decryption.

  Cybercriminals might decrypt lost or harvest historical data through cracking encryption keys. An organization's sensitive data that is protected by using today's cryptography might be vulnerable in the future. Encrypted data that is stolen during a data breach and encrypted media that is improperly disposed or stolen are both at risk.

- Impersonation attacks on identities

  Cybercriminals might create fraudulent code updates, insert malware, change configuration settings, and create damage. They might transfer assets on a blockchain or manipulate updates and forge transactions through fraudulent authentication. With quantum threats, identity over the internet and software authenticity cannot be guaranteed.

  Cybercriminals might impersonate a remote system or user and authenticate access, and control systems. They can remotely control critical business infrastructure or transport infrastructure. Systems that organizations are building today are at risk.

- Manipulate legal history by forging digital signatures

  Cybercriminals might carry out fraudulent authentication by deriving private keys from public keys. The legal underpinnings of digitalization are vulnerable because documents can be forged by using a derived private key. Also, a guarantee of proof of authorship or integrity no longer exists.

[5] A measure of the extent to which an entity is threatened by a potential circumstance or event, and typically a function of the adverse impacts that can arise if the circumstance or event occurs; and the likelihood of occurrence.

[6] Any circumstance or event with the potential to adversely impact organizational operations, organizational assets, individuals, other organizations, or the Nation through a system by way of unauthorized access, destruction, disclosure, modification of information, or denial of service.

## 1.2 Why are quantum computers a threat

Cryptographic algorithms are based on mathematics, and with enough time and computing resources they can be broken. Improvements in cryptographic algorithms always were needed over time, as computers and digital circuits increase in speed and capacity.

However, the nature of the mathematical algorithms that can be used on quantum computers is fundamentally different from what can run on conventional computers. Unfortunately for cryptography, specific algorithms that run on quantum computers can be efficient at breaking some current cryptographic algorithms.

To help understand how quantum computers are relevant to breaking current cryptography, we first need to recognize the different types of computers and how they work. These computers are sorted into the following categories:

- **Conventional computers**

  These computers are the computers that we use every day. Their circuits operate on binary values (bits) that can have only two states: a zero (off) or a one (on). Algorithms are implemented as sequences of computer instructions that operate on these binary values.

- **Supercomputers**

  A supercomputer is essentially a large and tightly coupled set of conventional computers, with high-speed communications between them. They reduced or offloaded input/output (I/O) routines by design to free up CPU cycles. Supercomputers are often used to solve problems that can be deconstructed into many separate computations, which are carried out in parallel on their computing nodes.

- **Quantum computers**

  Quantum computers process data by using an entirely different mechanism than conventional computers and supercomputers. Rather than representing data as binary values (bits) that can have only two states, the property of superposition conceptually lets quantum computers have an exponentially large number of possible compute states as more of their quantum bits (or qubits) are entangled[7]. Hence, the more qubits a quantum computer has available, the faster it can crack cryptographic algorithms.

  The computational power of quantum computers is growing rapidly. In 2021, IBM launched the 127 qubit Quantum Eagle processor with novel packaging and controls. In 2023, IBM debuted the 1,121 qubit Quantum Condor processor to explore potential Quantum Advantages–problems that can be solved more efficiently on a quantum computer than on the world's best supercomputers.

  IBM has a quantum technology roadmap with ambitious targets, progressing to one billion gates across 2,000 qubits by 2033 with the future Blue Jay processor.

### 1.2.1 Cryptography overview

Various methods were used for thousands of years to protect information when it is stored or sent to other people. Early methods were simple, like the Caesar Cipher, but they increased dramatically over time, particularly as the attackers improved their ability to break the codes. The fundamental feature of all cryptographic algorithms is the use of functions that are easy to compute if you know the cryptographic key, but difficult if you do not know the key.

[7] Quantum entanglement allows qubits to be perfectly correlated with each other. Using quantum algorithms that exploit quantum entanglement, specific complex problems can be solved more efficiently than on classical computers. For the technical definition, see: https://quantum-computing.ibm.com/composer/docs/iqx/terms-glossary#term-entanglement

The cryptographic algorithms are used for the following types of protection:

- **Confidentiality**

  This process keeps data secret from people who are not authorized to see it. The unencrypted data is called *plain text*, and the encrypted data is called *ciphertext*.

- **Integrity**

  This ability is used to prove that data was not modified.

- **Authentication**

  This ability is used to prove who someone is, or who created a piece of data.

- **Nonrepudiation**

  This ability is used to prevent someone from claiming they did not create a particular specific piece of data.

The cryptographic algorithms fall into the following categories, which are described next:

- Symmetric cryptography
- Asymmetric cryptography
- Hashing algorithms

## Symmetric cryptography

Symmetric cryptography is used to encrypt and decrypt data. It is called *symmetric* because the same key is used for encryption and decryption. Symmetric algorithms are generally fast, and are used for everything from encrypting communications links to protecting banking transactions.

In addition to encryption of data, the symmetric algorithms are used to construct methods of providing integrity, authentication, and other operations that are important to security. For integrity, these functions are called *Message Authentication Codes* (MACs).

The following symmetric cryptographic algorithms are most commonly used today:

- Triple-DES (TDES, 3DES, or TDEA)

  TDES is an older algorithm, which is gradually being phased out and replaced with the newer and stronger AES. TDES uses a key that is 112 bits or 192 bits long, and encrypts data in 64-bit blocks.

- Advanced Encryption Standard (AES)

  AES use keys that are 128 bits, 192 bits, or 256 bits, and it encrypts data in 128-bit blocks. The larger key lengths and encryption block sizes make AES stronger than TDES. AES also eliminates some design issues in TDES that make TDES susceptible to specific classes of attacks.

The symmetric algorithms use complex mathematical and logical operations to combine the data and the key in such a way that the ciphertext appears to be random values. With a strong algorithm, the ciphertext cannot be examined and anything about the plain text or the key cannot be determined.

Therefore, the only way to break the algorithm is to try all possible keys until you find the one that works. On the average, this effort often means trying half of the possible keys. For example, with AES using a 256-bit key, you must try an average of half of the $2^{256}$ possible keys, which is a huge number.

## Asymmetric cryptography

In asymmetric cryptography, which is also known as *public key cryptography*, two keys are used in combination. This configuration contrasts with symmetric key cryptography, where the same key is used for all operations. The asymmetric keys come in pairs that are known as the *public key* and *private key*, which are mathematically related.

As the names imply, the public key can be seen by anyone, while the private key is kept secret. The owner of the key generates the public and private keys together; then, it keeps the private key secret while distributing the public key to anyone who needs it. It is impossible to determine the value of the private key from the public key.

Several asymmetric cryptographic algorithms are commonly used. The two most common are Elliptic Curve Cryptography (ECC), and RSA, which is named for its inventors Rivest, Shamir, and Adleman. Unlike symmetric algorithms, differences exist in what the distinct asymmetric algorithms can do.

ECC is based on the mathematics of elliptic curves. The curves are defined by polynomials, and the ECC algorithm is based on multiplication of points on the curve. When a point is multiplied by itself, the result is another point on the curve. When this multiplication occurs many times, it is difficult to look at the final point that results from the multiplications and determine anything about the original point.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is used to compute and verify digital signatures by using ECC mathematics.

The Elliptic Curve Diffie Hellman (ECDH) algorithm is used to negotiate shared symmetric encryption keys between two parties. It is notable that mathematics of ECC do not provide a way to encrypt and decrypt data. This creation is possible only by using ECC if you first create a shared encryption key by using ECDH or a similar method and then, encrypt the data by using that shared key with a symmetric algorithm, such as AES.

The security of RSA is based on the difficulty of factoring large numbers. The public key and private key are each consist of a modulus and an exponent, where the modulus is the same for each, but the public exponent and private exponent are different. The modulus is the product of two large prime numbers, and security is based on the fact that it is infeasible to factor the modulus to find those two large primes.

RSA encryption and decryption are based on modular exponentiation, where the value to be encrypted is raised to the public or private exponent, but that computation is done by using modular arithmetic that constrains the result to be less than the value of the modulus.

Whenever a value is raised to an exponent and then truncated according to the modulus, information is lost, which makes the process difficult to reverse. RSA can be used to directly encrypt data, and it is used for digital signatures by encrypting a hash of the data you want to sign. It is also frequently used to encrypt keys to transport them to other parties.

## Hashing algorithms

A hash algorithm does not “encrypt” data; instead, it creates a fixed-length digital “fingerprint” (called a *hash*) from input data of any length. If even one bit of the input data is modified, the computed hash is entirely different.

Cryptographic hash functions meet two criteria: First, if you know the hash value, you cannot use it to learn anything about the content of the data that was hashed. Secondly, it is infeasible to find a different set of data that produces the same hash value.

Recommended hash functions today are the SHA-2 and SHA-3 families, which offer versions that create hashes 224 bits - 512 bits. The older hash functions SHA-1 and MD5 are no longer considered secure, although they are still in use in some applications.

## 1.3 Impact of Shor’s and Grover’s algorithms

When available, a sufficiently strong quantum computer can perform specific mathematical computations exponentially faster than a conventional computer or supercomputer. The most powerful conventional computer can take millions of years to solve the integer factorization problem to find prime factors for a 2048-bit composite integer.

The use of a quantum computer with Shor’s and Grover’s algorithms can break or weaken some current cryptographic algorithms. Shor's and Grover's are cryptanalysis algorithms when run on quantum computers.

Asymmetric algorithms derive security strength from one of three complex mathematical problems:

- Integer factorization
- Discrete logarithm
- Elliptic curve discrete logarithm

Examples of asymmetric algorithms and protocols are RSA, ECC, DH, ECDH, and ECDSA. Consider RSA, which derives its strength from the difficulty in solving the integer factorization problem. It is easy to multiply primes but difficult to take a composite integer and reduce it back to the prime factors. The difficulty in factoring rises exponentially (not linearly) as the number of bits in the key increases. The typical RSA key is 2048 bits. It is not possible with today’s conventional computers to factor an integer with 2048 bits.

A sufficiently strong quantum computer can solve the factoring problems within hours with Shor's algorithm because it provides an exponentially faster method for solving integer factorization, discrete logarithm, and elliptic curve discrete logarithm problems.

Shor’s algorithm has the potential to completely break the RSA and Diffie-Hellman crypto systems and their elliptic curve-based analogs, but it cannot be used to attack symmetric encryption or hashing algorithms. Therefore, asymmetric crypto algorithms are most vulnerable to compromise.

Armed with Shor’s algorithm, an adversary or cybercriminal can take a public key and derive the private key to enable impersonation and fraud attacks. Therefore, we need new algorithms that are based on different math problems for conventional computers to address a CRQC attack by using Shor's algorithm.

Symmetric algorithms derive security strength from the difficulty in mounting a brute force attack or exhaustive search exploration of all possible inputs to find the answer. For cryptography, this trial-and-error technique is used to guess the correct value or key.

Examples of symmetric or hashing algorithms include AES, TDES, SHA-2, and CMAC. Brute force attacks on symmetric and hashing algorithms take a long time to search the message digest or key space to find the message digest that maps to data or correct encryption key. For example, when found, the correct key can be used to decrypt encrypted data. For a key with 256 bits, $2^{256}$ options exist to try in a worst case scenario.

A quantum computer can weaken the symmetric algorithm strength significantly using Grover’s algorithm. Grover's algorithm is a quantum algorithm for so-called unstructured

search problems that offers a quadratic improvement over classical algorithms. What this means is that Grover's algorithm requires a number of operations on the order of the square-root of the number of operations required to solve unstructured search classically.

Grover's algorithm does not break all symmetric algorithms, but it can be used to speed up a brute force search for symmetric keys or reverse engineer a cryptographic hash. The risks to symmetric and hashing algorithms can be mitigated by switching algorithms or increasing key or hashing digest sizes because Grover's algorithm is ineffective if the search space is too large.

Grover's quantum algorithm can affect hash-based password systems because only a few passwords must be searched, and the low security level of TDEA and SHA-1 means they are both at risk.

To learn more about Grover's algorithm, refer to:
https://learning.quantum.ibm.com/course/fundamentals-of-quantum-algorithms/grovers-algorithm

## 1.4 Cryptographic vulnerabilities possible with quantum computers

All of today's approved cryptographic algorithms are strongly secure against conventional computers, including supercomputers. For example, consider AES with a 128-bit key. On the average, it takes $2^{127}$ guesses to find the right key. If we assume that a conventional computer can try one key every microsecond, it takes about $5.4 \times 10^{24}$ years to find the key, which is not feasible. Even the fastest supercomputers can reduce this time only slightly.

However, the problem with quantum computers is that they do not have to take this approach for some of today's algorithms. In particular, the asymmetric algorithms can be broken almost instantaneously by using Shor's algorithm, even for the longest keys in use.

The advent of quantum computers makes it possible to attack algorithms by using methods that did not exist when attackers used conventional computers. Shor's algorithm with a sufficiently large quantum computer can easily break RSA or ECC algorithms. For this reason, new asymmetric algorithms are being developed that use different mathematical principles that are not subject to attack with Shor's algorithm or any other known process on quantum computers.

The risk to symmetric and hashing algorithms is significantly lower. Shor's algorithm cannot be used against these, but another algorithm that runs on quantum computers that can reduce their security.

Grover's algorithm can be used to reduce search times, and it can be used to improve brute force attacks to find a cryptographic key. When searching for something in a space of N total items, Grover's algorithm reduces the effort to $\sqrt{N}$. For example, a AES 256-bit key can be found with difficulty of only $2^{128}$. However, this key is still considered unbreakable, and NIST and other organizations believe that AES, SHA-2, and SHA-3 provide entirely adequate security in the age of quantum computers.

Table 1-1 lists the security effect of various algorithms and protocols when a sufficiently strong quantum computer is available.[8]

*Table 1-1 Effect of quantum computing on cryptographic schemes*

| Cryptographic algorithm | Type | Purpose | Quantum computer impact |
|---|---|---|---|
| AES | Symmetric key | Encryption | Secure |
| SHA-256, SHA-3 | Hash algorithm | Hash functions | Secure |
| RSA | Public key | Signatures, key establishment | Broken |
| ECDSA, ECDH (Elliptical Curve Cryptography) | Public key | Signatures, key exchange | Broken |
| DSA (Finite Field Cryptography) | Public key | Signatures, key exchange | Broken |

Organizations must consider integrating quantum-safe protection into their digital transformation strategy and application modernization plans to mitigate these two vulnerabilities with current cryptography. Consider the following points:

- Public key algorithms are broken by a large-scale quantum computer by using Shor's algorithm. Organizations can mitigate this vulnerability by migrating to quantum-safe algorithms and schemes.
- Symmetric key and hashing algorithms are affected by a large-scale quantum computer. Grover's algorithm reduces the security strengths of symmetric and hashing algorithms. Organizations can mitigate this vulnerability by increasing the key or message digest sizes.

Secure processes rely on protocols that employ public key cryptography, including those protocols that are used to secure websites for banking transactions, secure email, and signing software. It will take 5 - 15 or more years[9] to replace most public key crypto systems that are used now.

## 1.5 Algorithms to counter CRQC attacks

As data value grows and the required protection increases exponentially, a sense of urgency exists to protect long-lasting data from potential CRQC attacks. Organizations must safeguard data today with new cryptographic algorithms that protect against potential future CRQC attacks that might affect system integrity and core business infrastructures.

The National Institute of Standards and Technology (NIST) has published the finalized standards for three quantum-safe algorithms that would thwart attacks from both conventional and quantum computers.

But what makes these algorithms quantum-safe? Algorithms are based on mathematical problems with no known quantum computer speedup. Five categories of cryptographic schemes are believed to be quantum-safe (see Table 1-2). Current quantum-safe algorithms are based on some of these schemes.

[8] See *The Impact of Quantum Computing on Present Cryptography*
[9] Refer to *Getting Ready for Post-Quantum Cryptography*, then search for "*5 to 15 or more years*"

*Table 1-2 Categories and examples of quantum-safe algorithm candidates*

| Category | Description |
|---|---|
| Lattice-based crypto | Crypto schemes from a field of mathematics that is called the geometry of numbers. The security of these schemes is based on the difficulty of solving mathematical problems over lattices; for example, the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP), such as Examples include FIPS 203 - ML-KEM (formerly know as CRYSTALS-Kyber) and FIPS 204 - ML-DSA (formerly known as CRYSTALS-Dilithium), Falcon. |
| Multi-variate crypto | A group of crypto systems that is based on the difficulty of solving nonlinear (usually quadratic) equations over finite fields. The idea is that solving systems of equations in many variables is difficult under constraints depending on the scheme. Examples include Unbalanced Oil and Vinegar, MAYO. |
| Code-based crypto | This cryptography uses error-correcting codes to build public key cryptography; for example, Classic McEliece. |
| Hash-based crypto | This cryptography includes digital signature schemes that are based on cryptographic hashes; for example, SPHINCS+. |
| Isogeny crypto | Super singular elliptic-curve isogeny cryptography is based on the isogenies or mappings between two elliptic curves; for example: SQIsign |

**Note:** Quantum-safe algorithms run on conventional computers to protect data; the algorithms that can break some conventional cryptography run on quantum computers. That is, quantum-safe algorithms do not run on quantum computers; instead, they run on conventional computers.

## 1.5.1 Quantum-safe algorithms

Quantum-safe cryptographic algorithms are designed to safeguard against attacks from conventional and quantum computers. "*These finalized standards include instructions for incorporating them into products and encryption systems,*" said NIST mathematician Dustin Moody, who heads the PQC standardization project.

The standardized algorithms are in the following two categories:

- Digital signature algorithms
- Key encapsulation mechanisms and key-establishment algorithms

### Standardized algorithms

The standardized algorithms include the following:

- **FIPS 203 - ML-KEM**

  Formerly known as CRYSTALS-Kyber, FIPS 203 - ML-KEM (Module-Lattice-Based Key-Encapsulation Mechanism) is the primary algorithms NIST recommends to be implemented for most use cases around public-key encryption and key encapsulation mechanism (KEM). Among its advantages are comparatively small encryption keys that two parties can exchange easily, as well as its speed of operation.

- **FIPS 204 - ML-DSA**

  Module-Lattice-Based Digital Signature (ML-DSA) is the primary quantum-safe digital signature algorithm standard that should be used for use case around authentication and digital signatures. It was formerly known as CRYSTALS-Dilithium.

- **FIPS 205 - SLH-DSA**

  Formerly known as SPHINCS+, Stateless Hash-Based Digital Signature Standard (SLH-DSA) is a hash based digital signature standard for authentication. This algorithm uses a different math approach than ML-DSA, and it is intended as a backup method in case ML-DSA proves vulnerable.

  SPHINCS+ is to be standardized to avoid only relying on the security of lattices for signatures.

Both FIPS 203 - ML-KEM (key-establishment) and FIPS 204 - ML-DSA (digital signatures) were selected for their strong security and excellent performance, and NIST expects them to work well in most applications. The security of these two algorithms is based on the difficulty of solving the learning-with-errors (LWE) problem over module lattices. The LWE problem involves solving a system of linear equations, where an error of ±1 was intentionally introduced. Because of the errors, the usual methods of solving a system of linear equations do not work, which makes it infeasible to solve for the secret value.

Besides these three final PQC standards, NIST is currently working on the fourth draft standard based on FALCON. FALCON will be applied for use cases that are too large for ML-DSA signature.

IBM Research® scientists were involved in the development of ML-KEM, ML-DSA, and Falcon. They have also made contributions to the development of SPHINCS+. IBM has already implemented the draft versions of the quantum-safe algorithms that have been selected for standardization by NIST, now known as: ML-DSA for digital signatures and ML-KEM as a key encapsulation mechanism.

For information about the standardized quantum-safe algorithms, see the following web pages:

- FIPS 203 ML-KEM: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf
- FIPS 204 ML-DSA: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf
- FIPS 205 SLH-DSA: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf

NIST called for additional digital signature proposals to be considered in the PQC standardization process. They are primarily interested in additional general-purpose signature schemes that are not based on structured lattices. For certain applications, such as certificate transparency, NIST may also be interested in signature schemes that have short signatures and fast verification. NIST is intent on diversifying the post-quantum signature standards. As such, any structured lattice-based signature proposal would need to significantly outperform ML-DSA and FALCON in relevant applications and ensure substantial additional security properties to be considered for standardization.

NIST has selected 14 candidates for the second round of the additional digital signatures for the NIST PQC standardization process. The candidates that advanced are CROSS, FAEST, HAWK, LESS, MAYO, Mirath (merger of MIRA/MiRitH), MQOM, PERK, QR-UOV, RYDE, SDitH, SNOVA, SQIsign* and UOV*.[10] For additional details about the candidates, evaluation criteria, and the selection process, see this NIST report at:
https://csrc.nist.gov/pubs/ir/8528/final

Find more information about the on ramp signature competition go to:
https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures

[10] Candidates marked by an asterisk (*****) were submitted by IBM.

The following candidate KEM algorithms have also advanced to the fourth round:

- BIKE
- Classic McEliece
- HQC

# 1.6 Quantum-safe capabilities with IBM Z

IBM z16 and later supports quantum-safe cryptography in the following ways:

- Infrastructure that protects the integrity of the system
- API functions that can be used by application programs

These methods are described in the subsequent sections.

## 1.6.1 Quantum-safe infrastructure in IBM Z

IBM Z adds features to protect the system from attacks, including threats that might use quantum computers. In particular, the system includes a secure boot feature in which it is protected with quantum-safe technology through the many firmware layers that are loaded during the boot process. Only authentic, IBM-approved firmware is accepted.

This hardware-protected verification of the firmware uses a dual-signature scheme, which uses a combination of quantum-safe and classical digital signatures. The protection is anchored in the IBM Z *Root of Trust*.[11]

Quantum-safe mechanisms also were added to the IBM Z cryptographic infrastructure. The Crypto Express Hardware Security Module (HSM) now uses a quantum-safe dual-signature scheme similar to the one described for the IBM Z server boot process.

Changes were made to the TKE feature to use quantum-safe cryptography when authenticating Crypto Express8S (CEX8S) coprocessors, verifying replies from the CEX8S coprocessors, and protecting key parts in flight for the Common Cryptographic Architecture (CCA). Finally, the IBM Z pervasive encryption functions were updated to use quantum-safe mechanisms for key management.

Other IBM Z enhancements include the following examples:

- IBM z/VM® guest support for quantum-safe APIs on virtualized Crypto Express features for IBM z/OS and Linux on IBM Z
- IBM RACF® quantum-safe encrypted VSAM database support, and other base infrastructure crypto-related enhancements

For details about the IBM Z cryptographic stack, see 4.1, “IBM Z cryptographic components overview” on page 52.

**Note:** IBM Crypto Express7S (CEX7S) includes CRYSTALS-Dilithium support only.

[11] Root of Trust is a source that can always be trusted within a cryptographic system

### 1.6.2 Quantum-safe API functions available to application programs

Integrated Cryptographic Services Facility (ICSF) provides APIs. ICSF is a software element of z/OS. ICSF works with the hardware cryptographic features to provide secure, high-speed cryptographic services in the z/OS environment. ICSF provides the application programming interfaces by which applications request the cryptographic services. These services include (but are not limited to) encrypting data by using software and Crypto Express HSM or CP assist for cryptographic (CPACF) functions.

ICSF offers two different cryptographic APIs for use by application programs:

- Common Cryptographic Architecture (CCA): An IBM proprietary API that includes general-purpose cryptographic functions and the special functions that are required by the payments industry.
- Enterprise PKCS #11: A standardized API that is widely used on many systems for many applications.

CCA and PKCS #11 provide API functions to support quantum-safe digital signatures by using ML-DSA, and to support key agreements by using a hybrid ML-KEM method. You can generate the public and private keys, generate and verify digital signatures, and negotiate a shared symmetric key by using the key agreement protocol.

Although the algorithms are needed to provide quantum-safe asymmetric cryptography, the CCA and PKCS #11 APIs contain the functions you need to implement quantum-safe symmetric cryptography and hashing.

You can encrypt data by using AES, with key sizes ranging 128 - 256 bits. You can use the SHA-2 or SHA-3 hash functions, with hash lengths up to 512 bits. In combination with the new digital signature and key agreement algorithms, this configuration gives a complete suite of quantum-safe cryptographic algorithms.

For digital signatures, one common approach today is to implement dual signatures where data is signed by using the older algorithms, such as Elliptic Curve, and the new quantum-safe algorithms. By doing so, you can meet standards that require the older algorithms, while also providing the higher level of protection that is offered by the quantum-safe algorithms. Meting those standards is easy by using CCA or Enterprise PKCS #11 on IBM z16 and later because the digital signature APIs now offer both classes of signature algorithms.

Finally, the Unified Key Orchestrator (UKO) V3.1.0.7 key management system and UKO setup with EKMF Workstation for secure room operation supports ML-DSA and ML-KEM keys. This support allows you to manage these new key types with the same tool that was available to manage other types ofcryptographic keys.

2

# The journey to quantum safety

As discussed in Chapter 1, “Cryptography in the quantum computing era” on page 1, we have entered a new cryptographic era. The cryptographic landscape is changing. New cryptographic algorithms must be implemented across the enterprise today to provide protection from current and future threats. For most organizations, it is a journey to quantum safety.[1] IBM is leading the way, assisting businesses and organizations on this journey.[2]

In this chapter, we examine the major milestones that need to be consider as you embark on your own quantum-safe journey. We will also share the guidance that is being provided by other organizations, like the National Cybersecurity Center of Excellence (NCCoE) and the European Telecommunications Standards Institute (ETSI). While NIST has published the final PQC Standards for public key cryptography, the required changes and adoption of these new PQC standards will likely need significant planning and preparation.

In the subsequent sections we will take a look at the quantum-safe journey milestones, high-level framework and what is need to get started:

- 2.1, “Quantum-safe journey milestones” on page 16
- 2.2, “A framework for the quantum-safe journey” on page 17
- 2.3, “Quantum-safe migration in review” on page 28

[1] Quantum safety is the practice of using cryptographic algorithms to protect data from attacks by future quantum computers.

[2] Reference: Quantum-safe security for IBM Z

## 2.1 Quantum-safe journey milestones

With any new technology comes new challenges and quantum-safe cryptography is no exception. It is necessary to survey the system landscape and at the same time use the knowledge and insights that are being provided by the standards bodies and the community. Engagement and collaboration with the broader ecosystem, including vendors, legal, business partners and internal organizations will be crucial for the ultimate adoption of the PQC standards.

The high-level milestones that organizations need to achieve in pursuing a quantum-safe cryptographic implementation are shown in Figure 2-1.

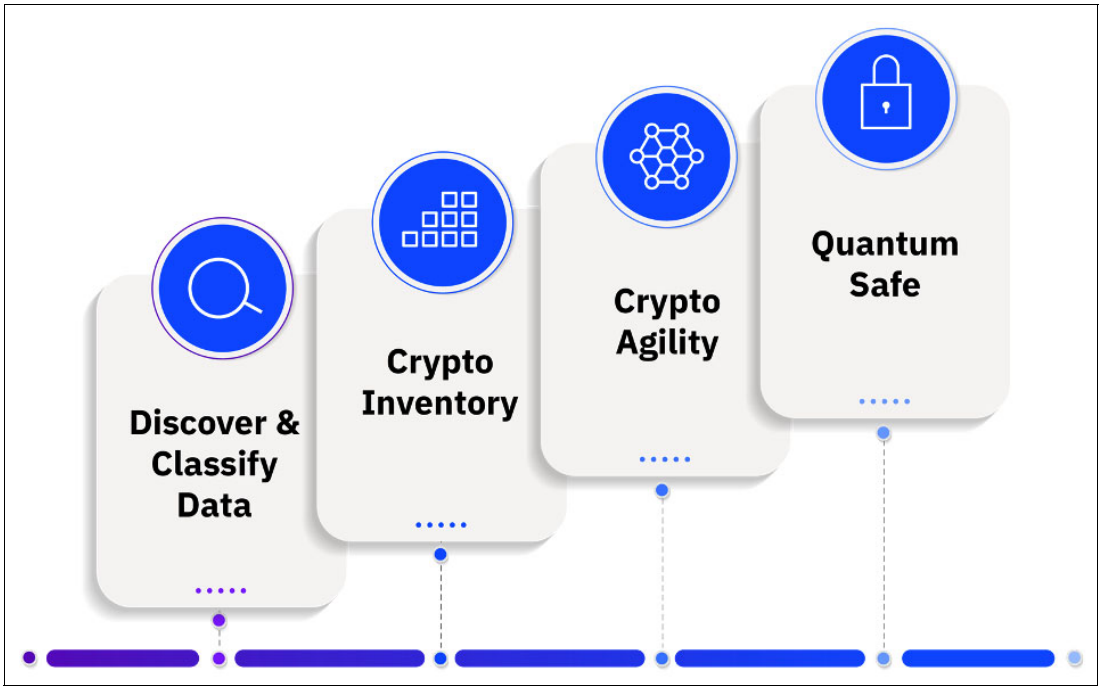



*Figure 2-1 High-level quantum-safe journey milestones*

The initial milestone, Discover & Classify Data involves conducting a data exploration process, classifying the identified data, and assessing whether it requires long term protection while evaluating its vulnerability to potential quantum attacks. The next milestone will involve cataloging the cryptographic assets that were identified in the discovery phase, described in 2.2.3, “Discovery” on page 21, in a Crypto Inventory. A cryptographic inventory serves as a centralized system for the secure and organized management of cryptographic assets, including keys, certificates, algorithms, and other related components, employed across an organization's hardware, software, and services. By leveraging such a system, you can effectively oversee the lifecycle of their cryptographic materials, ensure the proper utilization of cryptographic capabilities, and reduce the risks associated with cryptographic vulnerabilities or mismanagement.

Once you understand the risks, highlighted by the crypto inventory, it is time to prioritize and assess readiness for adopting post quantum standards. For example, decisions like whether to choose to adopt a hybrid approach by combining classical and quantum-safe cryptographic algorithms must be made. One of the aims of a hybrid approach is to provide a layer of security against quantum threats while still supporting legacy systems. It can also provide a flexible and secure way for organizations to transition to the new algorithms while minimizing disruption and managing risks. An important goal is to achieve crypto agility. Cryptographic

agility helps ensure that cryptographic systems can be updated or modified without significant disruption or security risks. IBM defines crypto agility as a system, device component, platform, application, or organization's ability to rapidly adapt its cryptographic mechanisms and algorithms in response to changing threats, technological advances, or vulnerabilities. Crypto agility will provide you with the infrastructure you need to continue and support your quantum-safe journey and will enable you to reach your quantum-safe milestone.

When your cryptographic algorithms and systems are designed to be secure against potential future threats posed by quantum computers, and the quantum-safe algorithms have been deployed, you are well on your way to successfully achieving the state of quantum safety. However, keep in mind that the threat and the PQC standards will continue to evolve, and you will need to put forward plans to keep pace with the latest standards and stay ahead of evolving threats.

For additional insights on cryptographic agility see this IBM Research blog:
https://www.ibm.com/quantum/blog/crypto-agility

## 2.2 A framework for the quantum-safe journey

The general framework laid out in Table 2-1 is a set of guidelines and best practices that can help you understand the threat, plan for the migration and execute the strategy. It can aid with transitioning to quantum-safe algorithms in a secure and efficient manner, while minimizing disruption to business operations and ensuring compliance with industry standards and regulations. Bear in mind that there is no one-size-fits-all solution. Each organization is different, therefore analysis and planning will be required to address the unique needs of your organization.

*Table 2-1 Framework: Quantum-safe journey*

| Stages | Definition |
|---|---|
| Preparation | ► Educate teams/security stakeholders<br>► Follow the standards community and quantum-safe computing<br>► Research migration best practices |
| Discovery | Create a crypto inventory (reusable security asset):<br>► Inventory data handled by the organization<br>► Inventory cryptographic assets and cryptography in-use<br>► Inventory suppliers of cryptographic assets |
| Risk assessment | ► Perform an analysis to discover the security risks<br>► Understand internal and external dependencies.<br>► Consider business and technical factors<br>► Leverage risk assessment reports to prioritize your execution roadmap |
| Risk mitigation | ► Reconsider and possibly redesign how crypto is consumed in your environment.<br>► Determine best mitigation action: retire it, accept it, or fix it |
| Migration to PQC | ► Adopt the approaches your organization needs to protect your data and systems<br>► Work towards building crypto agility<br>► Use your risk-based prioritized roadmap and implement your migration strategy |

From the list in Table 2-1, you can see that there are several stages along the journey. Each organization has different cryptography use cases and usage constraints. It is important that

the collateral that is created by each team be reviewed to provide the best options and plans for your situation.

Consider your cryptographic use in four broad areas:

- Infrastructure
- Applications
- Data protection
- Key Management

### 2.2.1 Get started now

For the last several years, experts were urging organizations to begin planning for the replacement of hardware, software, and services that use the cryptography that is likely subject to attack by a quantum computer.

*The threat that quantum computers pose to our current cryptographic systems is well known. Even though large-scale quantum computers are not yet here, it is critical to take action well before their arrival. Organizations need to be planning now, for the upcoming transition to new quantum-resistant cryptographic algorithms. Failure to do so may mean that your information will not be protected from these future attacks.*

- Dustin Moody, Mathematician, Post-Quantum Cryptography Project Leader, National Institute of Standards and Technology (NIST)

Based on history, it can take a long time to make changes in all the places where change is required. The initial inventory phases can show surprising findings. Not only do you find crypto that must be migrated, but you might also find areas where cryptographic protections are not in place or that cryptography is not correctly implemented and not suitable for the intended purpose.

You might discover that specific source code is no longer available or build tools are no longer available, which makes change difficult and time-consuming. It is advantageous to find automated tools that help with the inventory process.

The authors of code modules might be unknown or no longer work for the company. The new algorithms are not drop-in replacements. Key sizes, signature sizes, performance, and so on must be considered.

In addition, any number of your IT professional staff might need to get involved in your quantum-safe journey (see Table 2-2 on page 19 for examples).

*Table 2-2 Involvement of IT professionals for the quantum-safe cryptography journey*

| Stakeholder | Roles |
|---|---|
| IT security | ► Chief information security officer<br>► Chief security architect<br>► Key management personnel<br>► IT security personnel<br>► Mainframe security administrator<br>► Enterprise security Architect |
| Networking | ► Network administrator<br>► Network architect |
| Auditors | ► Security auditor<br>► Financial regulation office<br>► Compliance officer/auditor |
| Applications | ► Application architect<br>► Application programmer<br>► Application owner |
| Management systems | ► System administrator<br>► Hardware administrator<br>► Storage administrator |
| External parties | ► Customers<br>► ISV representative<br>► Business partners |

Another important reason to start the quantum-safe journey now is because you do not want to keep creating assets that are susceptible to quantum attacks. Use protection methods today so that today's data is protected in the future. New technology takes time to develop, test, and deploy. To avoid costly mistakes and to ensure you have the technology to address your use cases, organizations must start now.

### 2.2.2 Preparation

Before beginning the quantum-safe transition journey, it is essential to educate the key stakeholders on the topic of quantum-safe cryptography and the effects that quantum computing has on classical cryptography. Educate the senior management and senior technical leadership in your organization.

The leader's buy-in is critical for allocating the needed resources to establish the quantum-safe transition project. A leader for the project should be selected, followed by the selection of the core team. The security stakeholders in the overall organization also needs to be educated on the topic. This is important as their time, effort, and expertise will be required.

The core team needs to carefully follow the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) guidelines and also consider the activities of institutional bodies with governance over standards and regulations that are related to public key cryptography for specific geographies and industries.

Set up project goals with the intent of establishing a federated effort and a strategy. Carefully understand the mitigation options for various use cases and transition best practices. It is important to establish a diverse group of experts from the organization, including those people responsible for hardware, firmware, software, security architecture, and secure engineering.

Review your transition plans with the stakeholders to ensure that the proposed actions are secure and meet your organization's security guidelines and policies. Consider detailed design sessions to evaluate the proposals on a case-by-case basis if needed.

## Following industry and standards guidance

Be cautious when searching online, as it is a source of both accurate and inaccurate information about this topic. Make sure to evaluate the reliability and credibility of the sources you find. Several organizations have formed task forces or working groups to discuss quantum-safe cryptography and offer their guidance. We recommend that you review the work being done by these groups. Also, consider guidance from the institutions that provide recommendations for your industry and that recommend or enforce timelines for transition. Experts in the field provide insights that can prove to be useful for any organization looking to start this journey.

### *National Institute of Standards and Technology (NIST)*

NIST provides cryptographic key management guidance for defining and implementing appropriate key-management procedures, using algorithms that adequately protect sensitive information, and planning for possible changes in the use of cryptography because of algorithm breaks or the availability of more powerful computing techniques. NIST has created a draft document providing guidance on transitioning to the use of stronger cryptographic keys and more robust algorithms.

For more information, see the latest draft of the NIST transition guidance.

### *National Cyber Security Center of Excellence (NCCoE)*

NCCoE has formed a Post-Quantum Cryptography (PQC) Working Group. The workgroup is actively developing best practices in the form of white papers, playbooks, and demonstrable implementations for organizations to ease the transition from the current set of public key cryptographic algorithms to replacement algorithms that are resistant to quantum computer-based attacks.

For more information, see this NCCoE web page.

### *Electronic Telecommunications Standards Institute (ETSI)*

The ETSI Cyber Quantum Safe Cryptography (QSC) Working Group aims to assess and make recommendations for quantum-safe cryptographic primitives, protocols, and implementation considerations. These considerations are based on the state of academic cryptography research and quantum algorithm research, and industrial requirements for real-world deployment.

For more information, see this ETSI web page.

### *Federal Office for Information Security (BSI)*

The German Federal Office for Information Security (BSI) provides assessments of the security of selected cryptographic mechanisms, combined with a long-term orientation for their use. The recommendations they make are reviewed annually and adapted if necessary. A work stream on PQC, co-chaired by France, Germany and the Netherlands, has been created as part of the Network and Information Systems (NIS) Cooperation Group following a recommendation of the European Commission. They are preparing a roadmap for the transition to Post-Quantum Cryptography to ensure the quantum resilience of the European Union's digital infrastructures.

For more information, see this BSI web page.

#### *Canadian Forum for Digital Infrastructure Resilience (CFDIR)*

CFDIR has formed a Quantum Readiness (QR) working group. The QR working group is responsible for creating guidelines and best practices documents, which are updated annually. These documents are important because they assert quantum technologies are poised to play a major role in Canada's future, from its national security to its economic standing. The materials created by the QR working group are designed to help critical infrastructure sectors plan and prepare for the transition of digital systems to use standardized quantum-safe cryptographic technologies and solutions in order to protect the confidentiality and integrity of their data and systems.

For more information, see CFDIR Quantum Readiness Guidance.

#### *French National Cybersecurity Agency (ANSSI)*

ANSSI is committed to ensuring that public administrations, public services, and businesses can take full advantage of a secure and trustworthy digitalization. The goal is to provide direction to industries developing security products and outlining the transition agenda for quantum-safe cryptography.

For more information, see this ANSSI web page.

#### *Dutch General Intelligence and Security Service (AIVD, CWI & TNO)*

The General Intelligence and Security Service of Netherlands is advising businesses to start working on mitigating measures now. They have published a handbook to assist organizations with concrete steps and advise them how to mitigate the threat of quantum computers on today's cryptography.

For more information, see The PQC migration handbook

It will be necessary to establish some of your own operational best practices as you work through dependencies with partners and vendors. Work closely with risk and legal teams to ensure the methods and actions that are planned are in line with approved guidelines.

### 2.2.3 Discovery

Once you have educated the stakeholders, formed a core team and have a good understanding of the threat and best practices, it’s time to begin your PQC project. As discussed in 2.1, “Quantum-safe journey milestones” on page 16, one of the first milestones on the quantum-safe journey is to discover and classify the cryptographic data. The data inventory must contain information about your critical data assets. It is a comprehensive catalog of the data assets in the enterprise/organization. Document important information about the cryptography used to protect the data, the data protection requirements and how long that protection must be in place. Also, record any standards or regulations that govern the protection of the data.

After completing that task, the next step is to create a cryptographic inventory. This section will go into more details on what a crypto inventory is and the kinds of cryptographic assets and metadata you should include in the inventory.

#### What is crypto inventory

Crypto inventory typically refers to the process of tracking and managing cryptographic assets, keys, certificates, algorithms, and related components in a secure and organized manner. A cryptographic inventory can help organizations manage the lifecycle and security of cryptographic materials, ensuring proper encryption and decryption capabilities and mitigating the risk of breaches or mismanagement.

Here are the main components typically involved in a cryptographic inventory:

- **Cryptographic keys:** These are essential to cryptographic systems and include keys for encryption, decryption, signing, and verification. The inventory might track:
  - Key generation date
  - Key type (such as RSA, AES, ECC)
  - Expiration dates
  - Usage (such as for data encryption, signing certificates)
  - Key lifecycle (creation, usage, rotation, revocation)
- **Certificates:** Digital certificates, like SSL/TLS certificates or code-signing certificates, ensure trust in cryptographic communications. The inventory tracks:
  - Certificate type
  - Issuer
  - Expiry date
  - Associated private/public keys
  - Revocation status (such as CRLs, OCSP)
- **Algorithms:** Cryptographic algorithms used (such as AES, RSA, ECC) are part of the inventory, often with metadata like supported versions, modes or known vulnerabilities.
- **Secure storage:** Inventory management often includes the physical and logical storage of cryptographic assets (hardware security modules or HSMs, key management systems).
- **Access control:** Inventory management may involve who can access cryptographic keys or certificates, ensuring only authorized users or systems can use the keys.
- **Compliance and auditing:** Cryptographic inventory management often ensures compliance with regulatory frameworks (such as GDPR, HIPAA) and internal policies. Audits may also be performed to verify key usage and integrity.
- **Cryptographic hardware:** In some contexts, cryptographic inventory management extends to tracking specialized hardware used for cryptography, like HSMs, TPMs (Trusted Platform Modules), and secure enclaves.

## How to build a crypto inventory

Depending on the size and complexity of your organization, building a crypto inventory can be a huge undertaking. Consider creating a questionnaire that can be used to capture important information. The questionnaire should be tailored for your platform or target of evaluation, including several areas, such as hardware, firmware, operating systems, virtualization, applications, solutions, and data elements.

A software based questionnaire backed by a database repository is a useful tool for gathering the information from the key stakeholders. A software tool can provide centralized management of the information about your cryptographic assets making it easier to track. It can provide secure storage of the information and ensure only the authorized personnel have access. It can automate many tasks that would require manual interaction and facilitate collaboration among team members. It will also allow you to track progress over time.

The questionnaire helps stakeholders understand what they need to look for. The questionnaire can be used with the tools that are available to compile the baseline inventory. IBM Z provides tools that can help with crypto discovery and inventorying. For more information, see 4.2.2, “Establishing a cryptographic inventory” on page 62.

Another important tool that can be used during the crypto inventory process is a Cryptographic Bill of Material (CBOM). A CBOM is an object model and tool that describes cryptographic components and their dependencies in software. It is an extension of the Software Bill of Materials (SBOM) concept, which uses a standardized list of components to

describe software and systems. Support for CBOM is included in CycloneDX v1.6 and higher. Review this article for additional details.

Collaborate with component technical leaders to answer any questions they might have about the questionnaire, and how the questionnaire should be completed. Each component leader needs to work with their team to complete the questionnaire and return information to the core team. Table 2-3 provides some suggestions on the key areas that the questionnaire should cover.

*Table 2-3 Cryptographic inventory questionnaire*

| Area | Information collected |
|---|---|
| Identity | ► Name of component or application<br>► Feature or function that uses crypto<br>► Person responsible for component and contact Information |
| Symmetric crypto | ► Algorithm<br>► Function (encryption, decryption)<br>► Symmetric key size<br>► Length of time data needs to be kept secret<br>► Sensitivity level of the data protected (H/M/L) |
| Asymmetric crypto | ► Algorithm<br>► Function/protocol/method<br>► Asymmetric key size |
| Hashing | ► Algorithm<br>► Digest size |
| Crypto services | ► Crypto provider<br>► Crypto provider product version<br>► Vendor name<br>► How is crypto provided? (HSM, software library)<br>► How is the crypto implemented? (hardware, software)<br>► How is crypto provider version kept current? |
| Interoperability | ► Do you control the full stack?<br>► Do you work with a vendor or partner?<br>► Is the partnership internal or external to the team or organization? |
| Policies/standards/regulations | ► Are there policies governing the selection and use of the cryptography? If so, which?<br>► Are there standards or regulations governing use of the cryptography? If so, which?<br>► Are there associated configuration files?<br>► Can the component's crypto "state" or status, configuration status, and so on, be queried or monitored?<br>► Consider cyber resilience: Are there single points of failure or simple denial of service (DoS) choke points? |
| Key management | ► Where do the keys come from?<br>► Where are the keys stored?<br>► Is a key management system or key server used?<br>► Is a key transport protocol used?<br>► Are the derived or created keys used to wrap or protect other keys? |
| Preliminary assessment | Has a gap been identified? (Crypto being used must be updated, mitigation plan is needed?) |

Each component leader needs to gather this information and provide it to the core team for review. Depending on your organization's goals and needs you may choose to include additional information in this questionnaire; however, Table 2-2 on page 19 is a good place to start.

Leverage the questionnaires to help each component team develop preliminary plans for the use of stronger cryptography for symmetric crypto and hashing or quantum-safe crypto schemes for asymmetric crypto. Look at areas where crypto is being used and look for places where cryptographic protections might need to be added.

Getting answers to these questionnaires may turn out to be an iterative process. Hold design review sessions and use that as input for the guidance and feedback about the plans and strategies that are being developed. Because the cryptographic inventory is a living document, the inventory documentation must be updated as changes are made.

There are many tools that can aid you with the process of discovering crypto on IBM Z. We discuss them in more detail in Chapter 5, "Creating a cryptographic inventory" on page 77.

### Maintaining and securing the inventory

Maintaining and securing the inventory is critical. Make sure you treat the inventory as the security-sensitive artifact that it is. Access to the contents must be controlled. Component owners can access their information but not the information of other components unless a need to know exists and collaboration among teams is needed.

Also see IBM Z Crypto Discovery and Inventory.

## 2.2.4 Risk assessment

Once you have built the necessary crypto inventory and understand the affected areas, you will have the ability to perform a holistic risk assessment.

### Perform a gap analysis

It is important to compare the current cryptographic state with the desired state, with the goal of identifying the gaps. The purpose of the gap analysis is to identify the areas where the organization needs to make changes or improvements to meet your identified goals and comply with the latest cryptographic standards. By identifying these gaps, organizations can prioritize their efforts and develop a plan of action to address them. It is important to understand that during your risk assessment you may discover that you cannot change every area that is identified in your inventory. Therefore, you must prioritize. Develop a multi-phased roadmap, address the areas that are at most risk and protect the most critical assets first. Make the necessary changes so that you do not continue to use vulnerable cryptography where possible.

Also, consider the areas where the changes are simpler to make. Consider what preparatory steps can be made. For example, if using TLS, take steps to migrate to TLS 1.3 in preparation for the PQC enhancements. Current efforts are focused on updating the TLS 1.3 protocol to support the quantum safe algorithms rather than to introduce a new version of TLS. You may also come across scenarios where some items need to be implemented on day one and some items need to be updated over time. There are several factors that will influence the decisions about where items land on your roadmap. Many will fall under the category of internal and external dependencies.

## Understand internal and external dependencies

Evaluating dependencies is critical. The uncertainties, costs, and the value of the option to the system also needs to be considered. You will have dependencies both inside and outside your organizations, which will further affect the prioritization of changes.

Dependencies can determine the location and timing of changes on your roadmap. You must have a mitigation strategy in place. This strategy includes knowing the mitigation options that are available to you and when to use those mitigation options.

Here are some examples of the dependencies that you may face:

- NIST PQC standards not yet integrated into your ecosystem.
- Other technology and industry standards and guidelines not yet updated; for example, IETF community RFCs, including:
  - TLS/SSL/SSH standards
  - PKI standards for certificates
  - Network security and communication protocols
- Availability of quantum-safe hardware products from vendors
- Dependencies on software and hardware solutions
- Availability of crypto libraries and hardware that supports the quantum-safe algorithms

Identify all the operating environments where algorithm support and secured quantum-safe libraries are needed. Vendors and 3rd parties should be contacted to understand their quantum-safe roadmaps and plans. Based on feedback from these vendors, the roadmap may need to be revised. Appendix G of the Canadian National Quantum-Readiness Best Practices and Guidelines documents PQC roadmap questions to ask vendors. It also provides guidance for engaging with 3rd parties.

It is also critical that your strategy include extensive testing. Solutions must be prototyped to understand usability and performance effects. Some of the mitigations involve the use of longer keys and artifacts, which requires more space and resources. It is critical that you review your threat models with your secure engineering team to ensure you did not inadvertently introduce a vulnerability.

After prioritizing the work based on the risks and dependencies, you will be ready to create a multi-phase roadmap. Maintain flexibility in the roadmap as discoveries during the planned execution phase are expected.

If you need help performing your quantum risk assessment, IBM Technology Expert Labs can conduct a holistic quantum risk assessment by creating a comprehensive inventory of cryptographic materials, including keys, certificates, and algorithms. The assessment will help identify and mitigate vulnerabilities like weak encryption and poor key management. The following domains are covered by the assessment:

- Infrastructure encryption services
- z/OS® ICSF encryption services
- Key Management Services
- Network encryption services
- Data at rest encryption services
- Application encryption services

Contact IBM Technology Expert Labs to get additional details.

## 2.2.5 Risk mitigation

After understanding where cryptography is being used and completing the risk assessment, it is necessary to determine how you will address your findings. You must develop a plan of action to address the gaps and achieve the desired state.

### Decisions

Determine the best mitigation action for the asset in question: retire it, accept it, or fix it. It is important to leverage the risk assessment to inform your decisions about your transition roadmap.

- **Retirement**:
  - The asset is no longer needed or has been replaced by a newer more secure asset.
  - The asset has reached the end of its lifecycle or is no longer supported.
  - The asset has been compromised or is otherwise insecure.
- **Acceptance**:
  - The asset is secure, is not vulnerable and has not been compromised.
  - The asset is still needed and in use or based on the threat model is a hard to reach target.
  - The asset is compliant with current regulations, standards and policies and is scheduled to be updated prior to falling out of compliance.
  - The risks associated with the asset are acceptable to the organization
- **Fixing**:
  - The asset has vulnerabilities or weaknesses that can be addressed through updates.
  - The asset is not compliant with current regulations, standards or policies, but can be made compliant through updates or configuration changes.
  - The risks associated with the asset are not acceptable and can be reduced to acceptable levels or eliminated through remediation actions.

The decision to retire, accept or fix an asset should be made taking into account the assets usage, the risks it presents and the potential impact on the organization's security posture. Some decisions may require case by case analysis to make the right decision. Based on your risk-based prioritization, it will be necessary to look at the options you have to address the items which must be fixed.

## 2.2.6 Migration to PQC

After you understand where crypto is being used and the items that must be fixed are identified, it is critical to know your mitigation options. A transition strategy is needed that is based on industry guidance and the use cases that your organization must address.

The primary options include the following examples:

- **Select secure algorithms**: Follow NIST and if needed Commercial National Security Algorithm Suite (CNSA) 2.0 Guidance for recommendations on transitioning to the use of stronger cryptographic keys and more robust algorithms
- **Implement dual signing**: A dual signature consists of at least two signatures on a common message. According to guidance provided by NIST, one signature is generated with a NIST-approved signature scheme as specified in FIPS 196 or FIPS 204, while the other signatures can be generated by using a different signature algorithm. For quantum-safe, the second signature is a quantum-safe signature. The signatures must be parsed and verified separately; if either fails, the signature for the object fails.

- **Implement hybrid key establishment schemes**: This scheme is a combination of two or more components in which cryptographic key-establishment schemes are used. Based on NIST guidance, the scheme is considered secure if at least one of the schemes remains secure. Therefore, one of the components of the hybrid scheme must be NIST-approved; for example, a discrete-logarithm based scheme from NIST SP 800-56A or an integer-factorization scheme from SP 800-56B, and the other component is a post-quantum cryptography scheme. NIST SP 800-56C describes a hybrid key establishment construction. The specification describes a process that allows a key derivation method permitting a shared secret “Z” to be concatenated with a value protected by a quantum-safe key encapsulation mechanism (KEM). NIST has published an initial draft special publication containing explanatory and educational material to aid in the general understanding of KEMs. See the initial draft of NIST SP 800-227.

Each specific use case must be evaluated to determine whether the implementation costs, performance reduction, and solution complexity can be contained. The hybrid and dual schemes require a security review to ensure that security-related implementation errors were not introduced.

For more information about common cryptography use cases and the challenges a cryptographically relevant quantum computer (CRQC) can present and the quantum-safe solutions that are provided by IBM Z, see Chapter 3, “Using quantum-safe cryptography” on page 31.

## Cryptographic agility

It has probably become clear that piece by piece, enterprises must change the underlying cryptography that they use. However, this instance is not the last time such a change is required. Quantum safe migration provides an opportunity to rethink how applications use complex cryptography such that future changes, updates, and patches are much simpler to apply. Hence redesigning and modernizing keeping cryptographic agility in mind will be a key. It is a necessity to create designs that lend themselves to change with new crypto algorithms in the future.

When we think of cryptographic agility, we must broaden our view of its scope beyond cryptographic migration. We should not simply focus on swapping from one crypto algorithm or standard to another. We must think about how we transition to architectures that offer agility for ongoing cryptographic migrations over time.

We know that cryptographic algorithms break or become obsolete. The topic of agility is relevant throughout the lifecycle of crypto from its definition and introduction into standards through its retirement as being obsolete or no longer secure.

The early phases of a cryptographic algorithm's lifecycle are handled by experts in the field in academia and industry. Table 2-4 lists a some of the cryptographic agility areas that are most important to our discussion.

*Table 2-4 Cryptographic agilities*

| Agility | Definition |
|---|---|
| Algorithm | Ability to select algorithms based on their combined security functions or organizational policy |
| Protocol | Ability to move to new versions of a protocol, such as 1.1 to 1.2 to 1.3, for TLS |
| Implementation | Ability to add crypto features or algorithms to hardware or software, which results in new, stronger security features |

| Agility | Definition |
|---|---|
| Platform | Ability to adapt to platform-specific constraints or support for cryptographic operations |
| Retirement | Ability to retire crypto systems that became vulnerable or obsolete |

Cryptographic agility is an active area of research. An example of some of the guidance we see coming out of research areas recommends that we no longer hardcode crypto specifics in applications. Instead, the use of a higher-level abstraction layer allows for passing in algorithm specifics so that they can be changed when needed without changing the application based on policies. Some are in favor of this approach and some are not, citing concerns over the introduction of certain vulnerabilities and risks that need to carefully be managed. For example, if the policy controlling the runtime selection of algorithms is not correctly configured or is based on inaccurate information, it could lead to the selection of insecure algorithms. Policies should be regularly reviewed and updated to ensure they align with current security best practices and industry standards.

From a broader standpoint, cryptographic agility is about an information security system's ability to rapidly switch to alternative cryptographic primitives and algorithms without making significant changes to the system's infrastructure.

When considering your cryptographic strategy in light of quantum-safe transition, spend some time studying this topic and explore how to best improve your cryptographic agility.

For more information and an example, see 4.2.3, "Considering cryptographic agility" on page 65.

## 2.3 Quantum-safe migration in review

The following list summarizes the key points discussed in this chapter:

- Obtain senior level management buy-in
- Educate your organization on quantum risks and quantum-safe cryptography
- Create a quantum-safe crypto core team
- Inventory current crypto in use
- Control access to the inventory
- Identify areas that are most vulnerable
- Research cryptographic agility and quantum-safe cryptographic algorithms to determine which algorithms suit your use cases
- Identify crypto API providers and crypto hardware to accelerate performance
- Develop implementation validation and testing tools
- Identify all communications protocols with quantum-vulnerable crypto algorithms
- Identify automated crypto discovery tools
- Update the processes and procedures of developers, implementers, and users
- Develop a risk-based approach, considering security requirements, business operations, and mission impact
- Identify a transition timeline and resources
- Prepare to follow strategies to protect digital assets and systems

While these high-level steps are based on our own experience as well as the industry guidance, it is by no means a holistic list. You may find deviations or additional steps as you start your own journey. However, one important underlying take-away is that the journey to quantum safety will be a long and complex one. It is essential that you start now! In Chapter 3, “Using quantum-safe cryptography” on page 31 we share some examples of the types of threats and use cases that should be prioritized and can be used for your risk assessment and mitigation process.

There is no doubt that IBM Z has been at the leading edge of driving quantum safety for our clients, we are also committed to helping our clients with their distributed environments. Some of the solutions that are available today are:

- **IBM Guardium Quantum Safe**

  Guardium Quantum Safe is designed to help enterprises gain visibility into their cryptographic posture and manage associated risks effectively. Security users can identify cryptography use across the enterprise, track changes, and draw insights to make informed decisions about their post-quantum cryptography strategy. Through seamless integration and robust reporting capabilities, IBM Guardium Quantum Safe supports enterprises in maintaining strong data security controls, aligning with regulatory demands, and building cryptographic agility. It is a module of the IBM Guardium Data Security Center, which empowers teams to collaborate across the organization to protect their sensitive data.

- **IBM Quantum Safe Explorer**

  Quantum Safe Explorer equips organizations with the ability to find cryptographic operations across the entire enterprise application, identify coding bad practices, and aggregate the scans to draw insights from a portfolio dashboard view. The timely detection of cryptographic vulnerabilities allows CISOs and security analysts to develop a plan, create a budget, and prioritize remediation to address the quantum security risks.

- **IBM Quantum Safe Remediator**

  Quantum Safe Remediator provides you with capabilities to seamlessly upgrade your existing IT infrastructure to quantum-safe, enhances cyber resiliency, centralizes cryptography management and enables a seamless transformation to quantum-safe IT infrastructures through performance testing. By implementing encryption across the enterprise application, network, and third-party solutions, it helps build crypto-agility and protect against “harvest now, decrypt later” scenarios.

To learn more about these solutions visit our website:
https://www.ibm.com/quantum/quantum-safe

3

# Using quantum-safe cryptography

In this chapter, various use cases are introduced to illustrate the threats that many organizations face with the rise of quantum computing for use cases that are related to confidentiality, integrity, authentication, and nonrepudiation.

Across these use cases, the challenges are addressed and how the quantum-safe capabilities that are provided with IBM z16 and later can help to overcome these challenges and ensure the security of sensitive and valuable data into the future.

Also, each use case includes applications for quantum-safe encryption capabilities across different industries and the specific enhancements with IBM z16 and later that enable these adoption patterns.

This chapter includes the following use cases:



# 3.1 Protecting sensitive data

Classical cryptographic algorithms are widely used to protect data that is at rest and in flight. Cryptographic algorithms are used for having the capabilities of secrecy, integrity, authentication, and nonrepudiation.

Organizations use symmetric keys and public key encryption for data protection schemes. Most organizations encrypt sensitive data to protect it against insider threats, unauthorized user access, and accidental data exposures.

An organization's data is at risk on the internet and communication network when Shor's quantum algorithms[1] are used to break public key encryption schemes. Suppose that a cybercriminal gets access to a powerful quantum computer, they can decrypt lost or harvested confidential data by determining encryption keys.

In highly regulated industries, organizations must protect subject data rights to personally identifiable information (PII) or personal health information (PHI) and comply with standards, such as GDPR, NIST, ISO, SOX, CCPA, and PCI. Data privacy and security are critical to avoid penalties and the cost of data breaches because digital trust and brand reputation are at stake. Organizations need quantum-safe encryption to protect their sensitive data and maintain the confidentiality of trade secrets in the quantum era.

## 3.1.1 Problem statement

An organization's long-lasting data can be at risk from "harvest now, decrypt later" attacks. These attacks are carried out offline or as passive attacks on confidentiality versus as an online attack against a security protocol. That is, an adversary might carry out the attack by collecting data or public information today and later attempt to recover the secret key that is used to encrypt the data. This process might be done by mounting a brute force attack to find the secret key. The the private key that is used in a key negotiation step also might be derived by attacking Rivest, Shamir, Adleman (RSA) or Elliptic Curve Cryptography (ECC) if a public key protocol was used for secret key establishment. The bad actor's goal is to find the secret key that was used to encrypt the data.

Organizations must start the use of quantum-safe methods to protect their data now so that more vulnerable data is not produced. Several symmetric algorithms are not secure when Grover's algorithm[2] is used to search the key space.

Organizations can mitigate the risk by switching to strong encryption algorithms, such as AES. As discussed in 1.4, "Cryptographic vulnerabilities possible with quantum computers" on page 8, organizations should not continue to encrypt their sensitive data by using algorithms, such as 2TDEA or 3TDEA, whose strength is not sufficient in the quantum computing era. These algorithms for new encryption use cases must be retired or no longer used when possible.

## 3.1.2 Solving this challenge by using IBM Z capabilities

Organizations must be concerned about data protection today. Even though the threat of a quantum computer might be in the future, organizations must ensure that they are not continuing to create encrypted data by using methods that are not quantum-safe, especially for long-lasting data.

[1] Reference: Shor's Algorithm - Programming on Quantum Computers - Coding with Qiskit S2E7
[2] Reference: Grover's Algorithm - Programming on Quantum Computers - Coding with Qiskit S2E3

It is essential to understand data classification and where potential exposures exist. It is also critically important to have a plan for retiring versions of your data that were protected by using the retired algorithms. If that data remains in systems, an attacker might find it and break it without attacking the version that is newly protected with the newer algorithm.

In 2.2.3, “Discovery” on page 21, we discussed creating a cryptographic inventory. The security assets help organizations identify which data is encrypted and which encryption methods were used.

The inventories must be reviewed to highlight weak or vulnerable encryption cryptography that is used in the enterprise to protect data. IBM provides tools to help create the cryptographic inventory, such as ADDI, ICSF cryptographic usage tracking, CAT, zERT Network Analyzer, and the Crypto Statistics Monitoring tool, as described in 4.2.2, “Establishing a cryptographic inventory” on page 62.

After the vulnerable data assets are identified, a data protection strategy is created. The strategy is informed by using knowledge about how the sensitive data must be protected.

IBM z16 and later offer several features that can be used to protect the data, such as pervasive encryption. This encryption feature uses AES encryption with 256 bits. Its internal key management support uses quantum-safe protections with a hybrid key exchange mechanism that uses CRYSTALS-Kyber and Elliptic Curve Diffie Hellman (ECDH), and dual-signing scheme that uses CRYSTALS-Dilithium and Elliptic Curve Digital Signature Algorithm (ECDSA). The pervasive encryption features include Linux on IBM Z protected key dm-crypt, z/OS data set encryption, coupling facility encryption, and IBM z/VM encryption.

Quantum-safe APIs available through ICSF can be used to protect data. IBM z16 and later provide quantum-safe key management APIs and lifecycle management support for the organization to generate the encryption keys and also encryption APIs that can be used to encrypt the data (see Figure 3-1).

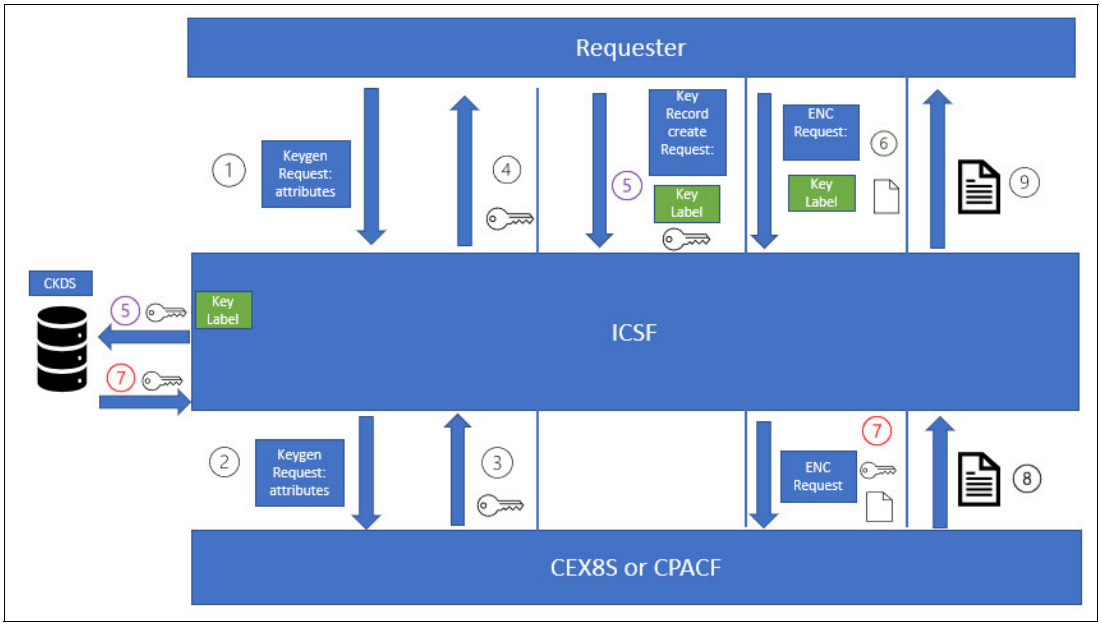



*Figure 3-1 Data encryption process*

The following process is used to encrypt data on the IBM z16 and later as shown in Figure 3-1 on page 33:

1. The Requester calls ICSF to generate an AES 256-bit CIPHER key. The Requester is a component that uses the quantum-safe APIs, such as an application or key management tool.
2. ICSF sends a request to Crypto Express8S (CEX8S) to generate a secure key.
3. CEX8S returns the secure key to ICSF.
4. ICSF returns the key to the Requester.
5. The Requester calls ICSF to add a key record to the CKDS that contains the key, which later is referenced by its associated key label.
6. The Requester calls ICSF to encrypt data by using the key label.
7. ICSF retrieves CIPHER key from the key data set. If requested, the secure key is converted to a protected key. A secure key request, including the key and data, are sent to the CEX8S. Protected key requests (including the key and data) are sent to CPACF.
8. The crypto engine (CPACF or CEX8S) returns the ciphertext to ICSF.
9. ICSF returns ciphertext to the Requester.

### 3.1.3 Industry applications

In highly regulated industries, organizations must provide data security and protection to meet the privacy regulations and compliance requirements that are outlined in their sensitive data retention policies. Examples are healthcare and insurance organizations with patient health information (PHI) in electronic medical records (EMR) that must be protected. These and other organizations must encrypt and store confidential documents for a long period.

Information that requires special attention includes tax documents, legal agreements, trade secrets, and clinical trials. Customer data and PII must be protected. Financial and banking organizations must protect customer data in mortgage and loan processing applications. They must also safeguard banking (PAN data) or payment card (IBM PIN® data). Organizations must ensure that their sensitive data is being protected by using quantum-safe encryption methods.

## 3.2 Sharing keys securely

This use case describes the requirement for organizations with confidential information, such as intellectual property, to share their sensitive data with Business Partners and third parties by establishing keys with traditional key exchange protocols.

In addition, the threat that quantum computing has on these processes are explained in-depth and how IBM z16 and later capabilities can overcome these challenges. Those quantum-safe capabilities can help ensure that partnering organizations securely share their intellectual property and proprietary information to avoid information disclosure and protect it from competitors to avoid reputation damage, profit loss, and brand impact.

Across many industries, organizations use their valuable information, sensitive data, and intellectual property to compete and succeed in a global market. These digital assets oftentimes constitute over 80% of the organization’s total value.

Organizations that have strategic relationships with each other, such as Business Partners, often need to share this valuable information to collaborate and develop solutions and

products. As important as it is to enable this collaboration and allow for sharing of information, it is equally important that this valuable information stays out of the hands of competitors and adversaries. The unintended disclosure of an organization's intellectual property and sensitive information can result in irreversible damage to the organization's reputation, along with profit loss and long-lasting brand impact.

Traditionally, organizations keep their sensitive information and digital assets protected by encrypting their data within their data boundaries. The ability to share these encrypted assets with partners implicates a need to securely share keys with their partners so that they can also access the data. This process is referred to as a *key exchange* and often used a key agreement protocol.

### 3.2.1 Problem statement

In the past, this key exchange process was used by partnering organizations in a way that allowed them to securely derive the same encryption key over a public, insecure channel. Regardless of who was eavesdropping on the public channel (whether a competing organization or a bad actor), the mutually derived encryption key maintained its confidentiality because of the ability to establish a secure connection over this public, insecure channel by way of key exchange schemes.

Traditionally, the ECDH key agreement protocol is a method that is used to derive keys for a key exchange process. This key agreement scheme allows two business parties, each with their own ECC key pair (consisting of a private and public key) to establish a shared secret by using an insecure channel, and it begins with each party sharing their public keys.

However, a key agreement scheme solves one piece of the key exchange puzzle. To create a truly secure key exchange protocol, you must also solve the "trust" problem that is associated with exchanging keys.

Although the public key can be shared over an insecure channel, a mechanism must be in place to ensure that the key that is labeled as party A's public key really belongs to party A and the key labeled party B's public key really belongs to party B to avoid a man-in-the-middle attack.

This problem can be addressed by cryptography; however, this use case focuses on the key establishment piece of the key exchange puzzle. For our example, we assume that a trust mechanism is in place.

If party A and party B wanted to participate in a key exchange process to obtain a shared key, they each generate their ECC public-private key pair and share their public keys with each other. Each party can use the ECDH protocol to derive the same shared key.

For example, party A uses their private key in combination with party B's public key in the protocol, while party B uses their private key in combination with party A's public key. The result is that both parties now have the same shared secret. This method of key exchange to derive the same shared secret historically was considered secure because neither party had to share their private key to perform the key exchange.

Traditionally, this use of public key cryptography for key exchange was considered sufficiently secure. The only means for a cybercriminal to derive the shared secret key was to obtain one of the party's private keys.

Although the public and private keys in an ECC key pair are mathematically related, it takes a conventional computer millions of years to derive the private key from one party's public key because of the significant computational capacity that is required to perform this operation.

However, the rise of quantum computing brings about many challenges to keeping the shared secret key secure.

Although specific symmetric key algorithms, such as AES encryption keys, are considered quantum-safe, the method of sharing those keys in a key exchange by way of public key cryptography is no longer considered to be quantum-safe, which results in the vulnerability of these symmetric keys.

After quantum computers have the computing power to perform Shor's Algorithm, an adversary with access to a quantum computer can solve the elliptic curve discrete logarithm problem exponentially faster than a conventional computer. As a result, the private key of one party can be derived from their public key in only a matter of hours.

After the key is in the hands of an adversary, the same shared key that is derived by the two parties might be generated with the ECDH key exchange scheme. Then, the bad actor can access the organization's valuable information.

To make matters worse, data that is encrypted today by using symmetric keys that are exchanged between two parties by way of public key cryptography is still not safe, even in the absence of quantum computers of sufficient scale.

Because the encrypted data, along with the public key, can be harvested today, an adversary with access to a quantum computer in the future can perform Shor's algorithm to break this public key cryptographic algorithm to expose the sensitive data.

### 3.2.2 Solving this challenge with IBM Z capabilities

Valuable information and digital assets that partnering organizations share with each other are no longer secure because of methods that are used by adversaries to harvest the encrypted data now.

IBM z16 and later provide quantum-safe capabilities to circumvent this security challenge and allow for the secure key exchange between two parties in a quantum-safe manner. With IBM Z, organizations have a reliable method to securely share encryption keys between parties through a hybrid key exchange scheme.

Although traditional key exchange schemes relied on public key cryptography alone to derive a shared secret, a hybrid key exchange scheme relies on classical cryptography (such as ECDH) and a quantum-safe cryptographic algorithm (such as ML-KEM). ECDH is a key agreement protocol and ML-KEM is a key encapsulation mechanism.

When used together, organizations have an efficient way to securely exchange keys, even in the face of high-powered quantum computers. This secure exchange is made possible with the introduction of quantum-safe APIs within the Crypto Express8S feature (CEX8S). This feature is provided with IBM z16 and later to securely derive quantum-safe encryption keys by using the hybrid key exchange mechanism that is performed in the CEX8S by using the CCA API or the Enterprise PKCS #11 API.

When two Business Partners (party A and party B) take advantage of this hybrid key exchange scheme on IBM z16 and later, they can safely derive keys that are used to protect their shared sensitive information (see Figure 3-2 on page 37).

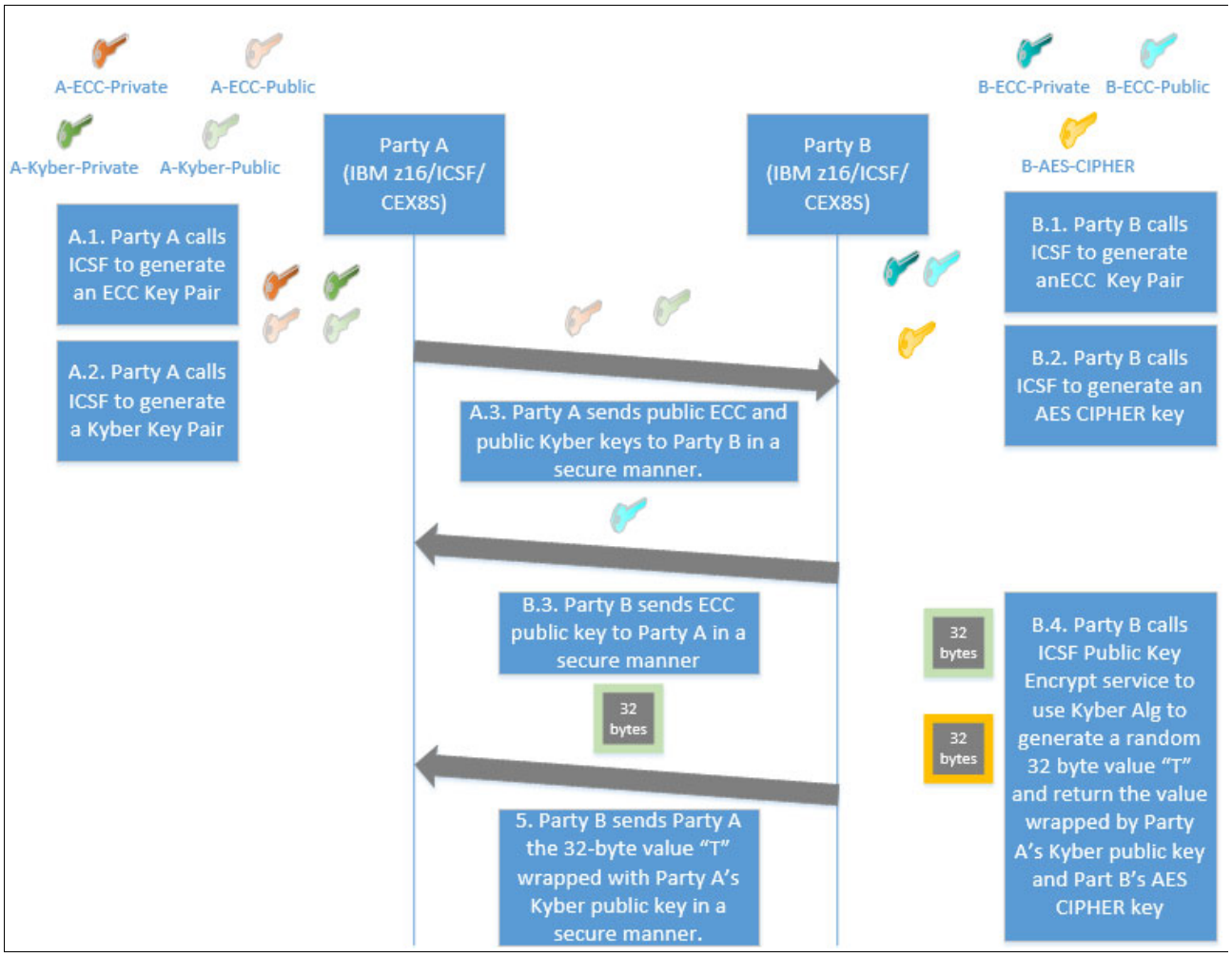



*Figure 3-2 Key and data setup for hybrid key exchange - CCA example*

By using this capability, party A generates an ECC key pair and a quantum-safe ML-KEM key pair (denoted by Kyber). Party B must generate an ECC key pair and an AES CIPHER key. The two parties then exchange their public keys.

Party B then uses CEX8S APIs to generate a random 32-byte secret and encrypt it with party A's ML-KEM public key and an AES key that is owned by Party B. The secret that is protected with the ML-KEM public key is transmitted to party A. Although it is being sent over an insecure channel, not even a quantum computer can break this algorithm because it is considered to be quantum-safe (see 1.5, "Algorithms to counter CRQC attacks" on page 9).

After party A receives the wrapped secret, they then use their ML-KEM private key to recover the 32-byte secret from party B. Now that both parties have each other's ECC public key and the 32-byte secret, they can each use the ECDH key agreement protocol to create a shared secret just as they do in the classical key exchange process (see 6.7, "Quantum-safe key exchange" on page 129).

However, this shared secret is used with the mutually known 32-byte secret in the key material creation process to compute the same shared key by using ECDH. This 32-byte secret is protected with the quantum-safe Kyber algorithm. By using the CEX8S APIs, the secret information is protected by the hardware security module and never appears outside in the clear in the memory of the host computer.

After this shared secret key is derived by both parties, it can be used to encrypt either party's data, and the data itself can be shared between parties (see Figure 3-3). Even in the presence of a quantum computer with sufficient computing power, the key cannot be derived. Although a quantum computer might derive either party's ECC private key, the 32-byte secret that is used in the ECDH protocol cannot be exposed because it was protected by using a ML-KEM key (denote by Kyber).

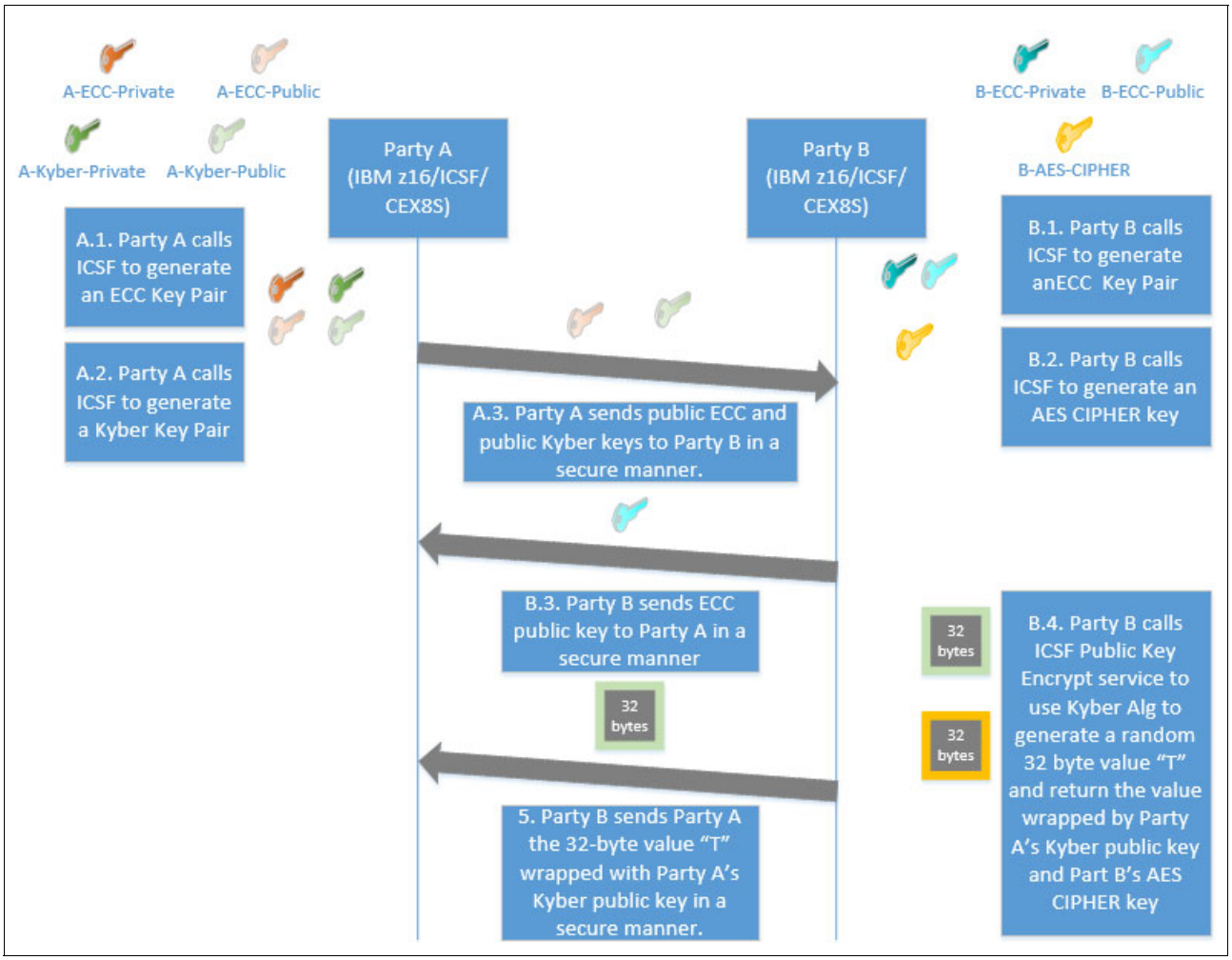



*Figure 3-3 Hybrid key agreement process - CCA example*

## 3.2.3 Industry applications

This section describes how organizations across many different industries with a need to share encryption keys securely with partners by using traditional key exchange mechanisms might be vulnerable to CRQC attacks.

It also outlines how they can benefit from the hybrid key exchange schemes that are provided with IBM z16 and later to ensure data confidentiality, even in the presence of sufficiently capable quantum computers.

### Pharmaceuticals

The pharmaceutical industry is an example of the need to securely share intellectual property and research findings with Business Partners to drive success. The process of developing new drugs and making them publicly available in the marketplace is lengthy and costly.

On average, it takes at least 10 years for a new drug to become available in the market, with six to seven of those years being dedicated to performing clinical trials. A study that was conducted by Tufts Center for the Study of Drug Development estimated that it costs pharmaceutical companies $2.6 billion to develop a new medication gaining marketing approval.

Pharmaceutical companies that are vying to successfully complete clinical trials for their new drug have only a 12% success rate. Even if a newly developed drug passes all required steps to bring it to market, pharmaceutical companies face heavy competition with each other.

To create efficiencies in the drug development process to lower costs and increase the likelihood that a drug passes clinical trials, pharmaceutical companies can benefit significantly from collaborating with each other and other technology organizations. An example of this multi-party collaboration within the pharmaceutical industry was the agreement between the drug companies to co-develop vaccines for respiratory diseases and make it available for use after FDA approvals.

Part of this collaboration requires the need to safely share intellectual property and research findings with each other. Their data must be protected from unintended disclosure by competitors and adversaries to avoid reputation damage, profit loss, and brand impact.

Although the rise of quantum computing undoubtedly benefits pharmaceutical companies regarding drug development, it also brings about challenges for them about collaborating with each other and sharing their intellectual property in a secure manner.

If an adversary with access to a quantum computer can break the encryption algorithms that are used to protect shared information, they can publicly expose a pharmaceutical company's research findings that can result in a significant advantage for their competitors to release a similar drug.

By taking advantage of the quantum-safe capabilities with IBM z16 and later, these organizations can use the benefits that quantum computers can provide in Research and Development while also ensuring that intellectual property that is shared between companies remains secure.

### Government bodies

Government bodies often partner with other government agencies and the private sector to create efficiencies in their operations. For example, information sharing programs are used that are dedicated to saving government resources through partnerships between various federal, state, and municipal government agencies. As part of these information sharing programs, the previously mentioned agencies are entitled to exchange taxpayer information with each other.

This example the need to facilitate the sharing of confidential information between partnering agencies. Although safeguards exist to protect the confidentiality of taxpayers' information, the threat of quantum computing presents challenges for maintaining this confidentiality in the future.

Government bodies greatly benefit from the capabilities that are provided with IBM z16 and later to enable secure information sharing with partners. By using quantum-safe hybrid key exchange schemes, government agencies can maintain the confidentiality of their citizens' information as it is shared to ensure that not even a quantum computer can intercept the information and break the encryption algorithms that are used to protect it.

### Banking

In financial services, such as banking, a significant need exists to protect business-critical data in the face of quantum computing. Although many examples exist of data that requires protection in banking, a simple example to illustrate is in the case of credit card information.

Banks are required to comply with the Payment Card Industry Data Security Standard (PCI DSS) for protection of customer's sensitive payment card information. As a result, security protocols are in place to protect their sensitive data as it is transmitted between point-of-sale (POS) terminals and the bank. This process typically is done by encrypting the data that is in transit between the two parties.

PCI DSS implies is that both parties must have a shared symmetric key that is used to encrypt the customer data at the POS and then decrypt it when in the hands of the bank. Even when quantum-safe symmetric key algorithms are used, such as AES for this encryption and decryption, the problem is that the shared keys are typically exchanged through classical public key cryptography schemes alone that are not quantum-safe. For example, they might use ECDH as their key agreement protocol to derive their symmetric encryption key.

As a result, the protection of their customers' sensitive payment card information can potentially be exposed if a cybercriminal harvests the encrypted data now to decrypt in the future after they obtain access to a quantum computer of sufficient scale.

Therefore, banks and other organizations that deal with sensitive payment card information can benefit greatly by using the hybrid key exchange schemes that are available with IBM z16 and later. By doing so, they securely derive a shared encryption key that cannot be derived in the face of high-powered quantum computers.

### Automotive and aerospace

Similar to the pharmaceutical industry, automotive and aerospace companies must protect their intellectual property from unintended information disclosure from their competitors to avoid reputation damage, profit loss, and brand impact. They also must collaborate with Business Partners to develop the latest cutting-edge capabilities in their respective industries.

To enable this collaboration, they must share their intellectual property and proprietary research findings with each other in a way that is secure and cannot be exposed to bad actors and competitors. This sharing also is done by way of encrypting confidential data in transit by using mutually derived encryption keys.

Because these keys are shared between partnering organizations by using traditional public key cryptography alone, it can be concluded that quantum computers eventually can expose the organizations' confidential information that is shared between parties by breaking the traditional public key algorithms that are used today.

Therefore, the quantum-safe key exchange mechanisms that are available with IBM z16 and later can be used. As a result, automotive and aerospace companies can be assured that the intellectual property they are exchanging with partners is never exposed in the clear to avoid unintended information disclosure.

## 3.3 Message integrity and secure logging

This use case addresses the problem of ensuring the integrity of messages and logs in the presence of quantum computers and how capabilities that are provided with IBM z16 and later can solve these challenges.

Organizations that send messages and digital assets to other parties, such as legal documents, audit logs, financial statements, and historical records, traditionally ensured the integrity of their messages by using digital signatures.

By using digital signatures, organizations can ensure that the receiver of a message or document knows whether the message is genuine and untampered. This feature makes it possible to prevent legal and documentation fraud.

However, with the rise of quantum computing, adversaries with access to a quantum computer of sufficient scale can undetectably rewrite history and modify documents, messages, and logs, and claim that the tampered message was indeed the original message the organization sent.

Although digital signatures can serve multiple purposes, the focus of this use case is on message integrity. Digital signatures can also help achieve identity authentication, or proof of authorship, as described in 3.4, “Proof of authorship” on page 46.

### 3.3.1 Problem statement

In the past, organizations relied on digital signatures to ensure the integrity of messages, digital documents, logs, and audit reports. Traditionally, when an organization sent a digitally signed message or document to another party, the other party verified the digital signature to ensure that the document or message was genuine and was not modified by an adversary. By using digital signatures, organizations avoided advanced legal and documentation fraud.

Digital signatures rely on public key cryptography. ECDSA is one example of a popular cryptographic algorithm that is used for digital signatures.

For an organization to send a message or document to another party and ensure it was not tampered with in-transit, they digitally sign the document by using ECDSA, for example. The process begins with the organization generating an ECC key pair (consisting of a private and public key). They then use the ECDSA signing algorithm to generate a digital signature.

As part of the algorithm’s signing process, a cryptographic hash of the message, document, or log is first calculated by using a hash algorithm, such as SHA-256. Because the hash generation is a one-way function, meaning the hash value is unique to the exact contents of the message or document, any effort to modify the message or document results in a different, inconsistent hash value.

Following the hash value generation, it is signed by using the organization’s private key, to which only they have access. The signed hash is then appended to the document or message and is sent to the party it was intended for, along with the organization’s public key.

The retrieving party can verify the integrity of the message or document by using ECDSA and the public key that was shared:

1. The party generates their own hash of the document or message by using the same hash function that was used to sign it.
2. They then use the shared public key to verify the signed hash that was appended to the document or message.
3. As part of the ECDSA verification process, the newly computed hash value is compared with the decrypted hash value that was appended to the document.

   If the two hashes match, the party can be assured that the message was not tampered with in-transit and the contents of the message or document are exactly what the original organization sent.

If the hashes do not match, the receiving party can conclude that the message was tampered with because the hash that they generated differs only from the appended hash if the message's contents was changed in-transit.

When digital signatures are used, the receiving party can reliably determine the integrity of the message because they assume that only the original organization includes the mathematically related private key that was used to sign the hash and nobody else. This assumption can be made because of the mathematical complexity of the elliptic-curve discrete logarithm problem.

The only way for an adversary to obtain the original organization's private key is to derive it from the associated public key by solving this mathematical problem. However, it is not feasible for a conventional computer to solve this problem because it likely takes millions of years. As a result, it traditionally was accepted that the appended signed hash is the hash value that was generated from the original message or document. Therefore, the integrity can be determined by comparing the receiver's hash output with the appended hash value.

With the rise of quantum computing, eventually it will be possible for a quantum computer of sufficient scale to solve this once considered complex problem in only a matter of hours by using Shor's algorithm.

An adversary with access to a quantum computer might take an entity's public key and derive the associated private key. After it is in the hands of an adversary, they can alter the contents of the message or document undetectably.

When the organization sends the message to the intended party, the adversary can run a "man in the middle" attack to intercept the original message, modify its contents, re-create a new hash value, and then, sign this new hash by using the organization's private key. This newly signed message is then forwarded to the intended party. The party has no way of ensuring its integrity because the hash value that was generated is identical to the newly appended hash value.

This ability to undetectably modify messages, documents, and logs by using quantum computers can result in disastrous consequences, especially when considering the various regulations, such as eIDAS (in the EU), and UETA and E-SIGN (in the US) that allows digital signatures to have equal legal status to traditional "wet" signatures. In addition, adversaries might tamper with potentially life-saving messages that are used in the automobile industry and messages that are used for international government communication.

### 3.3.2 Solving the integrity challenge with IBM Z capabilities

The challenge of ensuring message, document, and log integrity in the face of quantum computing can be alleviated by using the dual digital signature schemes that are made possible with IBM z16 and later. For more information about for the procedures that are used to enable these dual digital signature schemes, see 6.6, "Quantum-safe digital signatures" on page 122.

By using this IBM z16 and later capability, organizations can ensure message integrity by digitally signing their messages and documents by using a classical cryptographic algorithm and a quantum-safe cryptographic algorithm.

For example, a dual digital signature can be generated by using a classical cryptographic algorithm, such as ECDSA, with a quantum-safe cryptographic algorithm, such as ML-DSA. IBM z16 and later enables this through enhancements in the Crypto Express8S (CEX8S) features to provide quantum-safe algorithm APIs.

As in the past, it is important to continue the use of a classical cryptographic signature algorithm, such as ECDSA, to comply with the various standards and compliance requirements mandating the use of approved signing algorithms. However, with the use of a classical cryptographic algorithm for digital signatures, it is also crucial to begin safeguarding information with a quantum-safe cryptographic scheme to ensure that message integrity is maintained in the face of CRQC attacks.

To ensure that message integrity is maintained in the future, an organization can use IBM z16 and later to digitally sign their out-bound messages and documents by using two digital signatures. This process is explained next (for more information, see 6.6, “Quantum-safe digital signatures” on page 122).

An organization that wants to ensure the integrity of its messages begins by generating a hash of its message by using a cryptographic hash algorithm, such as SHA-256.

Then, the organization generates two key pairs. For example, they might generate an ECC public key pair and a quantum-safe ML-DSA key pair (ICSF supports the ML-DSA signature algorithm on the CCA and PKCS #11 APIs for IBM z16 and later).

The organization then signs this hash with ECDSA by using their ECC private key, which results in an ECC signature. With generating that ECC signature, the organization also signs the same hash by using their ML-DSA private key to generate a quantum-safe signature. The result is that two signatures exist for the single hash of the message, and both signatures are appended to the message for verification purposes.

Anyone who wants to verify that the message was not modified or tampered with in-transit uses the ML-DSA public key to verify the ML-DSA signature and the ECC public key to verify the ECC signature. As part of the verification process, the receiving party generates the hash value of the message and passes the hash value and the suitable digital signature to the signature verification function for each signature algorithm. If the ECC signature verifies and the ML-DSA signature verifies, the party can be assured that the message was not tampered with by a bad actor, and thus, message integrity is maintained.

As shown in Figure 3-4 on page 44, success indicates message integrity and authorship by the owner of the ML-DSA public key (denoted by Dilithium).

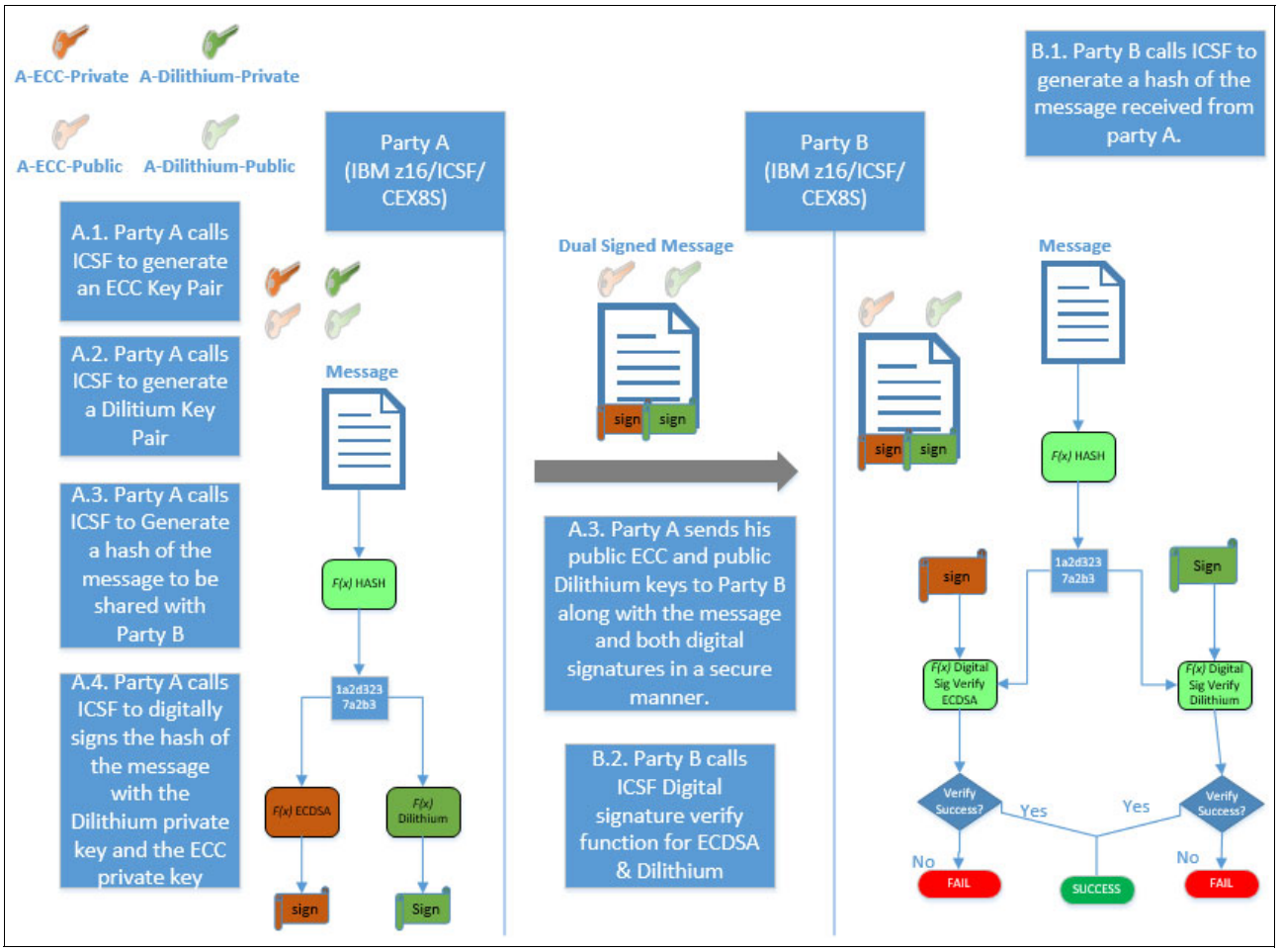



*Figure 3-4 Validating the authenticity (proof of authorship) and integrity of a message*

This dual digital signature scheme that is enabled with IBM z16 and later solves the challenge of ensuring message integrity, even if an adversary can access a quantum computer that can run Shor's algorithm.

Suppose that the adversary ran a man-in-the-middle attack to intercept the original message, modify its contents and then, re-create a new hash value. Although they can sign the new hash by using the organization's ECC private key that they derived by using Shor's algorithm, they have no way of signing the same hash by using the organization's ML-DSA private key (denoted by Dilithium), because it is quantum-safe.

A single ECC signature is of no value because the integrity verification works only if a valid ECC signature and a valid ML-DSA signature with the message are available.

## 3.3.3 Industry applications

This section describes the wide range of applications that quantum-safe digital signatures have across different industries. Also discussed is how financial services organizations, automotive companies, software vendors, and even law firms might all benefit from the dual digital signature schemes that are enabled with IBM z16 and later to ensure that their communications and data were not tampered with or targeted by adversaries with access to quantum computers.

## Banking

In the banking industry, digital signatures prove to be a vital necessity in securing financial transactions and messages between banks. Society for Worldwide Interbank Financial Telecommunication (SWIFT) was created in the 1970s to enable banks across the world to share information about financial transactions with each other. These high-value electronic transactions and messages can be shared between banks over the SWIFT Network, which relies on public key infrastructure to digitally sign and encrypt messages.

In credit card payments, EMV (Europay, MasterCard, and Visa) transactions and messages can be shared between banks and merchants through the Global Payment Network, which relies on public key infrastructure for digital signatures and encrypted messages.

The growing presence of quantum computing throughout the world threatens the long-term security of these types of banking and credit card payment transactions. Relying on traditional public key cryptography alone for securing these interbank transactions poses a vulnerability in message integrity because adversaries with access to quantum computers can inevitably forge transactions and messages.

The banking industry might also be one of the first targets of CRQC attacks because of the profit potential for cybercriminals. Adopting quantum-safe dual digital schemes for credit card payments and in the banking industry is a must to secure interbank financial transactions in the future.

## Automotive industry

In the automotive industry, manufacturers are constantly innovating new ways of improving driver safety. One of these innovations is the rise of vehicle-to-vehicle (V2V) communication that aims to make driving safer by allowing vehicles to communicate with each other to prevent crashes, traffic, and so on.

Quantum computing poses security challenges for this V2V communication in which messages between vehicles might be tampered with by bad actors. Messages that are transmitted between vehicles that are digitally signed by using traditional public key cryptography alone, such as RSA or ECDSA, no longer guarantee the integrity of these V2V messages when quantum computers of sufficient scale are available to bad actors.

Adversaries might manipulate messages to create life-threatening situations on highways and in busy cities. By taking advantage of the dual digital signing schemes that are available with IBM z16 and later, vehicle manufacturers can use quantum-safe digital signing to thwart message tampering attacks and make driving safer.

## Trusted legal documents

In the legal industry, regulations, such as eIDAS (in the EU) and UETA and E-SIGN (in the US) allow digital signatures to have equal legal status to traditional “wet” signatures.

Historically, trusted legal documents that use digital signature capabilities were not forged because the integrity of these documents was ensured through verification of the digital signatures by the receiving party.

However, quantum computers that can quickly derive the private keys serving as the underpinnings of these digital signatures eventually will make it possible to manipulate and tamper with legal documents by forging digital signatures.

To avoid advanced legal fraud and impersonation attacks in this era where digital signatures serve as legal evidence, it is critical that legal departments or legal firms begin safeguarding

their legal documents and communications with quantum-safe dual digital signature schemes that are available with IBM z16 and later.

Trusted legal documents with time stamping or notary systems provide evidence that an event or transaction happened at a particular time. Other use cases of trusted legal documents are email signing using an asymmetric encryption algorithm (for example, RSA) to sign an email to ensure the integrity of the email content. These trusted legal documents must use quantum-safe dual digital signature schemes to verify integrity and non-repudiation.

## 3.4 Proof of authorship

Organizations that made their valuable assets digitally accessible, such as legal documents, financial statements, historical records, and license code, often ensured nonrepudiation such that the public can trace back these digital assets to the original creator.

Proof of authorship was achieved by using digital signatures where anyone can verify the authenticity of a digitally signed document or digital asset. Digital signatures also make it impossible for any unknown entity to disseminate information or assets to the public and claim it came from someone else, which ensures responsibility for the distribution of digital information.

In the legal industry, various regulations are in place that allow digital signatures to have equal legal status to traditional “wet” signatures. Because legal documents might feature long lifetimes, the signatures on them might need to be secure for decades.

With the rise of quantum computing, adversaries with access to a quantum computer of sufficient scale can manipulate legal history by forging digital signatures. As a result, the legal underpinnings of digitalization are now vulnerable.

This use case illustrates how organizations can avoid advanced legal fraud and impersonation attacks by using the quantum-safe capabilities that are available on IBM z16 and later.

### 3.4.1 Problem statement

Organizations traditionally made their digital assets public while ensuring proof of authorship and nonrepudiation. Anyone accessing or viewing these digital assets can verify exactly from who it came, which is made possible by using digital signatures.

Nonrepudiation and proof of authorship are ensured because these digital signatures are generated by using a private key that is only known by the signer. As a result, the signer has no means of repudiating their signature that is added to the document.

If a dispute occurs, proof of authorship enables the document’s authorship to be supported by evidence, being the holder of the private key.

#### Digital signatures

Digital signatures are made possible with the use of public key cryptography. One example of a popular cryptographic algorithm for digital signatures is ECDSA. Organizations that wanted to digitally sign a document so that the public can trust that it came from them begins by generating an ECC key pair that consists of a private and public key.

They then use the ECDSA signing algorithm to digitally sign their document. The algorithm works by first calculating a cryptographic hash of the document by using a cryptographic hash algorithm, such as SHA-256. This hash is unique to the exact contents of the document.

If the document was modified, the resulting hash output is different. As part of the signing algorithm, the hash value then is signed by using the signer's private key, which only they can access. The signed hash is appended to the document and made publicly available along with the signer's public key.

Now that the document is digitally signed, anyone who wants to access the document can verify its authenticity by using ECDSA and the public key. The receiver uses the public key that they received to verify the signed hash that is appended to the document.

They also generate a hash output of the same document by using the same hash function that was used to sign the document. The algorithm then compares the hash value that the receiver computed with the decrypted hash value that was appended to the document. If the two hashes match, the public key sender is verified as the true author of the document because it is assumed that only they have the mathematically related private key.

### Elliptic-curve discrete logarithm problem

The strength of these digital signatures for proving authorship of a digital asset is a result of the mathematical complexity of the elliptic-curve discrete logarithm problem. The only way to derive the private key from the associated public key is to solve this mathematical problem, which is not feasible for conventional computers because it likely takes millions of years to solve the elliptic-curve discrete logarithm problem.

Therefore, it was always assumed that the private key holder is the author of the digitally signed document, which is how proof of authorship can be ensured. As a result, public key cryptography alone was considered secure for digital signatures.

With the rise of quantum computing, it eventually will be possible for a quantum computer of sufficient scale to solve this once considered complex problem in only a matter of hours by using Shor's algorithm. An adversary with access to a quantum computer might take an entity's public key and derive the associated private key, which can have disastrous consequences.

By using that entity's private key, the adversary can forge their digital signature to manipulate legal history and claim that the digital assets they create were authored by the entity whose private key they possess.

Making matters worse, many legal documents (and the signatures on them) are in public records; therefore, an adversary can easily obtain them and attack the cryptography with a quantum computer. No "hacking" is necessary to get to the data to be attacked. Documents, messages, certificates, software, and transactions all can be forged. Therefore, an identity over the internet is no longer guaranteed.

## 3.4.2 Solving this challenge with IBM Z capabilities

Capabilities that are provided by IBM z16 and later allow organizations to solve this challenge that is faced today with the threat of quantum computing by using dual digital signature schemes and quantum-safe key generation. For more information about enabling these dual digital signature schemes, see 6.6, "Quantum-safe digital signatures" on page 122.

With IBM z16 and later, it is possible for organizations to digitally sign their valuable assets by using a classical cryptographic algorithm and a quantum-safe cryptographic algorithm. The

ability to use a quantum-safe cryptographic algorithm, such as ML-DSA, with a classical cryptographic algorithm, such as ECDSA, is supported by the quantum-safe algorithm APIs within the new Crypto Express cards (CEX8S) that was provided with IBM z16 and later.

The value in the use of this dual digital signature scheme is two-fold: First, as in the past, it is important to use a classical cryptographic signature algorithm, such as ECDSA, because various standards and compliance requirements exist that require standards-approved signing algorithms.

Uncertainty also exists about how those standards might change in the future with the rise of quantum computing. Although newly developed algorithms might be considered “quantum-safe,” it is never guaranteed that these algorithms are not to be found insecure.

Therefore, as a minimum security measure, it is beneficial to continue the use of these classical signature schemes to thwart attacks from conventional computers. Also, to begin safeguarding valuable information in the future, even in the face of high-powered quantum computers, it is important to buttress this protection with a known quantum-safe cryptographic signature scheme with IBM z16 and later.

To maintain proof of authorship and nonrepudiation for an organization’s digital assets, they use IBM z16 and later to create two digital signatures that public entities can verify (see Figure 3-4 on page 44):

1. A hash of their document is generated by using a cryptographic hash algorithm, such as SHA-256.
2. The organization generates two key pairs. For example, they might generate an ECC public key pair and a ML-DSA key pair (ICSF supports the ML-DSA signature algorithm on the CCA and PKCS #11 APIs for IBM z16 and later).
3. The organization signs this hash with ECDSA; for example, by using their ECC private key, which results in an ECC signature. Along with generating that ECC signature, the organization also signs the same hash by using their ML-DSA private key to generate a quantum-safe signature.

   This process results in having two signatures for the single hash of the message, and both signatures are appended to the digital document for verification.

Anyone who wants to verify that this organization authored the document uses the ML-DSA public key to verify the ML-DSA signature and the ECC public key to verify the ECC signature. If both signatures are successfully verified, the verifying party can be assured that the document was authored by the organization.

The threat that is created by quantum computers of sufficient scale to forge digital signatures is negated because both signatures must verify.

Even if an adversary with access to a quantum computer can run Shor’s algorithm to derive the ECC private key by using the related public key, they cannot forge the organization’s signature because they cannot derive their ML-DSA private key.

Suppose a cybercriminal with access to a quantum computer can create a fake document and sign it with the derived ECC private key. The ECC signature is created, but the adversary does not have the ML-DSA private key to create a ML-DSA signature.

A single ECC signature is of no value because the verification works only if a valid ECC signature and ML-DSA signature exists with the document.

For more information, see 6.6, “Quantum-safe digital signatures” on page 122.

## 3.4.3 Industry applications

This section describes the wide range of applications that quantum-safe digital signatures have across different industries. It also discusses how financial services organizations and software vendors might all benefit from the dual digital signature schemes that are enabled by IBM z16 and later to ensure that their communications and data were not tampered with or targeted by adversaries with access to quantum computers.

### Software development and application code-signing

In software development and application code-signing, vendors or IT organizations aim to secure software distribution using digital signatures based on public key cryptography.

For example, updates to a software code release are often digitally signed using RSA or ECDSA encryption algorithms to ensure that customers can verify whether the software code was tampered with before installation. Often, these software codes or patches require automatic digital signature verification before installing the update.

The traditional algorithms used today for digitally signing software were reliable in ensuring its integrity; however, with the rise of quantum computing, integrity is no longer guaranteed because bad actors can tamper with the software code or patches undetectably.

This tampering can result in a customer installing malware on their systems. Therefore, it is critical that software vendors or software development department in the IT organization implement quantum-safe dual digital signature schemes and pre-hash for their code and patches to authenticate the origin of software updates.

### Digital assets tokenization and cryptocurrency

Digital asset tokenization, a promising application of Distributed Ledger Technology (DLT), is reshaping the financial landscape. It digitally represents assets like bonds and gold on a distributed ledger, opening up new possibilities for transactions, trading, and fractional ownership. Distributed ledgers use hash functions, digital signatures, and cryptographic keys to secure and authenticate the recorded data and transactions.

Banks are proactively exploring ways to mitigate the risk of security and integrity of tokenized digital assets and crypto currencies (Bitcoins, Ethereum, and so on) with distributed ledger technology (DLT) and digital wallets. This future-proofing approach addresses the potential cybersecurity threats posed by a powerful quantum computer, ensuring the security of digital assets.

HSBC has demonstrated its leadership in the field by successfully achieving the interoperability of DLT nodes and the exchange of digital gold tokens. They used an encrypted Post-Quantum Cryptography-Virtual Private Network (PQC-VPN) connection to transmit payment messages and sign transactions. Their encrypted PQC-VPN tunnel approach effectively mitigated the risk of slower latency, as the encryption process occurs outside the DLT systems while maintaining transaction processing speed. (Reference: HSBC document: Asset tokenization in the Quantum Age)

The Open Quantum Safe (OQS) project is playing a crucial role in the industry by developing an Open SSL implementation and Open VPN integration that supports Post-Quantum Cryptography (PQC) algorithms (ML-KEM as the key encapsulation mechanism and ML-DSA as the signature algorithm from the NIST PQC Standards). The OQS project is a significant step in developing quantum-safe solutions for OpenSSL. (Reference: https://github.com/open-quantum-safe)

As PQC-VPN and PQC algorithms are used in the DLT architecture, banks will have a new world of opportunities for digital asset tokenization and digital wallets for cryptocurrency

4

# Getting ready for quantum-safe cryptography

In this chapter, we introduce the cryptographic components that are available on the IBM Z platform.

We also describe an approach to help discover and classify data, establish a cryptographic inventory by using various tools, and adopt quantum-safe cryptography on IBM Z.

Discussions about best practices, mitigation options, and encryption key management tools are also included.

This chapter includes the following topics:



# 4.1 IBM Z cryptographic components overview

The cryptographic stack on the IBM Z platform consists of many different hardware and software components that provide unique capabilities to aid in securing your environment. In this section, the components that are available for quantum-safe cryptography are briefly described. We also review the required levels of hardware and software components to implement quantum-safe cryptography in your IBM Z environment.

## 4.1.1 IBM Z cryptographic hardware components

On the hardware side, quantum-safe cryptography is supported by the Crypto Express hardware security module (HSM) and the Central Processor Assist for Cryptographic Functions (CPACF).

Cryptographic functions can be categorized in the following groups from an application program perspective:

- Symmetric cryptographic functions and hashing functions are provided by CPACF or Crypto Express features
- Asymmetric cryptographic functions and digital signatures are provided by Crypto Express features, while some are also provided by CPACF

For more information about the different types of cryptography, see 1.2.1, “Cryptography overview” on page 4.

### CPACF

Each processor unit (PU) chip in the IBM Z platform has an independent cryptographic engine (known as a cryptographic assist). CPACF is a high performance, low-latency coprocessor that performs symmetric key encryption operations and calculates message digests (hashes) in hardware.

The following algorithms are supported:

- Advanced Encryption Standard (AES) for 128-bit, 192-bit, and 256-bit keys
- Data Encryption Standard (DES) and Triple Data Encryption Standard (TDES)
- Hashing algorithms:
  - Secure Hash Algorithm (SHA)-1
  - SHA-2
  - SHA-3
  - SHAKE

CPACF supports Elliptic Curve Cryptography (ECC) clear key, which improves the performance of Elliptic Curve (EC) algorithms. The following algorithms are also supported:

- EdDSA (Ed448 and Ed25519)
- ECDSA (P-256, P-384, and P-521)
- ECDH (P-256, P-384, P521, X25519, and X448)
- Support for protected key[1] signature creation

z/OS Integrated Cryptographic Services Facility (ICSF) uses CPACF to accelerate cryptographic functions. For ICSF to use these functions, Feature Code (FC) 3863 must be enabled. This FC is not enabled by default.

[1] A protected key is a data-encrypting key that is encrypted by a CPACF wrapping key and used within the IBM Z platform.

IBM z16 and later includes counters for CPACF to track cryptographic algorithms, bit lengths, and key security. The CPACF counters provide evidence for compliance (which cryptography is used), performance (frequency of cryptography use), and configuration (proof of change).

For more information, see “IBM ICSF cryptographic usage tracking” on page 63, and “Formatting cryptographic usage statistics records” on page 91.

## Crypto Express

Each Crypto Express HSM contains cryptographic engines that can be configured as a Common Cryptographic Architecture (CCA) cryptographic coprocessor (CEX*n*C)[2], as an Enterprise Public Key Cryptography Standard #11 (PKCS #11) cryptographic coprocessor (CEX*n*P), or as an accelerator (CEX*n*A) for public key and private key cryptographic operations that are used with SSL/TLS processing.

Crypto Express coprocessors enable secure key generation and operations under the direction of ICSF. It is recommended that at least two of each type (CCA coprocessor or Enterprise PKCS #11 coprocessor or accelerator) be configured for redundancy.

If one coprocessor must be taken offline, the second coprocessor that is loaded with same master keys[3] can handle new requests.

On IBM Z, up to 60 Crypto Express coprocessors can be configured, each supporting up to 85 cryptographic domains[4]. Each domain is protected by a master key, which prevents access across domains and effectively separates the contained keys.

Quantum-safe algorithms are supported by the Crypto Express7S and Crypto Express8S coprocessors. For more information about the supported algorithms for each coprocessor type, see 4.1.3, “Minimum hardware and software for quantum-safe cryptography support” on page 59.

## Secure boot technology

In addition to the quantum-safe cryptography support in the CPACF and Crypto Express features, IBM z16 and later secure boot technology uses quantum-safe and classical digital signatures to perform a hardware-protected verification of the Initial Machine Load (IML) firmware components. This firmware integrity protection is anchored in a hardware-based Root of Trust (RoT) to ensure that the system starts securely by keeping unauthorized firmware (or malware) from taking over during start.

## Trusted Key Entry

As an option, a Trusted Key Entry (TKE) Workstation can be used to securely manage Crypto Express HSMs on IBM Z. Changes to HSM administrative settings, including changes to the master keys, can be sent from a TKE Workstation to multiple Crypto Express HSMs in one management event. A TKE Workstation is required to manage administrative settings for Crypto Express HSMs in Enterprise PKCS #11 (EP11) mode or a compliant-mode domain for Crypto Express HSMs in CCA mode. The TKE feature contains a combination of hardware, firmware, and software. The secure administration of an HSM using the TKE Workstation requires smart card readers and smart cards. These parts are obtained through additional features associated with the TKE product.

[2] *n* is a 7 or 8, which represents a Crypto Express7S feature or Crypto Express8S feature, respectively.

[3] A master key is a special key-encrypting key (KEK) that is in a tamper-responding, Crypto Express adapter. A master key sits at the top level of a KEK hierarchy.

[4] A domain acts as an independent cryptographic device with its own master key. Domains in the same Crypto Express are isolated. Domains are assigned to IBM Z logical partitions (LPARs).

With TKE 10.0, quantum-safe encryption algorithms are used for key exchange when loading keys to Crypto Express HSMs that are running in CCA mode.

For more information, see “Trusted Key Entry” on page 73.

Figure 4-1 shows the hardware components for implementing quantum-safe cryptography in an IBM Z environment.

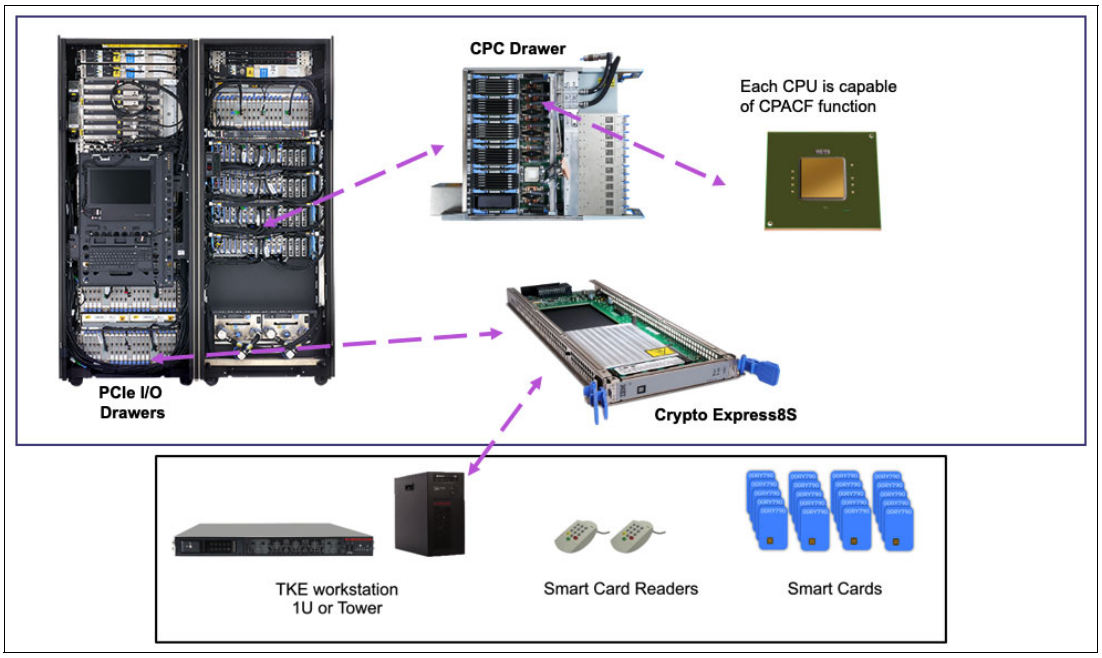



*Figure 4-1 IBM Z cryptographic hardware components*

For more information about the cryptographic hardware components, see I*BM z17 (9175) Technical Guide*, SG24-8579 and *IBM z16 (3931) Technical Guide*, SG24-8951.

## Crypto Express features

Crypto Express8S and Crypto Express7S features provide quantum-safe RoT and quantum-safe cryptographic APIs for application program use (see Table 4-1).

*Table 4-1 Crypto Express features for quantum-safe cryptography*

| Feature | Description |
|---|---|
| Crypto Express8S Dual-HSM (FC 0908) | This feature contains two IBM 4770 PCIe cryptographic coprocessors, which can be independently defined as a coprocessor or an accelerator. Supported on IBM z17 and IBM z16. |
| Crypto Express8S Single-HSM (FC 0909) | This feature contains one IBM 4770 PCIe cryptographic coprocessor, which can be defined as a coprocessor or an accelerator. Supported on IBM z17 and IBM z16. |
| Crypto Express7S Dual-HSM (FC 0898) | This feature contains two IBM 4769 PCIe cryptographic coprocessors, which can be independently defined as a coprocessor or an accelerator. Supported on IBM z17 IBM z16, and IBM z15. |
| Crypto Express7S Single-HSM FC 0899) | This feature contains one IBM 4769 PCIe cryptographic coprocessor, which can be defined as a coprocessor or an accelerator. Supported on IBM z17, IBM z16 and IBM z15 |

### *Crypto Express8S*

When the Crypto Express8S adapters are configured in CCA or EP11 mode, the following support for quantum-safe symmetric algorithms (AES, CMAC, and HMAC), hashing algorithms (SHA-2, SHA-3), and digital signature algorithms are enabled by using:

- ML-DSA & Pre-hash ML-DSA 4,4 (NIST FIPS 204)
- ML-DSA & Pre-hash ML-DSA 6,5 (NIST FIPS 204)
- ML-DSA & Pre-hash ML-DSA 8.7 (NIST FIPS 204)
- CRYSTALS-Dilithium 6,5 (Round 3)
- CRYSTALS-Dilithium 8,7 (Round 3)
- CRYSTALS-Dilithium 6,5 (Round 2)
- CRYSTALS-Dilithium 8,7 (Round 2)

Also, the following quantum-safe key encapsulation mechanisms (KEM) are supported:

- Hybrid key agreement scheme combining Elliptic Curve Diffie-Hellman (ECDH) and ML-KEM (768 or 1024), or CRYSTALS-Kyber (1024)
- ML-KEM 768 (NIST FIPS 203), implemented as a standalone KEM
- ML-KEM 1024 (NIST FIPS 203) is implemented as standalone KEM

### *Crypto Express7S*

When the Crypto Express7S adapters are configured in CCA or EP11 mode, the support for quantum-safe symmetric algorithms (AES, CMAC, and HMAC), hashing algorithms (SHA-2, SHA-3), and digital signature algorithms are enabled through CRYSTALS-Dilithium 6,5 (Round 2).

For more information about quantum-safe algorithms, see “Algorithms to counter CRQC attacks” on page 9.

## Hybrid key exchange mechanism

The use of secure keys[5] and protected keys in the IBM Z encryption process ensure that data-encrypting keys are not visible to unauthorized callers.

When first created, the data-encrypting key is wrapped (encrypted) as a secure key by ICSF by using a master key, which is stored in the hardware security module (HSM) of an assigned Crypto Express adapter when configured in CCA mode.

In IBM z16 and later firmware, a hybrid key exchange mechanism that includes CRYSTALS-Kyber, CRYSTALS-Dilithium, and ECDH is used to securely negotiate a shared transport key (AES 256-bit key) between the Crypto Express8S HSM and the CPACF. The shared transport key is used to protect data-encrypting keys that are sent from the Crypto Express8S HSM to the CPACF as part of the runtime process to create protected keys.

A CPACF wrapping key is used to rewrap a data-encrypting key as a protected key. The protected key is sent to ICSF for use by authorized callers. The CPACF wrapping key is in a protected area of the hardware system area (HSA) of the IBM Z platform, which is not accessible to the operating system, applications, or users (see Figure 4-2 on page 56).

[5] A secure key is a data-encrypting key that is encrypted by a master key or key-encrypting key and never appears in clear text that is outside of a secure environment, such as a tamper-responding HSM, or IBM Z firmware. Secure keys can be stored in an ICSF key data set or returned to the ICSF caller.

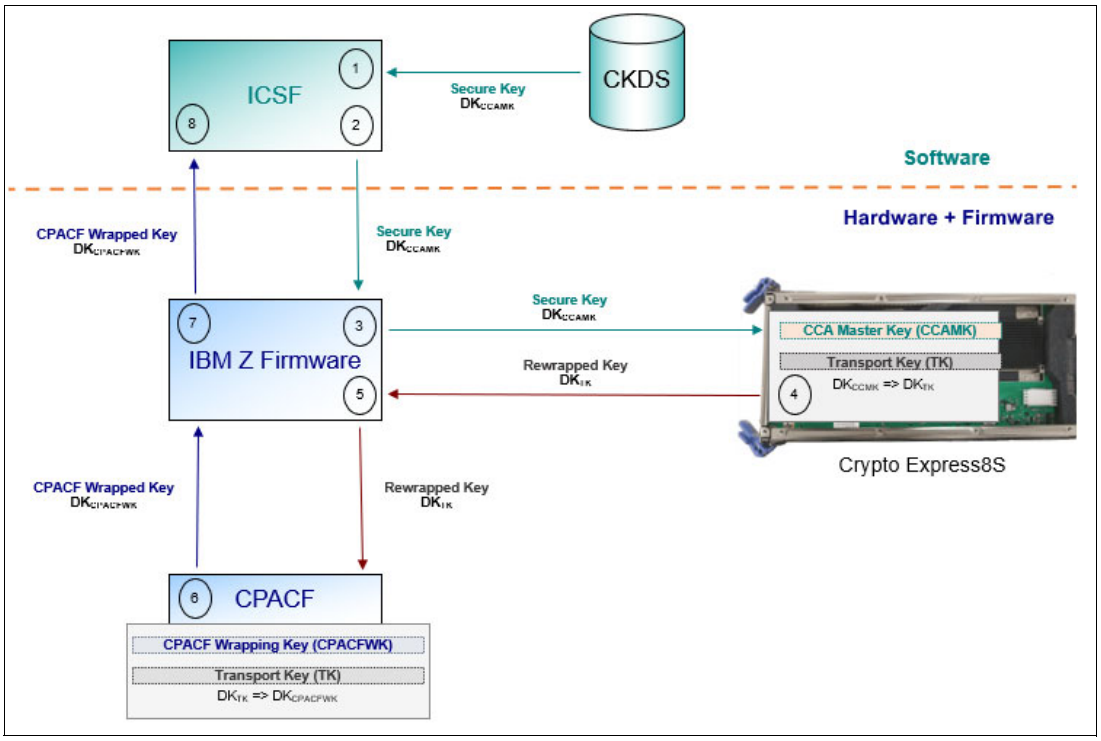



*Figure 4-2 IBM z16 (and later) and Crypto Express8S process to create protected keys*

The following process is used to create a protected key from a secure key, as shown in Figure 4-2:

1. ICSF retrieves the data-encrypting key (DK) that is stored in the CKDS as a secure key (encrypted by using a master key [CCAMK]).
2. ICSF starts the process by sending the secure key ($DK_{CCAMK}$) to IBM Z firmware.
3. IBM Z firmware sends the secure key to the suitable domain in the Crypto Express8S HSM.
4. Crypto Express8S HSM decrypts the secure key by using the master key and rewraps the data-encryption key by using a transport key (TK). The transport key is derived from two independent contributions of entropy as part of the hybrid key exchange mechanism:
   a. The ECDH calculation of the “Z” shared secret[6], by using the private key of the IBM Z firmware and the public key of the Crypto Express8S HSM that is in the CPACF, and the corresponding public and private keys that are in the Crypto Express8S HSM.
   b. The random number that is generated in the Crypto Express8S HSM is sent encrypted under the CRYSTALS-Kyber public key to the CPACF, which is then signed by a CRYSTALS-Dilithium private signing key.
5. The rewrapped data-encrypting key ($DK_{TK}$) is sent back to IBM Z firmware.
6. IBM Z firmware starts CPACF to unwrap and rewrap the data-encrypting key by using a CPACF wrapping-key (WK) to create a protected key ($DK_{CPCFWK}$).
7. IBM Z firmware returns the protected key ($DK_{CPCFWK}$) to ICSF.
8. ICSF caches the protected key in its address space and optionally returns the protected key to the authorized caller.

[6] Known only to the entities involved in a communication. Possession of that shared secret can be provided as proof of identity for authentication.

### 4.1.2 IBM Z cryptographic software components

ICSF provides the application programming interfaces (APIs) by which applications request cryptographic services, such as the following examples:

- Encryption and decryption
- Digital signature generation and verification
- Hash-based Message Authentication Code (HMAC) generation and verification
- Key and key pair generation
- Data hashing

ICSF callable services and programs can be used to generate, maintain, and manage operational keys (also known as *data-encrypting keys*), which are used in cryptographic operations.

Figure 4-3 shows the ICSF architecture and its relationship with the IBM Z hardware and software components.

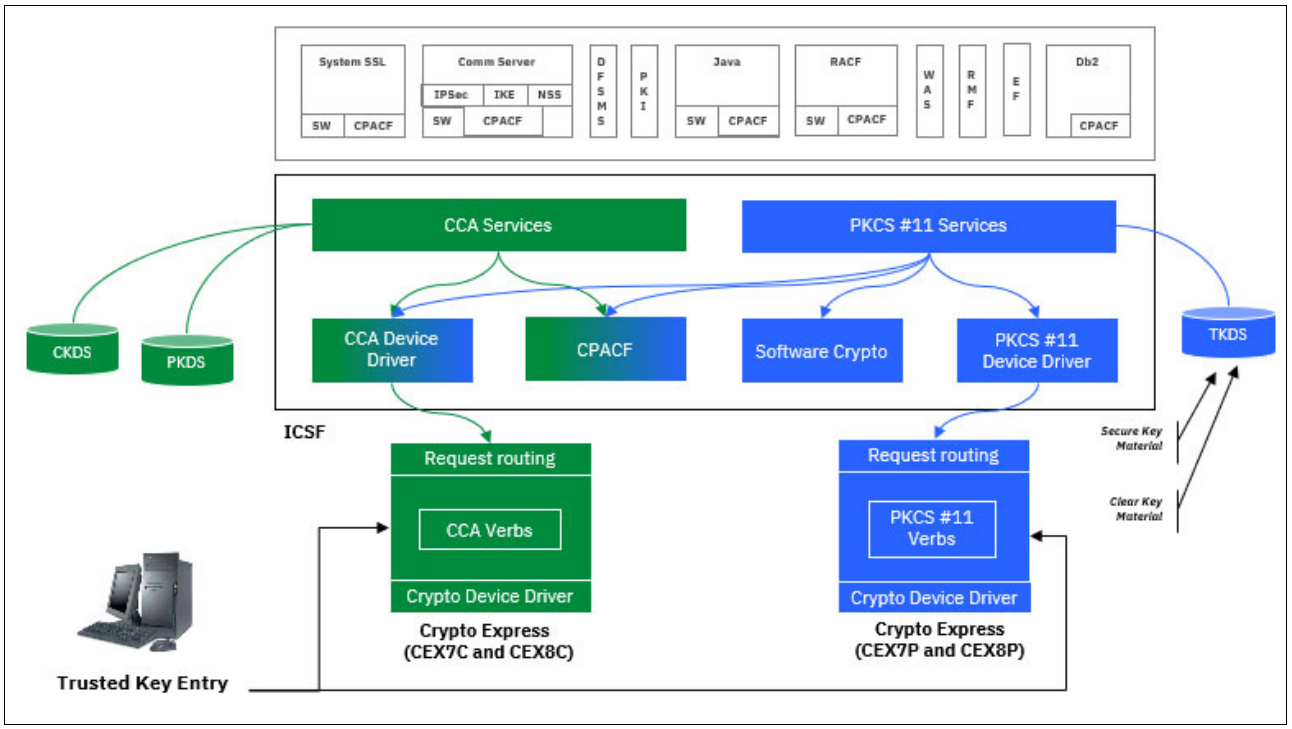



*Figure 4-3 IBM Z software cryptographic components*

ICSF supports two cryptographic architectures: CCA and Enterprise PKCS #11 (EP11). ICSF provides quantum-safe algorithms in both architectures. Software support for the algorithms is available by way of ICSF.

Hardware support for the algorithms is available through the Crypto Express7S or later. For more information about supported algorithms for each coprocessor type, see “Minimum hardware and software for quantum-safe cryptography support” on page 59.

ICSF provides callable services and utilities to generate and store cryptographic keys into ICSF Key Data Sets (KDS). Each KDS is a VSAM data set for persistent objects (such as keys and certificates) with programming interfaces for object management. Each record in the KDS contains the object and other information about the object.

The following types of ICSF Key data sets are available:

- CKDS: Cryptographic Key Data Set: Stores CCA Symmetric Keys such as AES, DES, and HMAC.
- PKDS: PKA Key Data Set: Stores CCA Asymmetric keys such RSA, ECC, and QSA.
- TKDS: Token Data set: Stores PKCS #11 Keys and Certificates.

If a PKDS is allocated and you want to store ML-DSA, ML-KEM, CRYSTALS-Dilithium, or CRYSTALS-Kyber CCA key tokens, you must convert your PKDS over to KDSRL format. For more information, see “Converting your PKDS to KDSRL format” on page 113.

Master keys are used to protect sensitive operational keys that are used in your system. The number and type of master keys active in your system depend on your hardware configuration and application requirements:

- DES master key protects DES keys
- AES master key protects AES and HMAC keys
- RSA master key protects RSA keys
- ECC master key protects ECC, RSA, ML-DSA, ML-KEM, CRYSTALS-Dilithium, and CRYSTALS-Kyber keys

Master keys are stored within the secure hardware boundary of the Crypto Express HSM. The values of the master keys never appear in the clear outside the Crypto Express HSM.

For a master key to become active, the current active master key verification pattern (MKVP) in the Crypto Express HSM and the MKVP in the KDS header must match. When a new Crypto Express HSM is added to your environment, it must be loaded with the same master keys that were used to initialize the KDS.

You can verify the MKVPs in the Crypto Express HSM match the MKVP in the KDS by using the **D ICSF, MKVPS** command (see Example 4-1).

*Example 4-1 Output from D ICSF, MKVPS command*

```
D ICSF, MKVPS
CSFM668I 18.08.59 ICSF MKVPS 129
  CKDS  ICSF.CKDS.NEW
    AES MKVP Date=2022-03-25 00:25:40
    DES MKVP Date=2022-03-23 13:24:53
                ID     AES      DES
    KDSMKVPS  ....   265995   0C3BE0
    SYSZ      7C00   265995   0C3BE0
    SYSZ      7C01   265995   0C3BE0
  PKDS  ICSF.PKDS.NEW
    ECC MKVP Date=2022-03-23 13:25:49
    RSA MKVP Date=2022-03-23 13:25:49
                ID     ECC      RSA
    KDSMKVPS  ....   2ADB6C   4727DB
    SYSZ      7C00   2ADB6C   4727DB
    SYSZ      7C01   2ADB6C   4727DB
  No TKDS defined or no EP11 adapters online
```

For more information about the hardware ICSF supports, and master key types and how they are entered when ICSF first starts, see *z/OS Cryptographic Services ICSF Administrator’s Guide*, SC14-7506.

### 4.1.3 Minimum hardware and software for quantum-safe cryptography support

Support for quantum-safe algorithms for CCA and PKCS #11 are provided with the IBM z17, IBM z16, and IBM z15 platforms. To ensure you can transition your applications to quantum-safe algorithms, verify that your environment meets the minimum hardware and software requirements for each algorithm.

Table 4-2 lists the minimum hardware and software prerequisites to support quantum-safe cryptography on IBM z17, IBM z16, and IBM z15, based on algorithm strength and version.

*Table 4-2 Minimum hardware and software prerequisites*

| Algorithm | Algorithm strength | Cryptographic hardware | ICSF required APARs | IBM Z platform[a] |
|---|---|---|---|---|
| ML-DSA | ML-DSA 4,4<br>ML-DSA 6,5<br>ML-DSA 8,7 | Crypto Express8S CCA coprocessor | HCR77D2<br>OA66395 | IBM z17 |
| ML-KEM | ML-KEM 768<br>ML-KEM 1024 | Crypto Express8S CCA coprocessor | HCR77D2<br>OA66395 | IBM z17 |
| CRYSTALS-Dilithium | CRYSTALS-Dilithium 6,5 Round 3<br>CRYSTALS-Dilithium 8,7 Round 3<br>CRYSTALS-Dilithium 8,7 Round 2 | Crypto Express8S CCA coprocessor | HCR77D1<br>OA61609 | IBM z16 |
| | | Crypto Express8S EP11 coprocessor[b] | HCR77D2<br>OA61609 | |
| | CRYSTALS-Dilithium 6,5 Round 2 | Crypto Express7S CCA coprocessor | HCR77D1<br>OA58880 | IBM z15 |
| | | Crypto Express7S EP11 coprocessor | HCR77D1<br>OA58358 | |
| CRYSTALS-Kyber | CRYSTALS-Kyber 1024 Round 2 | Crypto Express8S CCA coprocessor | HCR77D1<br>OA61609 | IBM z16 |
| | | Crypto Express8S EP11 coprocessor[b] | HCR77D2<br>OA61609 | |

a. Check with your IBM representative for IBM Z hardware driver levels that support quantum-safe cryptography on IBM z17, IBM z16, and IBM z15.
b. At the time of writing, EP11 coprocessor support is only provided for the IBM z17 with this feature and function.

### 4.1.4 Migrating from CRYSTALS-Dilithium and CRYSTALS-Kyber to ML-DSA and ML-KEM

This publication provides guidance for many useful tools for the wider enterprise migration from conventional asymmetric algorithms like RSA and ECC to post-quantum algorithms like ML-DSA and ML-KEM. Some organizations might have a smaller and more immediate concern for migration of their existing exploratory projects using CRYSTALS-Dilithium and CRYSTALS-Kyber to use the standardized algorithms ML-DSA and ML-KEM.

If there has been work toward cryptographic agility as the exploratory projects were planned and implemented, then this is a perfect test case for that policy-driven approach to algorithm

management. The steps involved should be a careful consideration and planning stage for the impact of changing to the new standard approaches, then an execution phase and transition period, and finally work to tie up loose ends and distill lessons learned for the next migration.

There are some considerations to be aware of as part of the planning stage:

- ML-DSA migration from CRYSTALS-Dilithium
  - ML-DSA is not binary compatible to CRYSTALS-Dilithium: you cannot verify a CRYSTALS-Dilithium signature with an ML-DSA public key
  - Pre-hash ML-DSA offers the option of hashing large objects that need to be signed before passing them to the ML-DSA signature service. However, these signatures are not binary compatible with pure ML-DSA signatures.
- ML-KEM migration from CRYSTALS-Kyber
  - ML-KEM is not binary compatible with CRYSTALS-Kyber: you cannot unwrap a CRYSTALS-Kyber encrypted secret with an ML-KEM private key
  - ML-KEM works slightly different – you will only be able to achieve a new shared secret with ML-KEM, you will not be able to encrypt a known value with an ML-KEM public key

**Note:** Maintaining this is very important for the security of the algorithm.

For both ML-KEM and ML-DSA keys, the sizes of similar strength to CRYSTALS-Kyber and CRYSTALS-Dilithium keys are not the same, but are very close. This means that there is not a drastic change in buffer sizes or key storage size, but a review should be planned.

With this in mind, the migration from CRYSTALS-Kyber and CRYSTALS-Dilithium keys to ML-DSA and ML-KEM keys is largely in the hands of the application that uses the cryptographic services. ICSF key support and tooling has been updated for ML-KEM and ML-DSA to assist as much as possible.

## 4.2 Steps towards quantum protection

As you begin to plan your transition to quantum-safe technology, several stages must be considered. These stages include discovering and classifying the data, creating a cryptographic inventory, considering cryptographic agility, and adopting quantum-safe cryptography. These topics are described in this section.

### 4.2.1 Discovering and classifying the data

It is important to classify and identify your most sensitive and valuable data. Classification of data is vital to help distinguish high-value information from information that does not have the same protection requirements. Protection requirement applies to data that belongs to your clients and the data you own.

This process also involves identifying the location of the data and understanding whether any compliance requirements or regulations are associated with the retention of that data.

Another essential consideration is identifying the internal data that your organization considers most valuable. It is important to create and manage a data inventory and define ownership of the data. Classifying data helps prioritize where to apply quantum-safe methods to protect the data. Some data must be protected because of regulations or standards that require compliance and those cases and the level of protection that is must be identified.

Some common types of data that must be protected include confidential business data, intellectual property data, and personally identifiable information (PII), such as social security numbers or drivers license numbers.

Confidential business data can include items, such as product release information, marketing strategies, financial reports, and communications with business partners.

Intellectual property can include design documents, research findings, trade secrets, and formulas. Each organization must identify the information that is most critical to protect.

Although not all data needs the same level of attention, prioritization is required in most cases during the quantum-safe cryptographic journey. It is important to know where your organization's most valuable or sensitive data resides.

Today, you might use encryption at the infrastructure level for data-in-flight or data-at-rest, and in your applications through middleware or custom application code.

Figure 4-4 shows where data encryption might be occurring today in your environment.

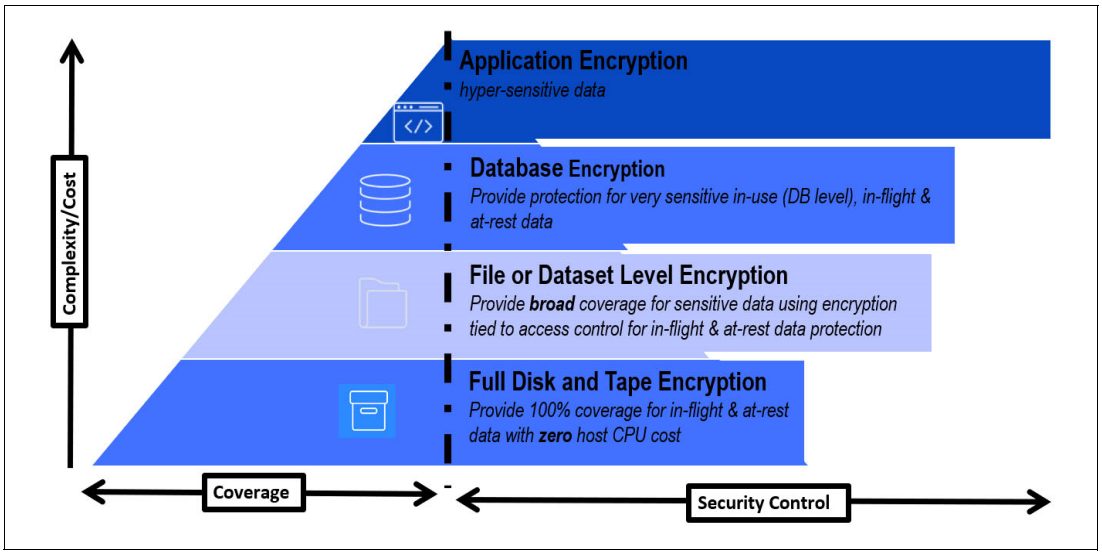



*Figure 4-4 The encryption pyramid*

IBM Z pervasive encryption[7] can be used to provide quantum-safe protection for data at-rest with AES.

During the discovery step, you need help from the applications owners to identify the data they are processing in their applications. You also need the help of your risk and compliance team because they might already have a mapping of sensitive information and its location. If not, now is the time to collect such information and define ownership and location of all sensitive data.

After the ownership of the data and its sensitivity is established, it is key to prioritize the analysis of the applications and assets if encryption is used and if it is used, which cryptographic algorithm is being used. Based on this priority list, the application can enter the cryptographic inventory phase.

[7] For more information, see Getting Started with z/OS Data Set Encryption and Getting Started with Linux on Z Encryption for Data At-Rest.

A sample application data asset inventory is shown in Figure 4-5.

| Data Type/Assset | Application | Owner | Sensitiviy | Priority |
|---|---|---|---|---|
| Data_1 | Application_1, Application_3 | Owner_1 | High | 1 |
| Data_2 | Application_2 | Owner_2 | Low | 3 |
| Data_3 | Application_4, Application_5 | Owner_1 | High | 2 |

*Figure 4-5 Sample data inventory sheet*

## 4.2.2 Establishing a cryptographic inventory

Another vital stage is the creation of a cryptographic inventory, which is an important security artifact. A cryptographic inventory helps document the cryptographic algorithms that are used and why they are used.

For example, if the cryptography is used to digitally sign documents, the cryptographic inventory indicates specific information, such as the name of the signing application, which cryptographic algorithms are used, what are the lengths of the cryptographic keys that are used, and the details of the crypto algorithm provider.

It is important to identify how the documents are signed to determine which mitigation steps are required. Therefore, creating a repository that shows the cryptographic algorithms that are in use within the organization is important.

Also, consider areas where cryptography can be hidden from obvious view. Examples include configuration options that determine the cryptography in use (such as TLS configurations) and key containers (such as certificates).

If a vendor developed the component that uses cryptography, it is necessary to discuss this topic with that vendor to understand the cryptography that is used and the vendor's plans for adopting quantum-safe technology. After you have this type of information, you must determine whether the current cryptographic protection is sufficient, or if a mitigation action is required.

It also is important that you understand which algorithms are not considered quantum-safe (see 1.4, "Cryptographic vulnerabilities possible with quantum computers" on page 8).

A cryptographic inventory includes many items, such as the following examples:

- Component or application under evaluation
- Function or feature that uses crypto
- Person responsible (Who owns or uses the component?)
- Symmetric algorithms, function, and key size
- Asymmetric algorithms, function, and key size
- Hash algorithms and digest size
- Crypto algorithm implementation (hardware and software)
- Crypto provider (HSM and library)
- Crypto vendor (IBM or open-source)
- Interoperability with business or crypto partners
- Key provisioning and storage

The analysis is performed based on the priority list that is established when the discovering and classifying the data step is done (see Figure 4-5 on page 62). Each type of data and

application must be assessed to identify the cryptographic algorithm in use. Encryption can be carried out at the infrastructure or at the application level.

Application owners should know whether and what encryption algorithm is used. If not, several tools can be used to assist during the cryptographic inventory step, including the following examples:

- IBM z/OS Integrated Cryptographic Service Facility (ICSF)
- IBM Application Discovery and Delivery Intelligence (ADDI)
- IBM Crypto Analytics Tool (IBM CAT)
- IBM z/OS Encryption Readiness Technology (zERT)

Also see IBM Z Crypto Discovery and Inventory.

All information that is captured must be recorded and stored in a safe place. A sample cryptographic inventory sheet (see Figure 4-8 on page 68).

## IBM ICSF cryptographic usage tracking

ICSF cryptographic usage tracking with ICSF HCR77C1 (or later) supports data collection (in SMF records) for Crypto Express coprocessors, ICSF callable services, and cryptographic algorithm. The use of the SMF records that are created by the usage tracking provides the following identification information:

- All the jobs (or tasks) that use ICSF cryptographic services
- Cryptographic algorithm that is used (along with their strength)

The use of ICSF cryptographic usage tracking is an efficient way to build, over time, an inventory of the cryptographic algorithm usage and identify the candidate to a migration to quantum-safe algorithm.

Because ICSF cryptographic usage tracking data is stored in SMF records, you can use the ICSF samples to format the records. You can also use other tools, such as IBM Security zSecure Audit, for reporting purposes.

You also can send your cryptographic usage to security information and event management (SIEM) software, such as IBM Security zSecure Adapter for SIEM, which can act as a central repository for tracking cryptographic algorithm usage real time.

A use case for the use of ICSF cryptographic usage tracking is a first view of your production application environments to identify where ICSF cryptographic services (and what algorithm) are being used. From the SMF records, you can identify all the cryptographic services that are used by applications, middleware, or infrastructure components.

After the classification of the results (which are identified as non-quantum-safe algorithm and which are quantum-safe), you can prioritize your approach and use other tools to further complete the inventory.

Later, the results of this analysis, which are classified as quantum-safe, can be analyzed with other tools to complete your cryptographic inventory.

## IBM Application Discovery and Delivery Intelligence

IBM Application Discovery and Delivery Intelligence (ADDI) is an analytical platform for application modernization. It uses cognitive technologies to analyze mainframe applications and quickly discover and understand interdependencies of changes.

By using ADDI and the Crypto Analysis function, you can:

- Discover where and what cryptography is used in applications

- Support migration and modernization planning:
  - Quickly reacting to security issues
  - Store results in a repository
- Capture valuable metadata and dependencies:
  - Identifies ICSF Crypto APIs
  - Identifies Rule Array and other important parameters

For example, you know that one of your critical applications (written in COBOL) is using cryptography, but you do not know exactly how this encryption capability was implemented and what encryption algorithm (and “options”) were used.

By using ADDI and its Crypto Analysis, you can quickly import the COBOL source and discover what ICSF cryptographic APIs and parameters were used.

For more information, see “Using IBM Application Discovery and Delivery Intelligence” on page 93.

## IBM Crypto Analytics Tool

The IBM Crypto Analytics Tool (IBM CAT) was developed to help provide up-to-date monitoring of cryptography-related information about z/OS in the enterprise. IBM CAT is designed to combine and present cryptographic information in a way that helps ensure compliance and policy enforcement.

The IBM CAT Agent, running on z/OS, collects cryptographic information across the enterprise that is then made available through an IBM Db2® for z/OS database to the IBM CAT Monitor that is running on your desktop.

The IBM CAT Monitor provides overviews, queries, and reports to better manage the cryptographic setup.

With IBM CAT, you can generate reports on your cryptographic configuration, compare the reports, and define and apply policy to your cryptographic elements (cryptographic cards, keys, and other cryptographic entities).

IBM CAT also helps you identify any cryptographic object that is not quantum-safe in your current keystores. IBM CAT also provides monitoring over time to ensure that no non-quantum-safe cryptographic object are created.

For example, if your applications use encryption through middleware, you can identify the keys in the keystore that belong to this application with the IBM CAT Monitor. You also can verify how cryptographic security was implemented to protect cryptographic objects, and who can access them[8].

For more information, see “Using IBM Crypto Analytics Tool” on page 99.

## IBM z/OS Encryption Readiness Technology

z/OS Encryption Readiness Technology (zERT) is a z/OS Communications Server feature that provides intelligent network security discovery by monitoring and reporting TCP and Enterprise Extender connections on your z/OS system for TLS/SSL, IPsec and SSH protection.

zERT writes its data collection in SMF records (type 119 subtype 11 and subtype 12).

[8] Only when using IBM RACF

zERT helps you answer the following questions:

- What TCP/IP and Enterprise Extender traffic is being protected (and which is not)?
- How is that traffic protected? For example, what protocols are being used, which cryptographic algorithms are being used, and what key lengths?
- Who on my z/OS system uses or produces the network traffic, whether it is protected or not?
- Where is the remote endpoint for that traffic?
- With zERT policy-based enforcement, you can write rules to enforce real-time notifications and even real-time defensive actions based on the cryptographic protection attributes collected by zERT for each TCP connection.

IBM zERT Network Analyzer is a web-based graphical user interface10 that z/OS network security administrators can use to analyze and report on data that is reported in zERT Summary records (SMF type 119 subtype 12). By using IBM zERT Network Analyzer, you can build your own cryptographic inventory for data that is in-flight by using its reporting capabilities. The reports can be exported in CSV format to be integrated in enterprise-level repository, if needed.

For more information, see “IBM z/OS Encryption Readiness Technology” on page 64.

### IBM z/OS crypto counters

SMF record type 30 was introduced for cryptographic usage of CPACF in a cryptographic counter section.

The z/OS SMF record type 30 provides accounting information and consolidate data from other record types. Records are generated when certain events occur during an address space's lifecycle, such as job submission, job initiation, step termination, job termination, and interval last.

In addition to the ICSF SMF record type 82, SMF record type 30 can be very useful in detecting application and vendor software that directly call CPACF instructions, as such application that may not use the ICSF application programming interface.

The crypto counters sections in the SMF record type 30 will report the number of times each cryptographic instruction has been used.

Each crypto counter can be easily mapped to different algorithms and key length helping with identifying non-quantum safe usage (see – A.2, “z/OS SMF record type 30 sample output” on page 146).

## 4.2.3 Considering cryptographic agility

Cryptographic agility is about the ability to quickly adopt new cryptography in an application, component, or system with minimal impact to the underlying infrastructure. Over time, we saw and understand that cryptographic algorithms change.

However, many applications used hardcoded cryptographic primitives that might not be easy to change. Going forward, we must consider ways to make it as easy as possible to manage the cryptography that is in use in our enterprises.

Several dimensions of cryptographic agility must be considered (see Table 2-4 on page 27):

- Update cryptographic algorithms when broken
- Change cryptographic algorithms when new regulatory requirements exist

- Monitor cryptographic algorithms to ensure that those algorithms are used correctly
- Retire cryptographic algorithms when obsolete

Cryptographic agility is a core component of cyber resiliency. If a cryptographic algorithm is found to no longer be secure, the ability to switch to a secure algorithm quickly is essential. We must consider effective ways to manage and use cryptography and automation to simplify the transition to new cryptography.

One way to achieve cryptographic agility is to decouple the cryptographic algorithm from the application code, which makes the change to another encryption algorithm faster, without changing the application code again.

For example, Advanced Crypto Service Provider (ACSP) can be used to provide an abstraction layer between the application code and the encryption of data by using the IBM Z capabilities.

## Advanced Crypto Service Provider

The enterprise cryptographic environment can be spread over several different systems with individual HSMs and application landscape. Therefore, ensuring that all the platforms and environments are remaining quantum-safe might be a challenge.

To help achieve cryptographic agility in a client/server environment, you can centralize the execution of cryptographic operations and establish Crypto-as-a-Service.

One of the IBM Z solutions for establishing Crypto-as-a-Service is to use the cryptographic provider Advanced Crypto Service Provider (ACSP).

With ACSP, you can use your low-used IBM Z cryptography and make it available for all your platforms where applications must use quantum-safe cryptography.

By using the User Defined Functions (UDF) with ACSP, you can implement business-specific functions that are made available through ACSP where the application requests cryptography, without needing to handle the encryption by alone. This ability makes the application crypto-agile because the cryptographic policy is applied at the ACSP server level, which ensures that the correct algorithm is used to encrypt and decrypt the data (see Figure 4-6).

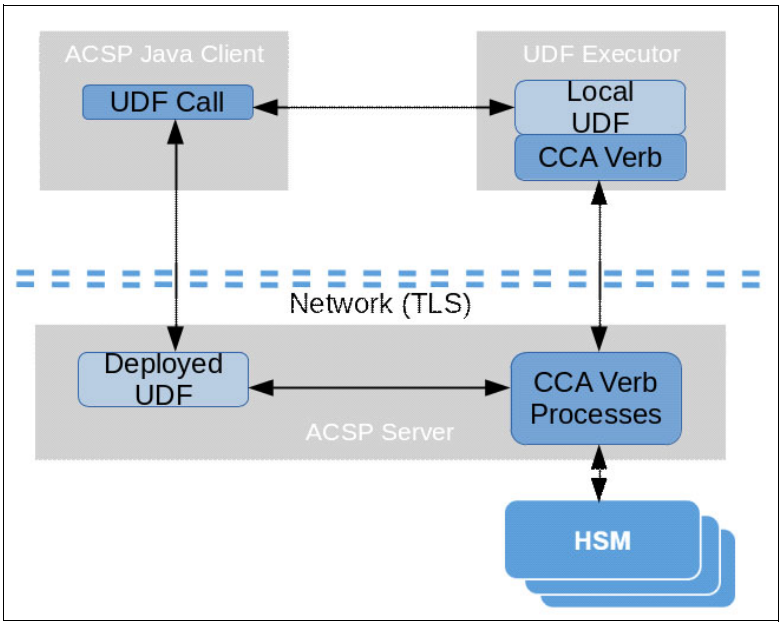



*Figure 4-6 UDF calling path for a local and deployed set-up*

A possible architecture that implements cryptographic agility by using ACSP is shown in Figure 4-7.

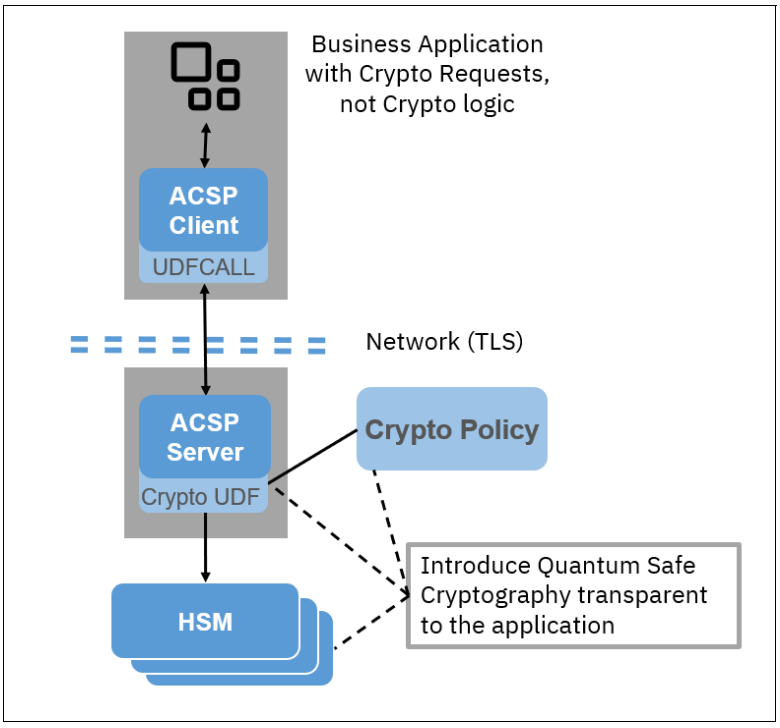



*Figure 4-7 ACSP architecture used for cryptographic agility*

ACSP includes a sample Agile UDF that supports runtime change in Digital Signature algorithms.

The test application and the java sample code delivered as part of the ACSP Solution supports switching between generating and verifying digital signatures with these three key types:

- ECC – Brainpool
- RSA – CRAESOPK
- CRYSTALS-Dilithium

For an example usage of the Agile Digital Signing UDF see 6.6, “Quantum-safe digital signatures” on page 122.

### 4.2.4 Adopting quantum-safe cryptography

After you know your current cryptography state, examine the technical mitigations that are available to you. Understand who can provide the necessary technology. Determine which options must be applied to each of your use cases.

For more information about common use case examples, see Chapter 3, “Using quantum-safe cryptography” on page 31.

A risk assessment is performed to determine the priority of implementation and testing. Some implementation options include strengthening the symmetric algorithms that are used for data protection.

Use hybrid key exchange methods or dual signing schemes that use classical and quantum-safe algorithms see 4.1.1, “IBM Z cryptographic hardware components” on page 52 and 3.3.2, “Solving the integrity challenge with IBM Z capabilities” on page 42. When necessary, the use of physical isolation techniques can be put in place to keep critical information off the network and data on systems that feature restricted access and controls.

Educating your teams about the options and early planning and testing is key to having a successful quantum-safe transition experience.

When all of the data, application, and cryptographic key and algorithm information is collected and recorded and a priority list established, the implementation phase can start. A sample cryptographic inventory sheet is shown in Figure 4-8.

| Data Type/Assset | Owner | Crypto usage/algorithm | Other platforms/applications | Mitigation/ Quantum-safe required | Priority |
|---|---|---|---|---|---|
| Data_1 | Owner_1 | Data encryption, AES 256 | n/a | n/a | n/a |
| Data_2 | Owner_2 | Data encryption, DES | x86 | AES 256 | 2 |
| Data_3 | Owner_1 | Digital Signature SHA256, RSA | n/a | Replace by ML-DSA | 1 |

*Figure 4-8 Sample cryptographic inventory sheet*

Based on the discovered items, the applications (or infrastructure components) must move to quantum-safe encryption if they are not quantum-safe (AES is quantum-safe).

An example of a non-quantum-safe condition is an application that signs data with RSA. To make it quantum-safe, a dual signature scheme is used following NIST recommendations.

A dual signature consists of two (or more) signatures on a common message. The verification of the dual signature requires all of the component signatures to be successfully verified.

In a dual signature, one signature is generated with a NIST-approved signature scheme as specified in FIPS 186, while another signatures can be generated by using a different algorithm.

Dual signatures can be accommodated by current standards in “FIPS mode,” as defined in FIPS 140, if at least one of the component methods is a correctly implemented, NIST-approved signature algorithm. It is up to the application to specify how to parse signatures and verify them separately.

For more information, see the following resources:

- This NIST post-quantum cryptography (PQC) web page
- This French National Cybersecurity Agency (ANSSI) web page

Some technical environment upgrades or modifications might be required to support quantum-safe cryptography. For more information, see 4.1, “IBM Z cryptographic components overview” on page 52, and “Ensuring the environment is ready” on page 116.

### 4.2.5 Where to find help at IBM

IBM Technology Expert Labs offers a quantum-safe assessment, which is a useful way to quickly understand how your company uses IBM Z cryptographic features and to inventory the cryptography in use today to maximize quantum-safe capabilities of your IBM Z environment.

The assessment analyzes current best practices and provides a practical roadmap of actions to strengthen your quantum-safe posture.

For more information, email us at mailto:systems-expert-labs@ibm.com

## 4.3 Best practices, mitigation options, and tools

This section provides information about how to use security best practices to build a secure z/OS cryptographic environment. It also includes some mitigation options that you can use along with an introduction to the key management tools, which must be considered during your quantum-safe journey.

### 4.3.1 ICSF best practices

Implementing a secured cryptographic environment on z/OS is the foundation for building a strong quantum-safe cryptographic environment. The following sections describe best practices for ICSF.

#### ICSF configuration

ICSF is likely the repository for all of your keys. Therefore, you must ensure it is configured for the maximum level of security for your encrypted data. We suggest configuring ICSF by using the following settings:

- No compatibility/coexistence mode

  Compatibility mode was introduced to run applications written with Programmed Cryptographic Facility (PCF), ICSF ancestor, without reassembling the application. Because these applications likely use weak non-quantum-safe algorithm, they must be converted to ICSF and use quantum-safe algorithm.

If you still use AMS REPRO encryption (which requires compatibility mode), you must use other means to encrypt your data. IDCAMS ENCIPHER/DECIPHER works with weak 56-bit DES key (no Triple DES support).

Also, changing the master keys in compatibility mode requires an IPL of the system.

Verify that the parameter COMPAT(YES) or COMPAT(COEXIST) is not specified in CSFPRMxx. The default is COMPAT(NO).

- No special secure mode

  When special secure mode (SSM) is enabled, ICSF enables the generation or entry of clear keys, which lowers the security of the system. Clear keys usage should not be allowed.

  Verify that the parameter SSM(YES) is not specified in CSFPRMxx. The default is SSM(NO). SSM can also be enable by a CSF.SSM.ENABLE.SAF profile in the XFACILIT resource class. Ensure it is not defined.

- Allocation of the xKDS

  The xKDS data sets that are managed by ICSF more likely contain all of your keys. You must ensure that the initial allocation of these data sets allows growth.

  To support quantum-safe keys, the xKDS must be in KDSRL format (see SYS1.SAMPLIB(CSFCDKS) and SYS1.SAMPLIB(CSFPKDS) as provided with ICSF HCR77D2 (z/OS 2.5). See 6.2, “Converting your PKDS to KDSRL format” on page 113 for more information.

  The CKDS, PKDS contains one initialization record with the values of the current master keys verification patterns (MKVPs) for the type of keys that are stored in the keystore (AES and DES MKVPs for the CKDS, ECC, and RSA MKVPs for the PKDS).

  Then, each VSAM record contains one key.

  The default found in the SYS1.SAMPLIB allocates a CKDS with 200 keys in the primary extent and 100 in each secondary extent. It also allocates a PKDS with 100 keys in the primary extent and 50 in each secondary extent.

  When allocating your CKDS, consider your number of keys and the growth that is expected from a data encryption perspective. This growth must include key rotation in the coming years and factor how long you must keep old keys “alive” in your CKDS to access old data from backups you must keep for regulatory purposes.

  An installation that includes 100 keys in their CKDS, an application encryption requirement growth of 10% per year, rotates their operational keys every year, and a 10-year data retention period must be prepared to store more than 500 keys.

  Taking a large margin and over-allocating the CKDS to 1000 records in the primary extend uses only 10 tracks total (VSAM DATA + INDEX).

## ICSF and z/OS security best practices

These z/OS security best practices aid in enhancing the protections across the z/OS cryptographic stack. Locking down the z/OS cryptographic stack enables you to realize the following benefits:

- Reduce the attack surface of your z/OS environment
- Enable routine security hygiene practices
- Use new capabilities and features
- Meet regulatory compliance requirement

### *Key label naming conventions*

Cryptographic keys that are stored in the ICSF key data sets can be referenced by their key label. A key label can be up to 64 characters and consist of alphanumeric characters, national characters (#, $,@), or a period. When determining a key label name, consider the following factors:

- LPAR that is associated with the key
- Type of data that is encrypted
- Owner that is associated with the key
- Date that the key was created
- Application that uses the key
- A sequence number for the key

Consider the following key label example:

```
SYS1.DB2.ENCKEY.202204.0001
```

### *Protecting cryptographic keys and ICSF services*

Access to CCA cryptographic keys is controlled through the CSFKEYS general resource class. When a key is used in an application, ICSF checks for a discrete CSFKEYS profile that matches the key's label.

If a covering profile (discrete or generic) exists, access to the key is granted based on whether the user or group has READ access. By default, the CSFKEYS class grants access to the key if no profile is in place.

Ensure that the CSFKEYS class is ACTIVE and RACLISTed and a backstop profile exists; for example, '*' or '**' with UACC(NONE).

Access to PKCS #11 tokens is controlled through the CRYPTOZ general resource class. By default, the CRYPTOZ class does not grant access to the PKCS #11 token if no covering profile exists.

Ensure that the CRYPTOZ class is ACTIVE and RACLISTed. Define USER.* and SO.* backstop profiles with UACC(NONE).

For more information about protecting PKCS #11 tokens with the CRYPTOZ class and the USER and SO (Security Officer) rules, see *z/OS ICSF Writing PKCS #11 Applications*, SC14-7510.

Access to ICSF callable services is controlled through the CSFSERV general resource class. When an ICSF service is called, ICSF checks for a discrete CSFSERV profile that matches the service name. By default, access to most ICSF services is granted if no covering profile exists.

Also, ensure that the CSFSERV class is ACTIVE and RACLISTed. Define a backstop profile; for example, '*' or '**' with UACC(NONE).

### *Key lifecycle and key usage auditing*

ICSF instances can be configured to audit the lifecycle of keys as they transition through the system. Keys can be audited from the time of their initial generation until their eventual deletion. Key lifecycle audit data is written as SMF Type 82 subtype 40, 41, and 42 records.

ICSF key lifecycle auditing includes the following options:

- AUDITKEYLIFECKDS: In the CKDS
- AUDITKEYLIFEPKDS: In the PKDS
- AUDITKEYLIFETKDS: In the TKDS

Key Lifecycle auditing can be enabled in the ICSF installation options data set or dynamically by using the **SETICSF** command.

ICSF instances can be configured to audit key usage. Key usage data can be used to determine which key was used, who used the key, and when the key was used. Key usage audit data is written as SMF Type 82 subtype 44, 45, 46, and 47 records.

ICSF key usage auditing includes the following options:

- AUDITKEYUSGCKDS: CCA symmetric tokens
- AUDITKEYUSGPKDS: CCA asymmetric tokens
- AUDITPKCS11USG: PKCS #11 keys

Key usage auditing can be enabled in the ICSF installation options data set or dynamically by using the **SETICSF** command.

For more information about ICSF key lifecycle and key usage auditing, see *z/OS ICSF System Programmer's Guide*, SC14-7507.

### *SAF protecting the ICSF Key Data Sets*

ICSF Key Data Sets contain the CCA and EP11 cryptographic keys that are used within ICSF callable services. Although use of the keys can be protected with the CSFKEYS and CRYPTOZ general resource classes, the Key Data Sets also must be SAF protected.

Without sufficient SAF protections on all three ICSF Key Data Sets, the ICSF keys are at a greater risk of becoming compromised. Regardless of whether you use clear or secure keys, each of your ICSF Key Data Sets must include a DATASET profile with UACC(NONE).

## Backing up ICSF keys

It is important to adopt a routine schedule of backing up the ICSF keystores. Backing up the keystore is recoverable. By regularly backing up the DASD volumes that contain the key stores, the entire volume can be restored if the volume becomes corrupted.

Also, create backups before and after major key management operations. For example, performing an unfamiliar key management operation, generating many new keys, or after a master key rotation.

When creating your backup, consider whether you should have an online or offline backup. An online backup provides the quickest recovery from a corrupted keystore or deletion, but it is also susceptible to attack by a bad actor.

An offline backup (for example to tape) safeguards a keystore from being compromised by malicious software with access to online devices. Although offline backups might not be as up to date, they are less likely to be compromised.

If you are using or plan to implement z/OS data set encryption, we suggest that you deploy a robust key management solution, especially as the number of keys to manage increases.

### 4.3.2 Mitigation options

z/OS includes many features and functions that directly provide quantum-safe encryption for your data when at-rest or when a quantum-safe digital signature is used. The use of quantum-safe encryption algorithm for data-at-rest can provide the following mitigation options if the data remains encrypted while in transit:

- z/OS data set encryption

  When implemented, z/OS data set encryption provides a high level of protection for your sensitive data when stored in z/OS data sets (extended format sequential data sets, extended format VSAM data sets, and basic and large non-extended format sequential data sets).

  z/OS data set encryption is based on AES-XTS block cipher mode with keys of 256 bits strength; therefore, it is considered quantum-safe.

  Also, z/OS data set encryption does not require application changes when it is implemented; therefore, it ensures a short path to quantum-safe encryption.

- JES2 spool encryption

  Because JES2 spool encryption is based on the same technology as z/OS data set encryption, the same level of protection to your data is provided during and after job execution.

  This option is another short path to quantum-safe encryption.

- Digital signatures and SMF

  It is important to protect your data and the data of your clients or business partners, but it is also important to ensure that audit trails are not tampered. SMF digital signature provides an easy way to digitally sign the SMF records.

  When written to the log stream, the SMF records are hashed and SMF periodically signs the hash by using a private key.

  When reading the records, the utility program IFASMFDP verifies that the records were not corrupted or tampered.

  The alternative signature algorithm introduced the use of CRYSTALS-Dilithium digital signature with RSA or ECDSA, which makes SMF digital signing quantum-safe. This support requires an IBM z15 or later environment for the entire sysplex. See for more information. 6.6.4, “Using digital signatures to protect SMF records” on page 124.

### 4.3.3 Key management tools

The use of a quantum-safe cryptographic algorithm is only a first step. Correctly managing the keys that are used by these cryptographic algorithms also is important to ensure they are safely stored and transported and maintain compliance.

#### Trusted Key Entry

TKE provides the compliant-level HSM management mechanisms for HSMs on IBM Z. TKE is used to securely manage all IBM Z HSM settings, including the HSM master keys. Because the TKE allows you to group HSMs and HSM domains together, TKE also greatly simplifies the management of complex IBM Z HSM environments.

TKE is an optional feature code of the IBM Z platform. If purchased, the TKE can be placed in a secure space separate from the IBM Z, or in the same security space.

The TKE feature contains a combination of hardware, firmware, and software. For more information, see this IBM Z Content Solutions web page.

In a quantum-safe environment, it is paramount that all master keys in the xKDS are safely managed and stored. Master keys should *never* be exposed in the clear or wrapped with non-quantum-safe keys in a virtual safe.

All the compliant-level HSM management mechanisms provided by the TKE require smart card readers and smart cards. When you store key parts on smart cards, all the key parts are generated in a cryptographic adapter coprocessor in the workstation and are never outside an HSM or the smart card in the clear.

The smart card readers and smart card are an additional feature of the TKE. With a first-time purchase, you should view readers and smart cards a required feature of the TKE. If you are upgrading an existing TKE, you will likely be able to use the smart card readers and smart cards you have. However, pay attention to technology changes. As standards change and encryption techniques evolve, you will need to move to newer smart cards over time.

With TKE v10.0, a CRYSTALS-Kyber handshake occurs between the TKE Workstation cryptographic adapter and the target Crypto Express HSM when CCA master key parts are loaded into the target.

**Note:** Crypto Express8S and Crypto Express7S features that include quantum-safe capabilities that are enabled cannot be in the same TKE domain group as those features without quantum-safe capabilities enabled.

## IBM UKO and IBM EKMF Foundation

You can transition to Unified Key Orchestrator (UKO) or setup UKO with Enterprise Key Management Foundation (EKMF) for secure room operation at any time during your quantum-safe encryption journey.

UKO is a key management software solution that centrally orchestrates and secures the lifecycle of encryption keys across your enterprise for both on-premises and multiple cloud environments, including IBM Cloud®, AWS KMS, Azure Key Vault and Google Cloud.

UKO with EKMF is a flexible and highly secure key management system for the enterprise. It provides centralized key management on IBM Z and distributed platforms for streamlined, efficient, and secure key and certificate management operations.

For more information, see the IBM UKO web page. UKO with EKMF workstation is available for secure room operation.

All keys and certificates are stored in a central repository with metadata, such as activation dates and usage. By storing all key material in a central repository, backup is easily achieved by including the database in database backup procedures. This feature facilitates easy recovery if keys or certificates are lost.

The UKO V3.1.0.7 solution and UKO with EKMF support ML-DSA and ML-KEM keys.

See Figure 4-9 on page 75 for the UKO template to create an ML-DSA key.

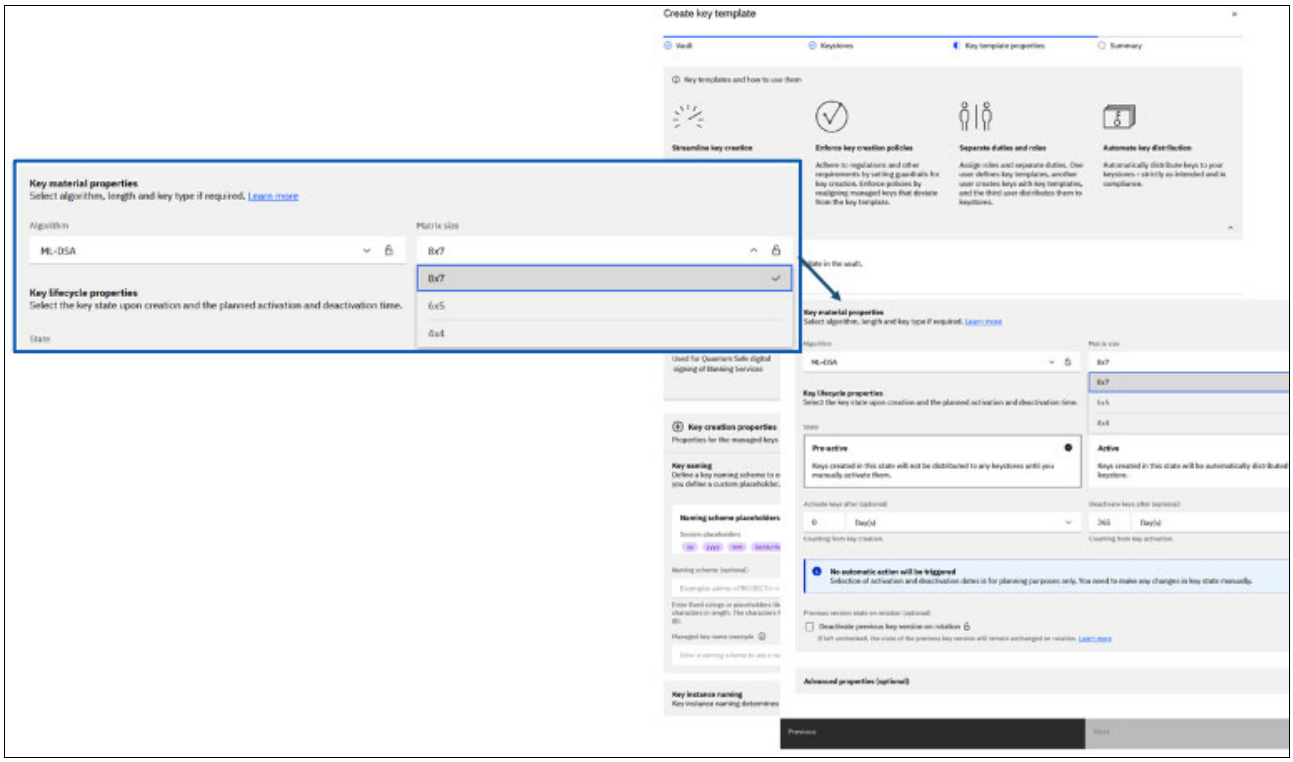



*Figure 4-9 UKO key template for ML-DSA key pair*

See Figure 4-10 for the UKO template to create an ML-KEM key.

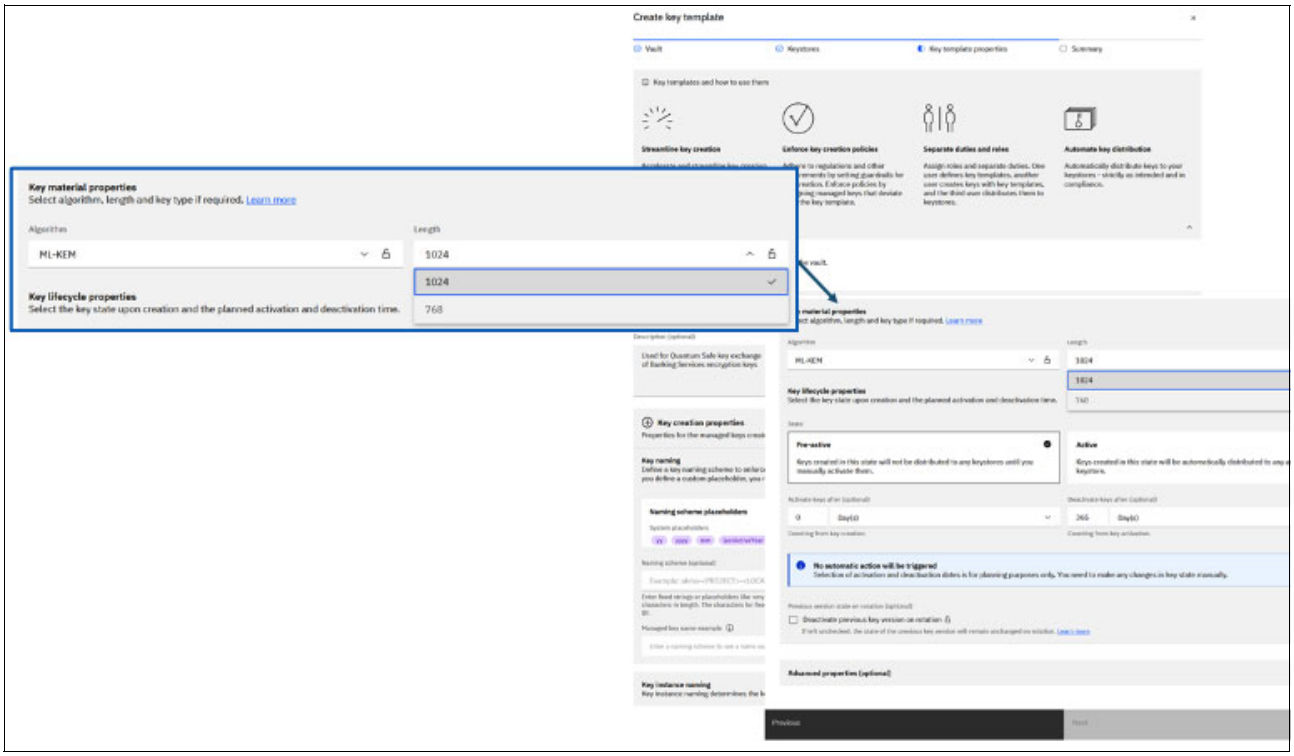



*Figure 4-10 UKO key template for ML-KEM key pair*

For key management support on z/OS, APAR PH66966 is required to update the KMG agent.

5

# Creating a cryptographic inventory

Before transitioning to quantum-safe algorithms, it is important to create a cryptographic inventory that covers all aspects of cryptography in your enterprise. This process includes classifying the data to help you identify where your sensitive data is stored inside and outside the IBM Z environment.

The cryptographic inventory also lists the certificates, encryption protocols, algorithms, and key lengths that are used and indicates those that are weakened by quantum computers. For more information about for an approach for creating a cryptographic inventory, see “Establishing a cryptographic inventory” on page 62.

This chapter shows you how to configure, run, and interpret the results for each of the cryptographic inventory creation tools and includes the following topics:



## 5.1 Collection tools overview

IBM provides several tools that can aid in the cryptographic discovery process and developing a cryptographic inventory for your IBM Z environment. Each tool provides unique information for creating a cryptographic inventory:

- IBM z/OS Encryption Readiness Technology (zERT) collects and reports the cryptographic security attributes of TCP and EE application traffic that is protected by using the TLS/SSL, SSH, and IPsec cryptographic network security protocols.
- IBM z/OS Integrated Cryptographic Service Facility (ICSF) cryptographic usage tracking records are written as SMF records aggregating usage of cryptographic engines, cryptographic services, and cryptographic algorithms.
- IBM z/OS SMF record type 30 includes crypto counters for CPACF usage outside the ICSF cryptographic services.
- IBM Application Discovery and Delivery Intelligence (ADDI) analyzes COBOL application files that capture valuable metadata and dependencies by identifying important ICSF parameters for crypto algorithms.
- IBM Crypto Analytics Tool (IBM CAT) creates snapshots of the z/OS environment by extracting security and cryptographic information that is based on defined policies.

Also see IBM Z Crypto Discovery and Inventory.

After the information is collected, perform a gap analysis to determine whether business, compliance, and audit requirements are being met. The gap analysis aids you in prioritizing updates for your environment.

A process for making use of the various tools to identify the cryptographic algorithm, key length, and key label information that is related to COBOL programs is described in 5.5, “Using IBM Application Discovery and Delivery Intelligence” on page 93.

## 5.2 Using IBM z/OS Encryption Readiness Technology

IBM z/OS Encryption Readiness Technology, known as zERT, provides intelligent network security discovery by monitoring and reporting TCP and Enterprise Extender (EE) connections on your z/OS Systems for TLS/SSL, IPsec and SSH protection.

There are four functions that make up zERT, these include:

- zERT discovery
- zERT aggregation
- zERT Network Analyzer
- zERT policy-based enforcement

With zERT discovery, attributes are collected and recorded at the connection level. These attributes are provided in SMF 119 subtype 11 “zERT Connection Detail” records. These records describe the cryptographic protection history of each TCP and EE connection. At least one record is written for every such connection, so the number of subtype 11 records could be quite large in some environments.

With zERT aggregation, attributes collected by zERT discovery are aggregated by security session. These attributes are provided in SMF 119 subtype 12 “zERT Summary” records. These records describe the repeated use of security sessions over time. Aggregation can

greatly reduce the volume of SMF records while maintaining the fidelity of the information, which is well suited for reporting applications.

Having all this information collected and reported by the TCP/IP stack makes it possible to build a clear picture of your z/OS network protection status. There are over a dozen z/OS-based products from a variety of independent software vendors (ISVs) as well as IBM that consume zERT SMF data and present it within the context of the specific product.

One such product is the IBM zERT Network Analyzer, that is an optional function provided by z/OS Communications Server. The IBM zERT Network Analyzer (see Figure 5-1 on page 79) is a z/OSMF plugin that provides a web-based graphical user interface that allows z/OS network security administrators to analyze and report on zERT data.

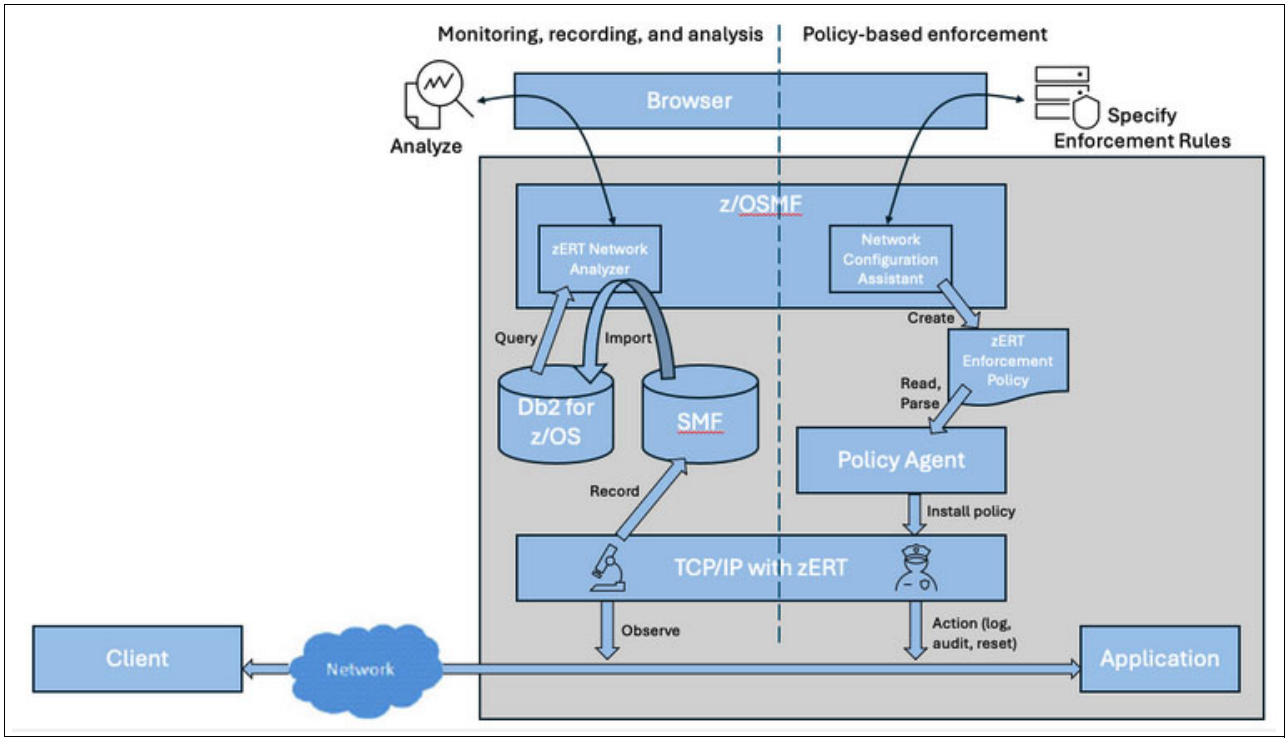



*Figure 5-1 zERT Process*

The zERT Network Analyzer ingests SMF 119 subtype 12 “zERT Summary” records from SMF dump datasets and populates a Db2 for z/OS database with a specialized schema. Once the database is populated, z/OS network security administrators can use the zERT Network Analyzer UI to create and run their own queries to investigate the specific aspects of TLS/SSL, SSH, and IPsec protection on their z/OS systems that interest them. The query results can be viewed in a hierarchical HTML report format or can be exported to a CSV file for later processing.

**Note:** Db2 for z/OS V11 or later is required as the database repository for IBM zERT Network Analyzer.

zERT policy-based enforcement (called zERT enforcement) introduced with z/OS V2R5 Communications Server, allows the TCP/IP stack to act on the data collected by zERT discovery to enforce your local z/OS network crypto standards in real time, as configured in a policy. zERT enforcement rules describe acceptable or unacceptable cryptographic protection attributes for TCP connections and any actions to take when a connection matches the rule.

The action may include:

- Allow the connection (this is the default)
- Write an audit record (zERT connection detail by policy record) to SMF or real-time NMI
- Write a syslog daemon message
- Write a console message
- Reset the connection

Most actions can be specified in combination (for example, we suggest generating a log message any time the reset action is used).

### 5.2.1 Configuring zERT discovery, aggregation and reporting options

zERT discovery and reporting configuration in the TCP/IP profile:

- Enable the zERT in-memory monitoring with the GLOBALCONFIG ZERT parameter (the default value is NOZERT).
- For the zERT connection detail (SMF 119 subtype 11) records to be written to the System Management Facility (SMF) data sets, specify SMFCONFIG ZERTDEtail parameter (the default value is NOZERTDETail)
  - Also, the z/OS SMF configuration parmlib member should enable the collection of the SMF type 119 records.
- The SYSTCPER real-time network monitoring (NMI) service makes zERT 119 SMF zERT Connection Detail (subtype 11) records available to network management applications as they are generated. For more details, see: https://www.ibm.com/docs/en/zos/3.1.0?topic=server-requests-sent-by-client-systcper-service
  - If you have a product that uses the zERT SYSTCPER NMI service to collect zERT connection detail (SMF 119 subtype 11) records, specify NETMONITOR ZERTSERvice parameter (the default value is NOZERTSERvice).

zERT aggregation and reporting configuration in the TCP/IP profile:

- Enable the zERT aggregation function with the GLOBALCONFIG ZERT AGGregation sub-parameter (the default is NOAGGREGATION)
  - Enable the zERT aggregation recording interval using the INTVAL and SYNCVAL sub-parameters on the GLOBALCONFIG ZERT AGGREGATION statement. The default is to use the system's SMF interval which might generate more SMF 119-12s than needed. Using an INTVAL of 6,12 or 24 can cut down the number of records significantly, thus reducing the number of records the zERT Network Analyzer has to import and store.
- For the zERT summary (SMF 119 subtype 12) records to be written to the SMF data sets, specify SMFCONFIG ZERTSUMmary (default is NOZERTSUMMARY)

**Note:** The z/OS SMF configuration parmlib member should enable the collection of the SMF type 119 records.

- The SYSTCPES real-time network monitoring (NMI) service makes zERT 119 SMF zERT Summary (subtype 12) records available to network management applications as they are generated. For more details, see the z/OS Communications Server: IP Programmer's Guide.
  - If you have a product that uses the zERT SYTCPES NMI service to collect zERT Summary records, specify NETMONITOR ZERTSUMmary parameter (the default value is NOZERTSUMmary).

Note that the zERT discovery and aggregation functions are enabled independently of the destinations to which records are written. All parameters can be dynamically enabled or disabled.

## 5.2.2 Using IBM zERT Network Analyzer

There are over a dozen z/OS-based products from a variety of ISVs and IBM that consume zERT SMF data and present it within the context of the specific product. In this section, we will demonstrate how zERT Network Analyzer can be used during the cryptographic inventory step to analyze and report on network encryption. IBM zERT Network Analyzer is a web-based tool that provides z/OS network security administrators with the ability to import, query and analyze data recorded in SMF 119 zERT Summary (subtype 12) records by the z/OS Communications Server zERT aggregation function.

### Configuration tasks overview

Before IBM zERT Network Analyzer is used, the following configuration tasks are required:

1. Enable collection of zERT summary (SMF Type 119 subtype 12) SMF records. Dump the collected zERT Summary records to a sequential data set using the IFASMFDP for SMF data sets or IFASMFDL program for log streams. Separating the SMF 119-12 records into their own dataset improves zNA import performance.
2. Enable the IBM zERT Network Analyzer plugin in z/OSMF by adding ZERT_ANALYZER to the PLUGINS statement of the IZUPRMxx parmlib member. See 'IZUPRMxx reference information' in IBM z/OS Management Facility Configuration Guide for information on how to do this.
3. Authorize the user IDs that are to be allowed to log into the IBM zERT Network Analyzer (see SYS1.SAMPLIB(IZUNASEC).
4. Work with your Db2 for z/OS database administrators to create the Db2 database definitions and connect IBM zERT Network Analyzer to the Db2 database.

For more details, see:
https://www.ibm.com/docs/en/zos/3.1.0?topic=server-requests-sent-by-client-systcpes-service

### Populating IBM zERT Network Analyzer database

The IBM zERT Network Analyzer import function executes under the z/OSMF task's user ID. As such, the z/OSMF task's user ID (usually IZUSVR) must be authorized to read the SMF dump data sets that contain the zERT Summary records.

Complete the following steps:

1. In the main IBM zERT Network Analyzer page, select **Data Management** → **Import SMF Data** → **Add data set**. Add all the required SMF dumps data sets.
2. Select all of the data sets that you want to import in the database and then, click **Import Selected**. Then, confirm that you want to import the selected data sets (see Figure 5-2).

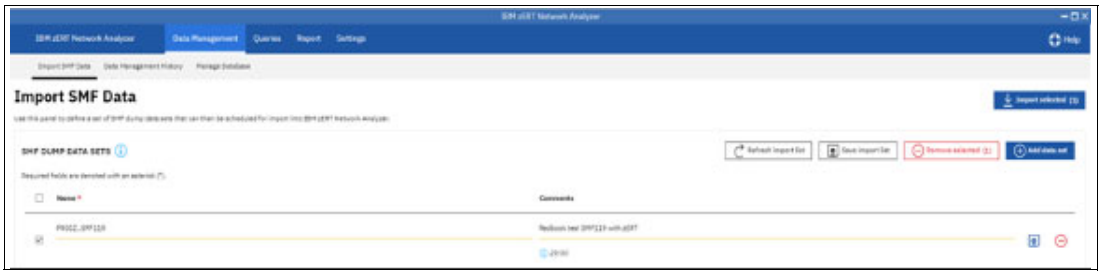



*Figure 5-2 IBM zERT Network Analyzer Import SMD Data*

The import operation is asynchronous, and the completion of the import can be checked in the Data Management History window by reviewing the Status column (see Figure 5-3). The task can be selected to expand to a detailed view of the import operation (number of records added, duplicate, and ignored).

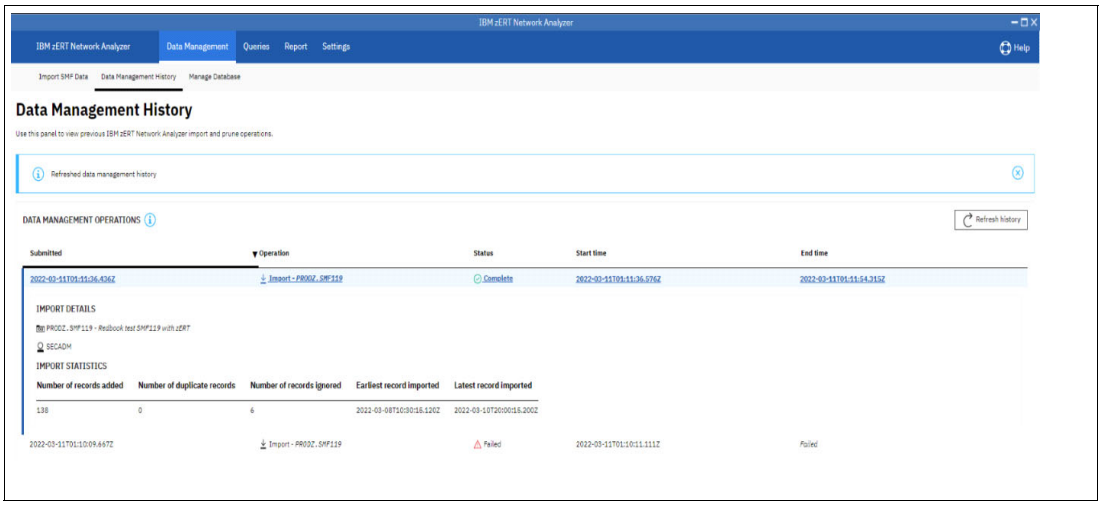



*Figure 5-3 IBM zERT Network Analyzer SMF import details*

## Building your first query

Now that the SMF data is imported into the Db2 database, you can create and run a query against the data in the IBM zERT Network Analyzer database. Complete the following steps:

1. In the main IBM zERT Network Analyzer page (see Figure 5-4), select **Queries** → **New query**. Here, you must provide a query name and (optionally) a description for the query.

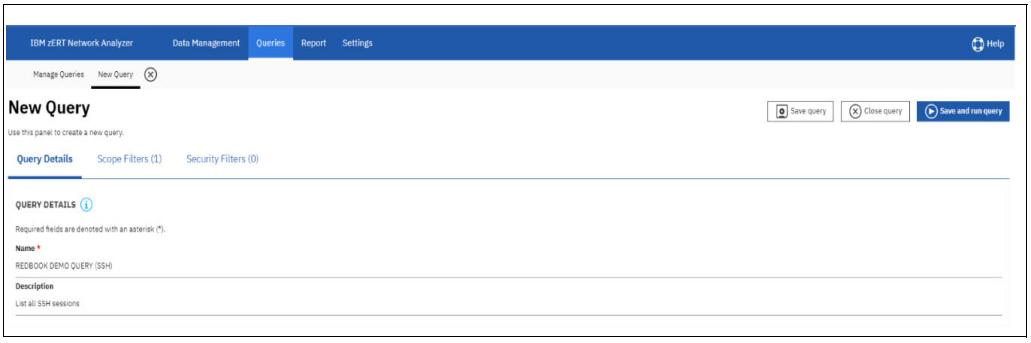



*Figure 5-4 IBM zERT Network Analyzer creating a new query*

By default, the query reports on all of the data that is available in the IBM zERT Network Analyzer database. However, to limit the output of the query and select specific information, you can add scope and/or security Filters.

**Scope Filters** can limit the output to a specific date range (such as, the last 30 days), the sysplex/systems/TCP/IP stack on which the security sessions were reported, the TCP or Enterprise Extender endpoints such as server IP address, server port range, and client?IP address (see Figure 5-5).

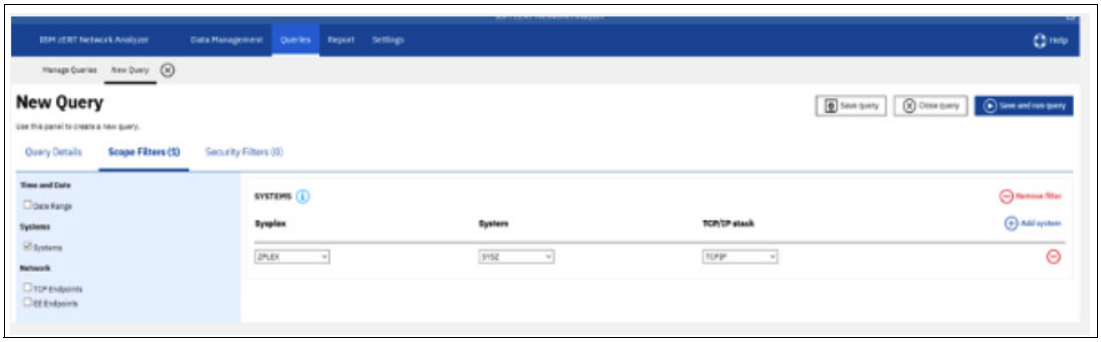



*Figure 5-5 IBM zERT Network Analyzer query scope filter*

**Security Filters** can limit the output of the query based on cryptographic protection characteristics for the security sessions. With Security Filters, you can select specific security protocol that you are interested in (unprotected, IPsec, TLS, and/or SSH) or limit based on protocol-specific attributes, such as encryption algorithm, message authentication algorithm, and so on (see Figure 5-6).

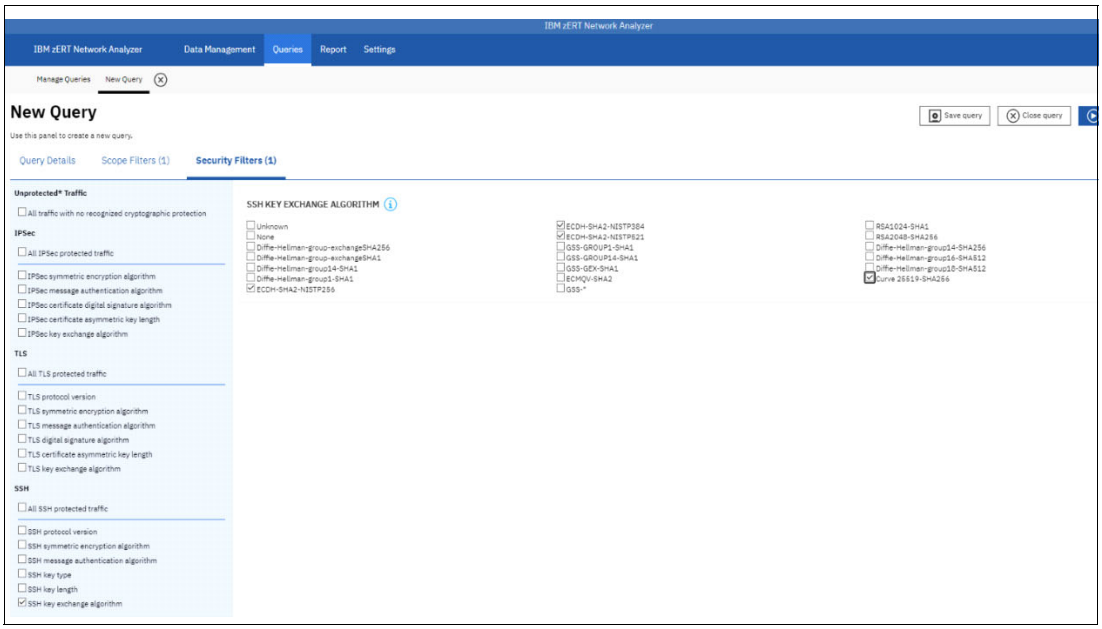



*Figure 5-6 IBM zERT Network Analyzer Security session filters*

2. After your query definition is complete, click **Save and run query**.

## Viewing the query result

The Report panel shows the results of the query that is made over zERT data. You can use the Report panel to view or export the report (see Figure 5-7 on page 84).

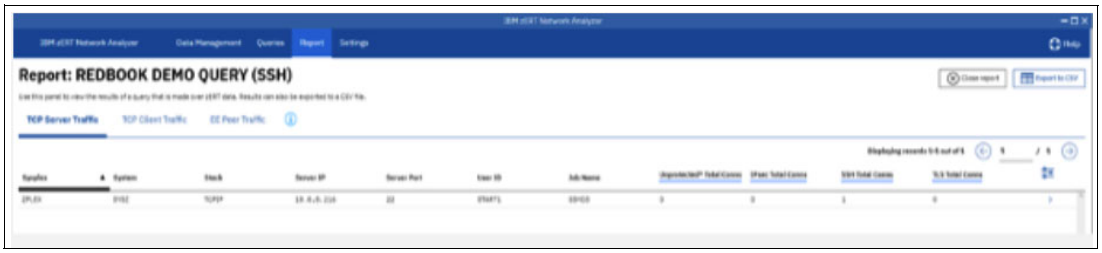



*Figure 5-7 IBM zERT Network Analyzer query result*

The Report panel displays a tabular summary organized by endpoint role of the local z/OS system: TCP server, TCP client or Enterprise Extender (EE) peer. Each summary row contains information about an endpoint with the Sysplex, System, Stack, and Server IP along with the ports, jobname (in the TCP server and EE peer tabs), and summary information about the type of session that was reported (unprotected, IPsec, SSH, and TLS).

To view details about TCP clients or foreign EE peers associated with the summary row, click on the summary row and it expands to show the details. The Client details view shows the list of Clients IP, jobname (TCP clients tab), along with a summary of the type of session.

You can get more information about the sessions for a specific client or all the clients by selecting the IP addresses that you want to analyze and then clicking **View security session details** (see Figure 5-8).


*Figure 5-8 IBM zERT Network Analyzer client details*

The Security Session Details view (see Figure 5-9 on page 85) summarizes the cryptographic details about the security sessions that are used by the connections. For each security protocol (TLS, IPsec, SSH, unprotected), you can view the Cryptographic Details (Negotiated Cipher, Authentication algorithm, etc.), Certificate Details (such as Key type, Key length, signature method etc.), Distinguished Name Details (Issuer and Subject) and Traffic Details, as applicable.


*Figure 5-9 IBM zERT Network Analyzer security session details*

If you need to save the data from the report before closing it, you can use the export function to export the results to a comma separated value (CSV) format file. The exported results contain all the summary and detail information of the report. Each column represents a unique field in the IBM zERT Network Analyzer database representation of a security session. The columns are described in *Export operation output file results*.

### Creating your own data-in-transit cryptographic inventory

From the previous section, we discovered that we can create targeted reports about specific cryptographic protection attributes by using the IBM zERT Network Analyzer. You can build a cryptographic inventory for your data-in-transit, with specific attributes, such as the clients/servers you are targeting, the security protocols or cryptographic algorithms, which is useful to prepare a transition for specific applications/client-servers in the enterprise.

## 5.2.3 Enforcing network security requirements with zERT policy-based enforcement

With z/OS 2.5, a feature called zERT policy-based enforcement (zERT enforcement) was introduced to enforce your network encryption standards and requirements based on the data collected by zERT discovery. With this feature, you can configure policy-based rules that describe connection attributes (such as local and remote IP addresses and ports, jobname, userid and connection direction), cryptographic protection attributes (security protocol, protocol versions, symmetric encryption, message authentication and key exchange algorithms) and any actions to take when the observed cryptographic attributes of TCP connections match those rules. The TCP/IP stack then compares each new TCP connection that terminates on that stack with the set of zERT enforcement rules. If the connection matches a rule, the actions associated with that rule are taken. If a connection is protected by multiple protocols, it could match to more than one rule. The actions you can specify in a zERT rule are to take a reporting action or to reset the connection. You can specify up to three reporting actions:

- Log a message to syslogd
- Log a message to the console which goes to the TCP/IP job log
- Write an audit record (SMF 119 subtype 11) with a new event type that indicates that the record was written as a result of zERT enforcement.

The default action, when none of the actions is specified, is to silently allow the connection.

Note that we have an aggressive message suppression logic to prevent message flooding. So, if it is important to have a record of every single match of a given zERT rule, you should use the AuditRecord action.

## Creating a policy

To create a zERT Enforcement Policy, it is recommended to use, the z/OSMF Network Configuration assistant for z/OS Communications Server (see Figure 5-10) as is done with the others policy agent-based policies (such as AT/TLS and IPsec).

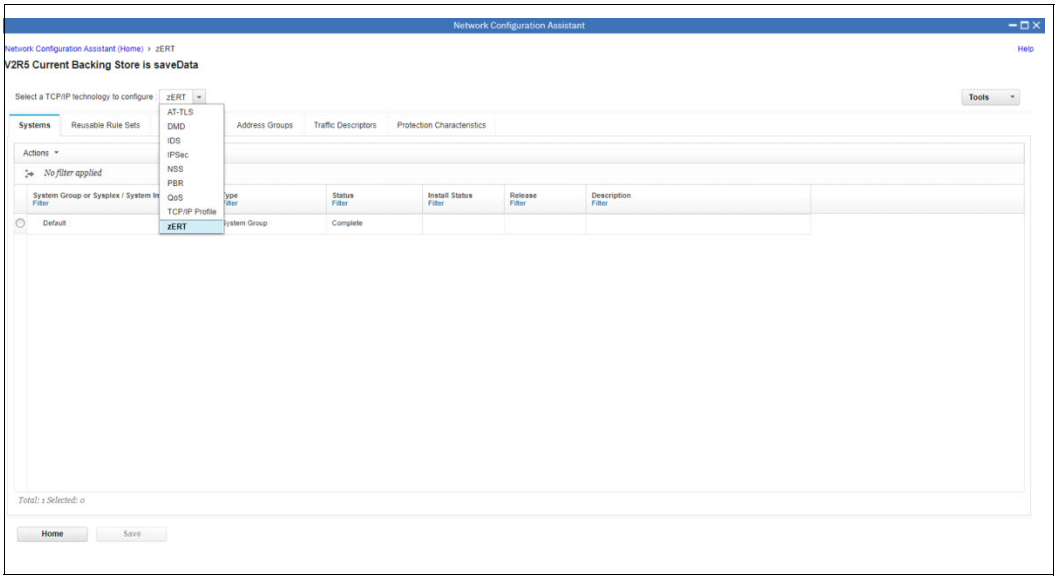



*Figure 5-10 IBM Network Configuration Assistant main panel*

You define a network security policy separately for each security protocol that is supported by zERT. The following security protocols are supported by zERT:

- TLS
- IPSec
- SSH
- No recognized protection

A single connection is evaluated against the zERT rules governing whichever security protocols are used for that connection.

For a given security protocol such as TLS:

- Create a single general rule with the highest priority, that describes the generally accepted levels of protection for example TLSv.1.3 or TLSv1.2 protocol versions, with acceptable symmetric encryption, message authentication and key exchange algorithms. The typical action here is to silently allow the connection because these protection attributes meet the standards. In Figure 5-11 on page 87, the AcceptableTLSRule zERT rule is an example of a general rule with acceptable protection attributes. It silently allows any connection using TLSv1.2 or TLSv1.3 with acceptable protection attributes.
- Next, create one or more specific rules with a lower priority for specific workloads. These can either be exceptions to the generally accepted standards that should be allowed or cases that should be flagged and/or blocked. For example, you could have a set of clients for a specific workload that you know are using a TLS protocol that is not generally accepted but the clients are in the process of being upgraded. For such exceptions, you might want to allow the connections without taking any action. There might also be some TLS protocols that should never be used. For such cases, the action to take might be reset the connection and log a message to the console so you can be notified of such cases. In Figure 5-11, the catchweakTLS rule is a specific rule for TLS connections using SSLv2, SSLv3, TLSv1.0 and TLSv1.1 protocol versions that are considered as weak protocols. The action taken is to reset the connection and log to the console.

- Finally, create a catch-all rule for any connection that does not match the general rule and does not match any of the specific rules defined. This rule defines what to do when zERT sees a connection that doesn't meet the other rules for this security protocol. A typical action for this might be to log the connection to syslogd and/or write an audit record to SMF. Our example in Figure 5-11 takes the audit action.

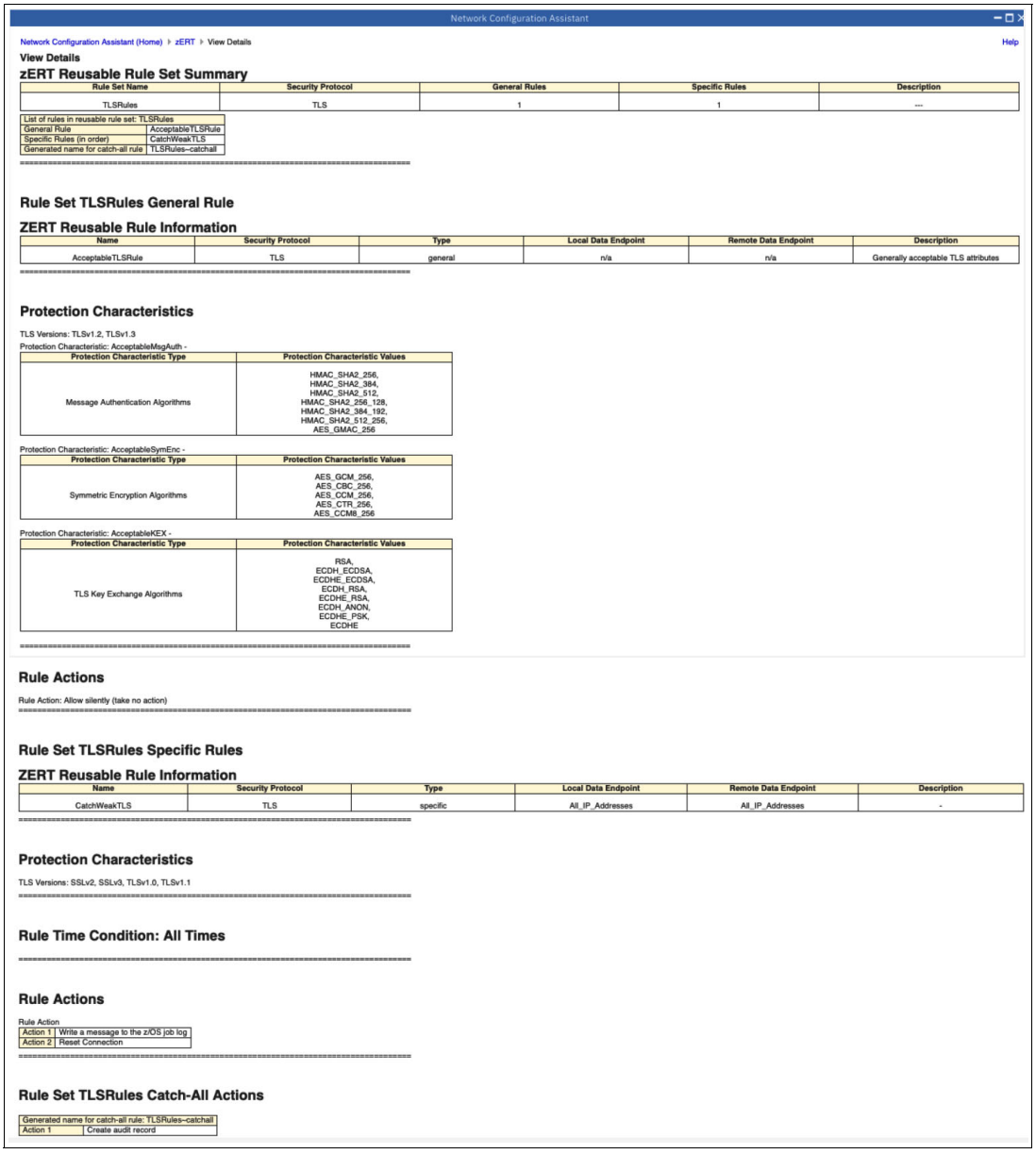



*Figure 5-11 zERT TLS rule set information*

**Tip:** Do not use the reset action for "no recognized protection" rules since there are several cases where transient no recognized protection sessions can be reported.

After activating the above policies through the policy agent, we can see in the MVS console (refer to Example 5-1) that message EZZ8771I is issued. We also see that when a TN3270 client starts a TLS connection using a weak TLS protocol version (SSLv3), the TCP/IP stack

matches the connection to the catchweaktls rule and the connection is reported to the console and the connection is reset (message EZZ8562I).

*Example 5-1 MVS console*

```
EZZ8771I PAGENT CONFIG POLICY PROCESSING COMPLETE FOR TCPIP : TTLS
EZZ8771I PAGENT CONFIG POLICY PROCESSING COMPLETE FOR TCPIP : ZERT
EZD1289I TCPIP ICSF SERVICES ARE CURRENTLY AVAILABLE FOR AT-TLS GROUP
AZFGroupAction1

EZZ6035I TN3270 DEBUG CONN   DETAIL 373
  IP..PORT: 10.0.0.111..53854
  CONN: 000056FA  LU: SYSZTCP8 MOD: EZBTTRCV
  RCODE: 1001-01  Client disconnected from the connection.
  PARM1: 00000000 PARM2: 00000000 PARM3: 00000000
EZZ6034I TN3270 CONN 000056FA LU SYSZTCP8 SESS DROP  CLNTDISC 374
  IP..PORT: 10.0.0.111..53854
IKT100I USERID           CANCELED DUE TO UNCONDITIONAL LOGOFF
IKT122I IPADDR..PORT 10.0.0.111..53854
EZZ6034I TN3270 CONN 000056FA LU SYSZTCP8 CONN DROP  CLNTDISC 376
  IP..PORT: 10.0.0.111..53854
EZZ6034I TN3270 CONN 0000571C LU **N/A**  ACCEPTED      23 378
  IP..PORT: 10.0.0.111..55623

EZZ8562I CONN RESET BY ZERT POLICY 379
EZZ8552I STACK= TCPIP CONNID= 0000571C CONNDIR= INBOUND
EZZ8553I LOCALIPADDR= 10.0.0.216 LOCALPORT= 23
EZZ8554I REMOTEIPADDR= 10.0.0.111 REMOTEPORT= 55623
EZZ8555I TRANSPROTO= TCP JOBNAME= TN3270 USERID= TCPIP
EZZ8556I SECPROTO= TLS SECPROTOVERSION= SSLv3
EZZ8557I SYMENC1= RC4_128 MSGAUTH1= HMAC_SHA1
EZZ8559I KEX= RSA
EZZ8560I RULE= catchweaktls
EZZ8561I ACTION= actlogreset
```

This example shows a simple policy, but during your quantum-safe journey, you can define rules to report on connections that use non-quantum safe encryption algorithms.

## 5.3 Using ICSF cryptographic usage tracking

Beginning with ICSF FMID HCR77C1, ICSF instances can be configured to collect cryptographic usage data when crypto operations are performed by that ICSF instance. ICSF creates an SMF record type 82, subtype 31 to aggregate crypto usage statistics for each job or user that is associated with the crypto usage in a specified period.

**Note:** It is essential to collect the records over a sufficient period, capturing as much workload and key usage as possible. This process helps build a more comprehensive cryptographic inventory.

ICSF cryptographic usage tracking (see Figure 5-12 on page 89) features the following options for collecting statistics:

- ENG: Crypto Express adapters, CPACF, and software
- SRV: ICSF callable services and UDXes

- ALG: Cryptographic algorithms that are used within ICSF crypto operations

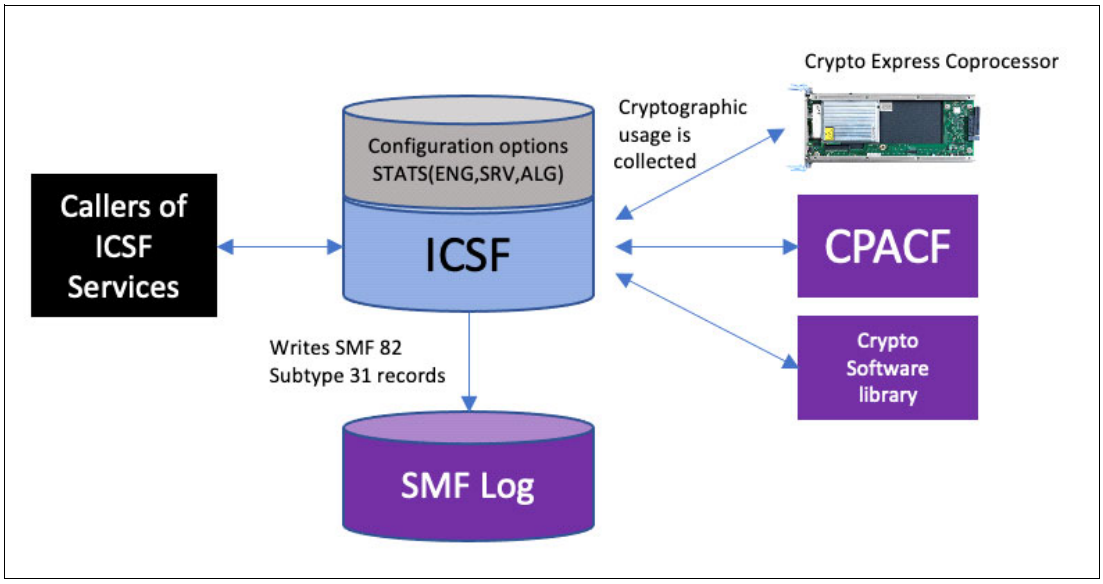



*Figure 5-12 ICSF cryptographic usage tracking overview*

## 5.3.1 Configuring SMF for ICSF cryptographic usage tracking

Before ICSF can write SMF records for cryptographic usage, the SMFPRMxx member in PARMLIB (see Example 5-2 on page 89) must be updated to contain the following components:

- The collection interval (INTVAL).

  Cryptographic usage tracking is synchronized to the SMF recording interval. In our example, ICSF records crypto usage every 5 minutes.

- The synchronization value (SYNCVAL).

  Synchronizes the recording interval with the end of the hour of the TOD clock. In our example, ICSF starts recording crypto usage at the end of the hour.

- The Cryptographic Usage Statistics subtype 31 for ICSF type 82 records (TYPE).

  Specifies the type of records to be recorded. In our example, SMF 82 subtype 31 is specified.

*Example 5-2 SMFPRMxx member example*

```
DSNAME(SMF.MANA,SMF.MANB)            /* SMF Data sets        */
INTVAL(05)                           /* INTERVAL – 5 minutes */
SYNCVAL(00)                    /* SYNCHRONIZATION – 0 minutes */
SYS(TYPE(0,2,3,82(31),83,128:132)) /* ICSF SMF 82 subtype 31 */
```

## 5.3.2 Enabling cryptographic usage tracking within ICSF

Cryptographic usage tracking can be enabled within ICSF by using one of the following methods:

- Through the installation options that are used for ICSF initialization

  The CSFPRMxx member in PARMLIB must contain the STATS option. As shown in Example 5-3, all three STATS options are enabled. STATS(ALG) must be specified to

enable algorithm usage tracking. STATSFILTERS(NOTKUSERID) can be optionally specified to exclude the task level user ID from the stats aggregation criteria. This option is intended for environments that features a high volume of operations that are running under task level user IDs, which reduces the number of SMF 82 Subtype 31 records written.

*Example 5-3 CSFPRMxx options*

```
CKDSN(SYS1.CKDS)
PKDSN(SYS1.PKDS)
TKDSN(SYS1.TKDS)
STATS(ENG,SRV,ALG)
STATSFILTERS(NOTKUSERID)
```

- Dynamically enable cryptographic usage tracking by using the **SETICSF** command (see Example 5-4).

*Example 5-4 SETICSF command*

```
SETICSF OPT,STATS=(ALG)
```

The STATS setting can be verified by using the **DISPLAY ICSF,OPT** command (see Example 5-5).

*Example 5-5 DISPLAY ICSF,OPT output*

```
D ICSF,OPT
 CSFM668I 12.44.17 ICSF OPTIONS 907
   SYSNAME = SY1          ICSF LEVEL = HCR77D2
     LATEST ICSF CODE CHANGE = 02/21/22
     Refdate update interval in Days/HH.MM.SS = 005/00.00.00
     Refdate update period   in Days/HH.MM.SS = 000/01.00.00
     MASTERKCVLEN = display ALL digits
     AUDITKEYLIFECKDS: Audit CCA symmetric key lifecycle events
       SYSNAME   LABEL    TOKEN
       SY1        Yes      Yes
     AUDITKEYLIFEPKDS: Audit CCA asymmetric key lifecycle events
       SYSNAME   LABEL    TOKEN
       SY1        Yes      Yes
     AUDITKEYLIFETKDS: Audit PKCS #11 key lifecycle events
       SYSNAME   TOKOBJ   SESSOBJ
       SY1        Yes      Yes
     AUDITKEYUSGCKDS: Audit CCA symmetric key usage events
       SYSNAME   LABEL    TOKEN     Interval Days/HH.MM.SS
       SY1        No       No                000/02.00.00
     AUDITKEYUSGPKDS: Audit CCA asymmetric key usage events
       SYSNAME   LABEL    TOKEN     Interval Days/HH.MM.SS
       SY1        No       Yes               000/01.00.00
     AUDITPKCS11USG: Audit PKCS #11 usage events
       SYSNAME   TOKOBJ   SESSOBJ   NOKEY  Interval Days/HH.MM.SS
       SY1        No       No        Yes             000/05.00.00
     STATS:
       SY1       ALG
     COMPLIANCEWARN: Compliance warning events
       SY1       PCI-HSM 2016      Yes
     TRACKCLASSUSAGE:
       SY1       NONE
```

### 5.3.3 Formatting cryptographic usage statistics records

After ICSF cryptographic usage tracking is enabled for algorithms, run applications on the ICSF instances with tracking enabled. As a result, SMF cryptographic usage records are generated.

ICSF provides formatters in SYS1.SAMPLIB (CSFSMFJ) that is the JCL that can be submitted to read SMF record type 82 and format them into a report. CSFSMFR is the REXX exec that is used to run the report against the SMF records.

A formatted report of SMF record type 82, subtype 31 (hex '001F') is shown in Example 5-6.

*Example 5-6 Formatted report of SMF record type*

```
Type=82 Subtype=001F Crypto Usage Statistics
Written periodically to record crypto usage counts
22 Feb 2022 15:12:27.73
   TME... 005389D5 DTE... 0122053F SID... SP21    SSI... 00000000 STY... 001F
   INTVAL_START.. 02/22/2022 19:11:30.001815
       INTVAL_END.... 02/22/2022 19:12:27.737573
USERID_AS.....DATAOWN
   USERID_TK.....
   JOBID.........J0000055
   JOBNAME.......DATAOWN
   JOBNAME2......
   PLEXNAME......SYS1
   DOMAIN........0
   ENG...CARD...8C11/99EA6127...17
   ENG...CPACF...150
   ALG...DES56......2
   ALG...AES128.....2
   ALG...RSA1024....1
   ALG...ECCBP192...1
   ALG...MD5........45
   ALG...RPMD160....15
   ALG...SHA1....... 70
   ALG...SHA3-224... 13
   ALG...SHA3-256... 15
   ALG...SHA3-384... 13
   ALG...SHA3-512... 13
   ALG...SHAKE128... 12
   ALG...SHAKE256... 14
   SRV...CSFKYT..... 2
   SRV...CSFDSG..... 2
   SRV...CSFOWH..... 264
   SRV...CSFOWH1.... 3
   SRV...CSFIQF..... 485
   SRV...CSFIQF2.... 2
************************************************
```

#### Interpreting cryptographic usage statistics SMF records

After you formatted the cryptographic usage statistics SMF records, you can begin identifying algorithms that are used in applications that must be replaced.

In Example 5-6 on page 91, algorithms DES56 and RSA1024 were used in crypto operations within the SMF recording interval. The SMF record lists the HOME address space ID or HOME address space job name, which are the job or task that started the cryptographic request.

The SMF record also can list the SECONDARY address space job name (for example, the caller that made the program call or space switch to ICSF), the HOME address space user ID, and the task level user ID if available.

The usage event is recorded for jobname DATAOWN. It occurred on system SYS1 and used crypto domain 0.

DES56, RSA1024, and SHA-1 are examples of weak algorithm candidates to prioritize for migration in your cryptographic inventory.

# 5.4 IBM z/OS SMF record type 30 cryptographic counters

The SMF record type 30 was enhanced to report CPACF cryptographic instruction use counter values supported by the system. The SMF record type 30 can be generated by using a function provided with z/OS 2.5 and APAR OA61511or later versions of z/OS.

## 5.4.1 Checking, enabling, and disabling the function.

The functions is enabled and disabled through an IEASYSxx parameter at IPL or with a **START IEACTRS** command.

You can check if the data gathering function for the crypto counters is active using the z/OS command:

```
DISPLAY IPLINFO,CRYPCTRS,STATE
```

When the function is active, the following output is expected:

```
IEE261I CRYPCTRS State: Active
```

The cryptographic counters gathering function can be disabled by issuing the command:

```
START IEACTRS,CRYPTO=INACTIVE
```

Or the function can be (re)enabled using the command:

```
START IEACTRS,CRYPTO=ACTIVE
```

## 5.4.2 Processing SMF record type 30

The SMF record type 30 can be processed by identifying and locating the cryptographic counters offsets using the SMF30CPO, SMF30CPL, SMF30CPN, and SMF30CPA entry names from the records (see Figure 5-13 on page 93). Once identified and located, the cryptographic counters can be extracted.

| Offset (dec) | Offset (hex) | Name | Length | Format | Description |
|---|---|---|---|---|---|
| 190 | BE | SMF30USN | 2 | binary | Number of zEDC usage statistics sections. |
| 192 | C0 | SMF0_End_V1 | 0 | n/a | End of version 1. |
| 192 | C0 | SMF30CPO | 4 | Binary | Offset to the CrypCtrs section |
| 196 | C4 | SMF30CPL | 2 | Binary | Length of the CrypCtrs section |
| 198 | C6 | SMF30CPN | 2 | Binary | Number of CrypCtrs sections |
| 200 | C8 | SMF30CPA | 4 | Binary | Number of CrypCtrs sections in subsequent (continuation) records |
| 204 | CC | SMF30NPO | 4 | Binary | Offset to the NNPICtrs section |
| 208 | D0 | SMF30NPL | 2 | Binary | Length of the NNPICtrs section |
| 210 | D2 | SMF30NPN | 2 | Binary | Number of NNPICtrs sections |
| 212 | D4 | SMF30NPA | 4 | Binary | Number of NNPICtrs sections in subsequent (continuation) records |
| 216 | D8 | SMF30_Cont_Recs_To_Follow | 2 | Binary | The number of continuation records to follow this record. When SMF30_RecCont_LastRec is on, this field will be zero. |
| 218 | DA | SMF30_RecCont_Flags | 1 | Binary | Record continuation flags<br>Bit 0 - SMF30_RecCont_FirstRec - When on, this record is the first of a set of two or more continuation records<br>Bit 1 - SMF30_RecCont_AdditionalRec - When on, this record is the second or subsequent but not last record in a set of two or more continuation records.<br>Bit 2 - SMF30_RecCont_LastRec - When on, this record is the last of a set of two or more continuation records |
| 219 | DB | SMF0_End_V2 | 0 | n/a | End of version 2. |

*Figure 5-13 SMF record type 30 cryptographic counters*

The SMF30_CrypCtrs_Entry_ID is a reference to the called instruction. This entry identifier can be found in the IFASMFCN macro instruction from SYS1.MACLIB (see Figure 5-14).

The SMF30_CrypCtrs_Count is the number of times the cryptographic instruction has been called.

| Offsets | | Name | Length | Format | Description |
|---|---|---|---|---|---|
| 0 | 0 | SMF30_CrypCtrs_Entry_ID | 2 | binary | Crypto counter entry identifier. |
| 2 | 2 | SMF30_CrypCtrs_Count | 8 | binary | Crypto counter count value. This field contains the instruction counts used within the scope of the record, either for the current interval, job step, or job. |

*Figure 5-14 SMF30_CrypCtrs_Entry_ID*

For a sample report for SMF record type 30 refer to A.2, “z/OS SMF record type 30 sample output” on page 146.

## 5.5 Using IBM Application Discovery and Delivery Intelligence

IBM ADDI (see Figure 5-15 on page 94) is an analytics platform for mainframe application modernization. It identifies and visualizes application dependencies and helps you quickly understand the impact of changes.

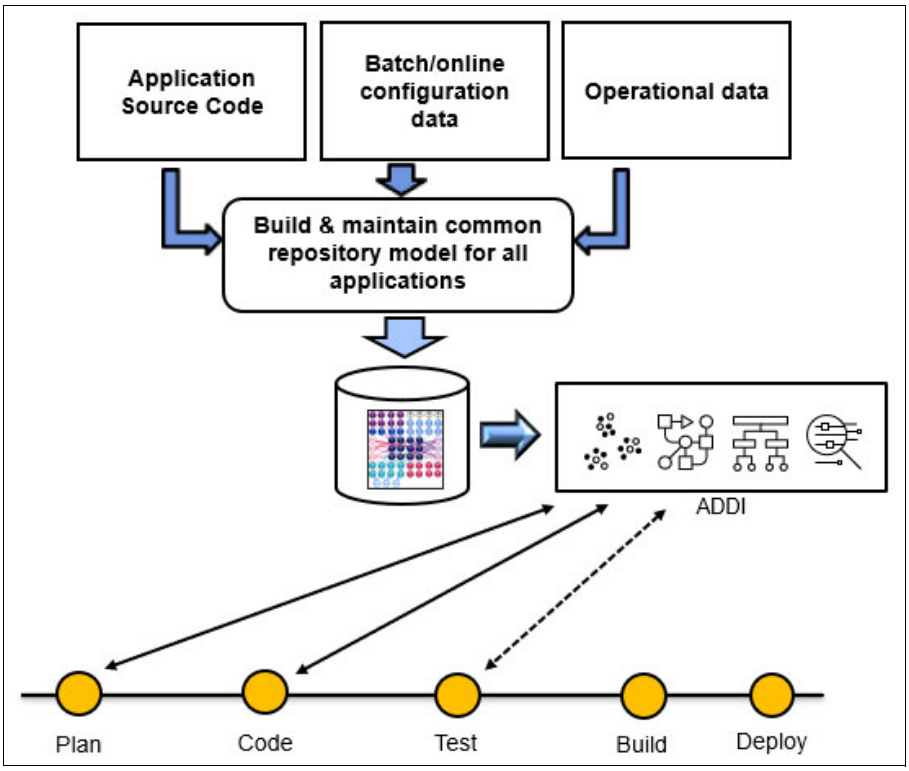



*Figure 5-15 IBM ADDI flow*

IBM Application Discovery Build Client is part of the IBM ADDI product suite. By using IBM Application Discovery Build Client to perform application crypto analysis, you can realize the following benefits:

- Efficiently locate and identify where crypto is used in applications.
- Capture valuable metadata and dependencies by identifying important ICSF parameters, such as “Rule Array”.
- Store analysis results in a repository.
- Plan migration and modernization efforts.
- React quickly to potential security issues.

**Note:** As of this writing, this support is available for COBOL applications only.

## 5.5.1 Configuring IBM AD Build Client for ICSF crypto analysis

The following steps assume that the installation and configuration process is complete for ADDI. For more information about this process, see this IBM Documentation web page.

Complete the following steps to configure and run IBM Build Client against your COBOL application:

1. Verify that the CRYPTO resolutions file (`CAPIResolutions.json`) is available in the `\bin\release` folder. This folder is where the AD Build Client executable (`IBMApplicationDiscoveryBuildClient.exe`) is stored.

   The following default path is used by the installation:

```
C:\Program Files\IBM Application Discovery and Delivery Intelligence\IBM
Application Discovery Build Client\bin\release
```

**Note:** After the installation is complete, `CAPIResolutions.json` can be found in the `\bin\release\Samples` folder. Copy the `.json` file and place it in the `bin\release` folder.

2. Open the AD Build Client tool and create a project by clicking **File** → **New** → **New Project** (see Figure 5-16).

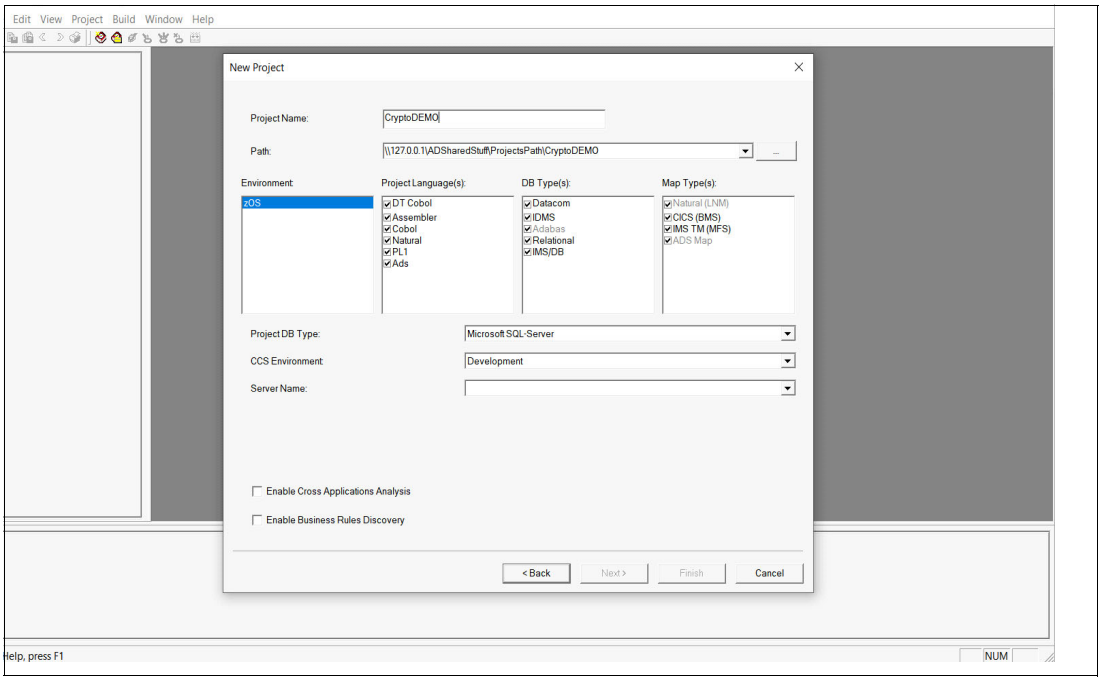



*Figure 5-16 Creating an IBM AD Build client*

3. Upload your COBOL application files to the project. You should downloaded these files from your mainframe. These COBOL files are scanned and analyzed for CALL statements, which are calls to ICSF cryptographic services. Complete the following steps:
   a. Right-click **zOS Cobol** project folder → **Add Files**.
   b. Locate the files on your local machine, select them, and click **OK**.
   c. Repeat step 2 until all application files are added to the project.

d. After the project tree is populated, click **File** → **Save Project** (see Figure 5-17).

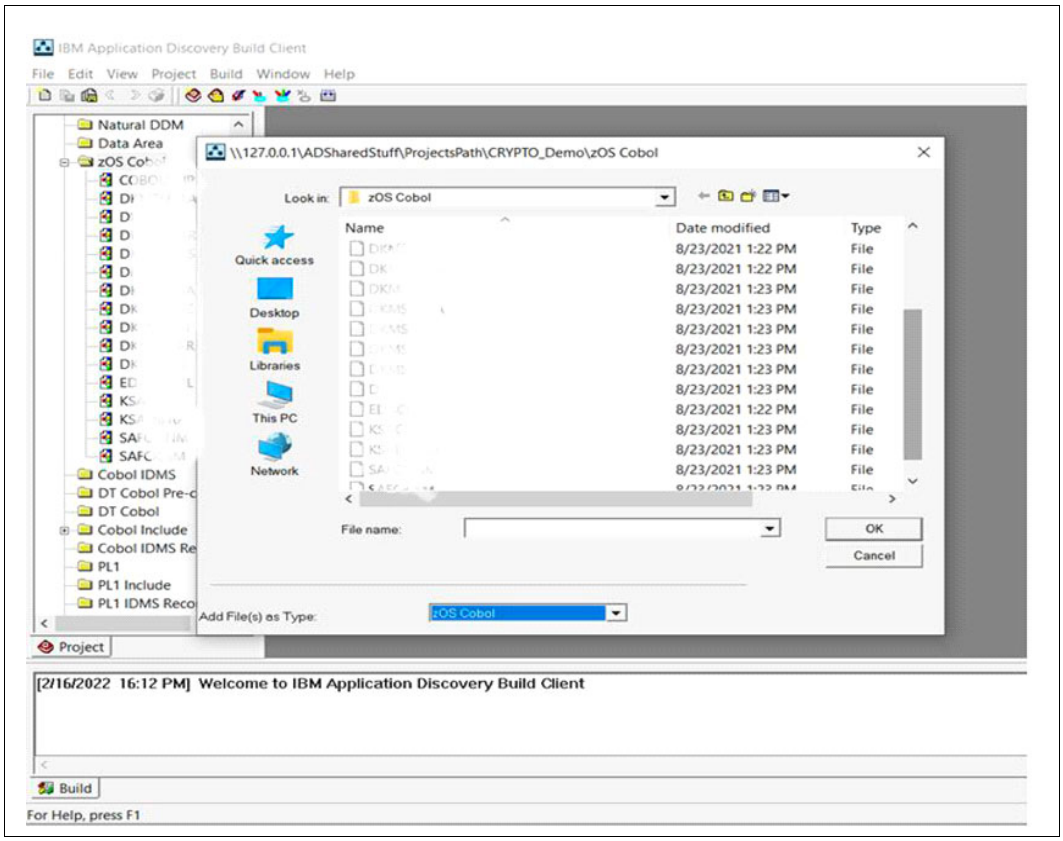



*Figure 5-17 IBM AD add COBOL application window*

4. Build the project by clicking **Build** → **Build Project**. Then, wait until the build process completes. Two files are generated that are related to ICSF crypto CALLs:
   - `GenericAPI_<timestamp>.csv`
   - `GenericAPI_<timestamp).html`
5. Locate the Project path on disk and the change directory to:

   `<Project path>/Reports/GenericAPI.`

   For example:

   `C:\IBM AD\Mainframe Projects\<your_project_name>\Reports\GenericAPI`

   The `.csv` file can be read as is or used as input by other tools as raw input.

   The `.html` file can be used as a report where the results can be filtered according to various criteria.

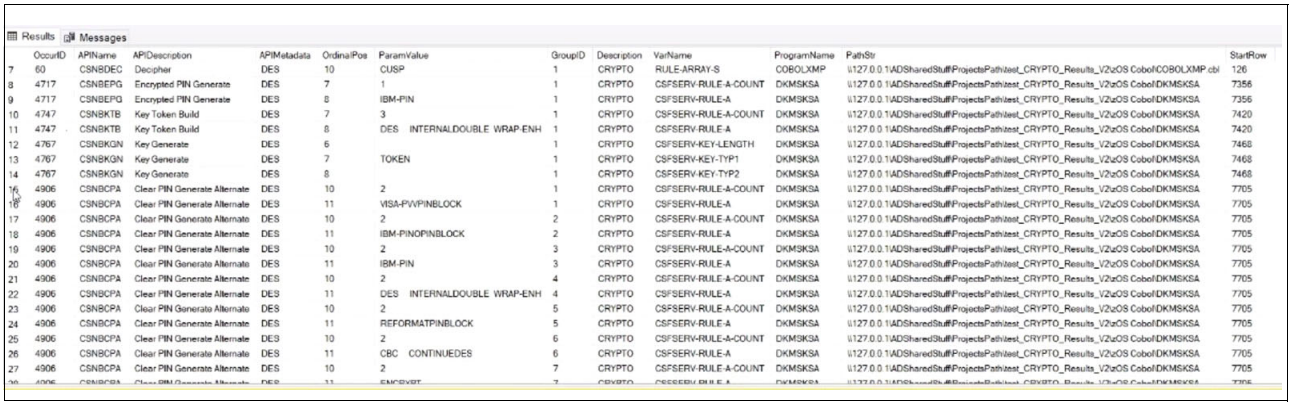
## 5.5.2 Interpreting IBM AD Build Client file results

In Figure 5-18, each CALL statement is uniquely identified by an ID in column A (OccurID); CALLs to ICSF CRYPTO service are in column B (APIName). The file where the CALL is stored is in column K (PathStr) at the line in column L (StartRow).

The CALLs parameter values are in column G (ParamValue) and form one of more tuples in column H (GroupID), which correspond to the parameter positions column F (OrdinalPos) that is specified in the `CAPIResolutions.json` for the respective service.

If no CALL statements to ICSF cryptographic services are found within the COBOL source files, an empty report is generated with the message:

```
GenericAPI query returned no results.
```



Results Messages

| | OccurID | APIName | APIDescription | APIMetadata | OrdinalPos | ParamValue | GroupID | Description | VarName | ProgramName | PathStr | StartRow |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 60 | CSNBDEC | Decipher | DES | 10 | CUSP | 1 | CRYPTO | RULE-ARRAY-S | COBOLXMP | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\COBOLXMP.cbl | 126 |
| 8 | 4717 | CSNBEPG | Encrypted PIN Generate | DES | 7 | 1 | 1 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7356 |
| 9 | 4717 | CSNBEPG | Encrypted PIN Generate | DES | 8 | IBM-PIN | 1 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7356 |
| 10 | 4747 | CSNBKTB | Key Token Build | DES | 7 | 3 | 1 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7420 |
| 11 | 4747 | CSNBKTB | Key Token Build | DES | 8 | DES INTERNALDOUBLE WRAP-ENH | 1 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7420 |
| 12 | 4767 | CSNBKGN | Key Generate | DES | 6 | | 1 | CRYPTO | CSFSERV-KEY-LENGTH | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7468 |
| 13 | 4767 | CSNBKGN | Key Generate | DES | 7 | TOKEN | 1 | CRYPTO | CSFSERV-KEY-TYP1 | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7468 |
| 14 | 4767 | CSNBKGN | Key Generate | DES | 8 | | 1 | CRYPTO | CSFSERV-KEY-TYP2 | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7468 |
| 16 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 1 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 16 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 11 | VISA-PVVPINBLOCK | 1 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 17 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 2 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 18 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 11 | IBM-PINOPINBLOCK | 2 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 19 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 3 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 20 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 11 | IBM-PIN | 3 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 21 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 4 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 22 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 11 | DES INTERNALDOUBLE WRAP-ENH | 4 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 23 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 5 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 24 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 11 | REFORMATPINBLOCK | 5 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 25 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 6 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 26 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 11 | CBC CONTINUEDES | 6 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |
| 27 | 4906 | CSNBCPA | Clear PIN Generate Alternate | DES | 10 | 2 | 7 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | \\127.0.0.1\ADSharedStuff\Projects\Pathvtest_CRYPTO_Results_V2\zOS Cobol\DKMSKSA | 7705 |

*Figure 5-18 IBM AD Build client results example*

When reviewing the results, pay attention to the APIMetadata and ParamValue columns. These columns provide insight into the type of algorithms that are used within the ICSF crypto service call. When building your cryptographic inventory, consider adding programs that contain ICSF calls with weak algorithms, such as DES.

## 5.5.3 Interpreting the CRYPTO CAPIResolutions.json resolutions file

The CRYPTO Resolutions (`CAPIResolutions.json`) file is a configuration file with the possible API calls, the parameters to capture, and information about ICSF calls. The ADDI build process uses this file to capture the data and can be extended or customized (see Example 5-7).

*Example 5-7 CRYPTO Resolution file snippet example*

```
{ "formatVersion":"1.0",
…
"GenericAPIs":
[
    ...
      { "GenericAPIName":"CSNBENC1,CSNEENC1",
        "Description":"Encipher",
        "Metadata":"DES",
        "Metadata1": "Encryption/Decryption",
        "Parameters":
         [
            { "Name":"RULE_ARRAY_COUNT",
```

```
                "Position":9,
                "DefaultValues":"0" },
             { "Name":"RLE_ARRAY",
                "Position":10,
                "DefaultValues":"INITIAL, TOKEN" }
         ]
      }
    ...
 ]
}
```

Where:

- `GenericAPIName`: Name of the ICSF service (if that service has several names, you can initialize it with all of the values separated by comma)
- `Description`: Short description of the service
- `Metadata`: Algorithm used by that service
- `Metadata1`: Category to which that service belongs
- `Parameters`: Array of parameter objects where each object includes the following information:
  - `_Name`: Parameter name (for example, for the Rule Array Parms column: `RULE_ARRAY_COUNT` and `RULE_ARRAY` names are used. For any Other Parms, `PARAM<No>` is used; for example, `PARAM6`, `PARAM7`, and `PARAM8`).
  - `_Position`: Position of that parameter for which we want to collect its values.
  - `_DefaultValues`: Comma-separated default parameter values, if applicable.

## 5.5.4 Extending the CRYPTO CAPIResolutions.json resolutions file

If you use an abstraction layer within the COBOL source files that is built on top of the standard cryptographic libraries (CCA and ICSF), the `CAPIResolutions.json` file must be edited and extended with the client interfaces (see Example 5-8).

*Example 5-8 Custom CAPIResolutions.json file*

```
{
     "GenericAPIName":"MYENCPHR",
     "Description":"Custom Encipher",
     "Metadata":"DES",
     "Metadata1": "Custom Implementation of Encryption/Decryption",
     "Parameters":
      [
         { "Name":"RLE_ARRAY_COUNT",
            "Position":9,
            "DefaultValues":"0"
         },
        { "Name":"RLE_ARRAY",
           "Position":10,
           "DefaultValues":"INITIAL, TOKEN"
         }
      ]
}
```

# 5.6 Using IBM Crypto Analytics Tool

The IBM Crypto Analytics Tool (IBM CAT) can provide a comprehensive overview of the Enterprise Cryptographic entities and their usage.

IBM CAT collects data from desired z/OS LPARs where cryptographic entities and usage are in scope for analysis and validation against Enterprise polices. The findings are presented in generated reports.

IBM CAT includes a policy rules engine that can be used to analyze the extracted data and flag whether objects are compliant or noncompliant according to the policy rules (such as reporting non-quantum-safe keys).

## 5.6.1 IBM CAT overview

IBM CAT features the following components (see Figure 5-19):

- A data collection for z/OS that consists of load modules and compiled REXX execs to extract cryptographic and security-related information from z/OS systems.
- Data is collected per LPAR in scope, locally captured in datasets for later load into the IBM CAT Db2 database as “snapshots”.
- The IBM CAT Db 2 database may be loaded with data from several z/OS systems.
- The Crypto Analytics Tool monitor client is a workstation program that accesses the Crypto Analytics Tool database by using a JDBC connection to generate the reports.

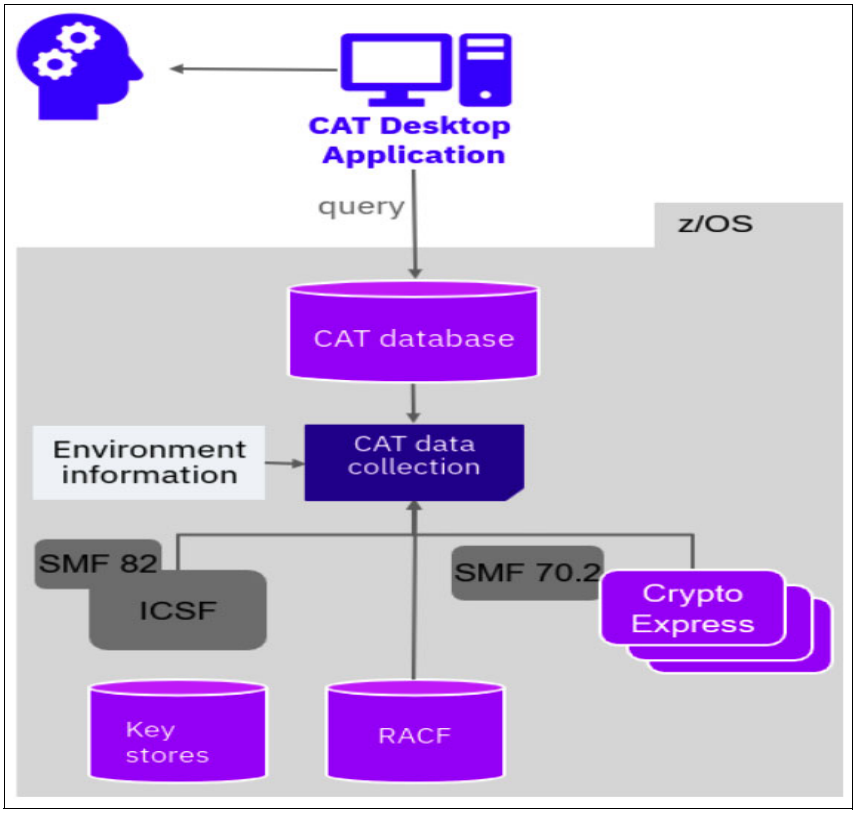



*Figure 5-19 IBM CAT architectural overview*

## 5.6.2 Reported elements

Whenever you run the IBM CAT data collection and load the IBM CAT Db2 tables, these are populated with the results of the IBM CAT data collection. And now you essentially have a cryptographic inventory that you can work with.

An IBM CAT data collection is called a snapshot. The following elements are reported by IBM CAT in a snapshot (see Figure 5-20.

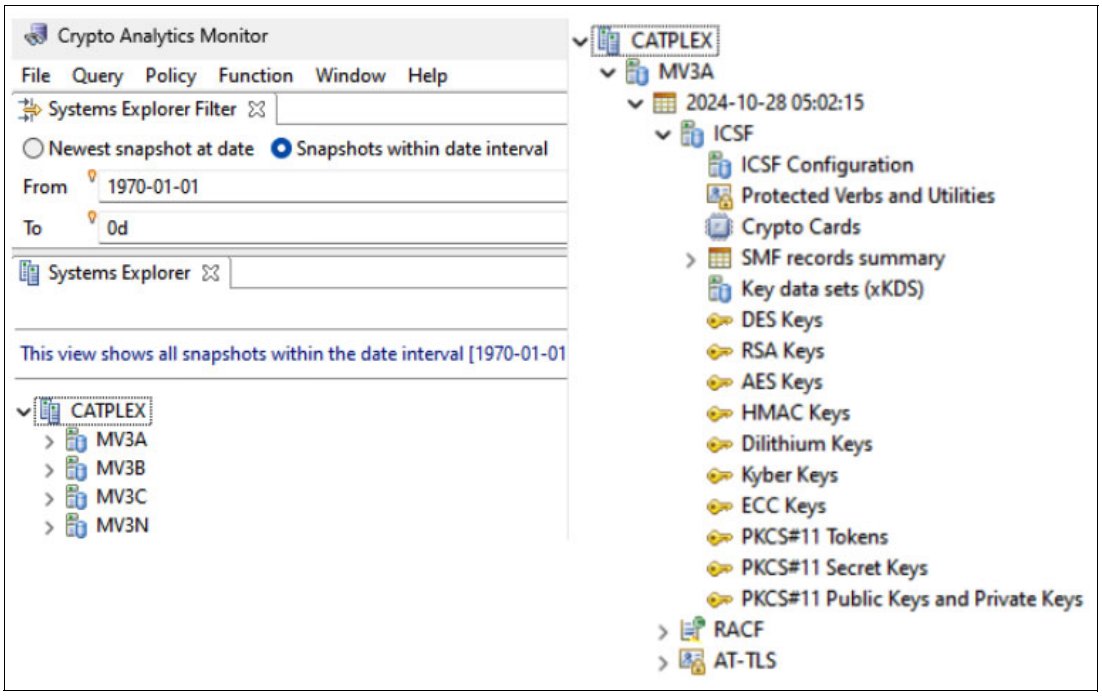



*Figure 5-20 IBM CAT snapshot*

IBM CAT Monitor showing ICSF in a sysplex with 4 systems, one having a snapshot open for investigation:

- ICSF configuration

  The ICSF configuration report provides basic information about the ICSF status at the time of the snapshot (sysplex mode, initialization status, CKDS format, and so on). The report also provides the status of runtime options (such as special secure mode and PCF coexistence), the ICSF policies, exit routines, and UDX services. SMF records type 82 subtypes 31 – 40 – 41 – 42 – 44 – 45 and 46 report cryptographic activity made via ICSF.

- Protected verbs and utilities

  The protected verbs and utilities report lists all the ICSF services and extracts from RACF the profiles that are protecting these services along with the access list to the profiles.

- Crypto Express

  This report provides a list of adapters that are assigned to the LPAR where the snapshot was taken, along with the status of the master keys. The report also provides the access points (ACPs) setup. Cryptographic hardware activity is captured in the SMF record type 70.2 – these are available in only the IBM CAT database and not reported in the IBM CAT monitor.

- Key data sets (xKDS)

  The Key data sets report provides the name of the xKDS along with the status of their RACF protection. It also includes the Masters Keys verification patterns.

  The key dataset must be in KDSLR in order to hold ML-KEM and ML-DSA keys. Currently IBM CAT does not distinguish between KDSR and KDSRL

- Key types

  For each type of key (DES, RSA, AES, and so on), this report lists all the keys (with a search option) and their type, size, metadata, and more. For each key, their RACF protection profiles can be displayed along with the options for the ICSF segment, such as SYMCPACFWRAP.

- RACF

  The RACF report provides the list of all the RACF profiles in the classes that are related to cryptography, such as CRYPTOZ and CSFSERV. The report also provides the list of users and groups with access to these profiles, along with the certificates and key rings that are present in RACF.

- AT-TLS

  The AT-TLS report provides the AT-TLS configuration with all entities listed, Rules, Scope, Statement, Keyword and values.

### 5.6.3 Monitoring functions

IBM CAT provides the following monitoring functions:

- Queries

  In CAT Monitor, predefined queries can be performed. Two types of queries are available: Queries that compare two selected snapshots, and queries that generate a report for a selected snapshot.

  For example, if you compare two different snapshots after a key generation event by using the comparison function, the comparison report highlights the new keys that are found in the keystore.

- Policy check

  With the policy check, set of policy rules can be specified that can be used to analyze the extracted data and flag whether the objects are compliant or noncompliant according to the policy rules. The set of policy rules can be tailored according to your own policy.

### 5.6.4 IBM CAT use case

In this use case, we show how you can use IBM CAT to generate a policy report to identify non-Quantum-safe keys in your ICSF keystore. For example, analyzing AES keys that are stored in the ICSF CKDS and report any policy inconsistency, such as a weaker AES 128 key, when your preference is to use a AES 256 key.

For this example, it is assumed that the IBM CAT data collections are loaded into the IBM CAT database and represent valid snapshots from relevant LPARs. The IBM CAT Monitor is installed on a workstation, the JDBC license is copied into the <CATMonitor>\configuration\license directory, and the Db2 connection information is provided.

First, we activate the default policy, verify the content of the policy for AES keys and then, apply it to our AES 256 keys.

## Activating the policy

We enable the built-in Policy. On the workstation where IBM CAT is installed by selecting **Window** → **Preferences**.

By default, the built-in policy is not activated. To activate it, select the policy and click **Set Active** and then, **Apply** (see Figure 5-21).

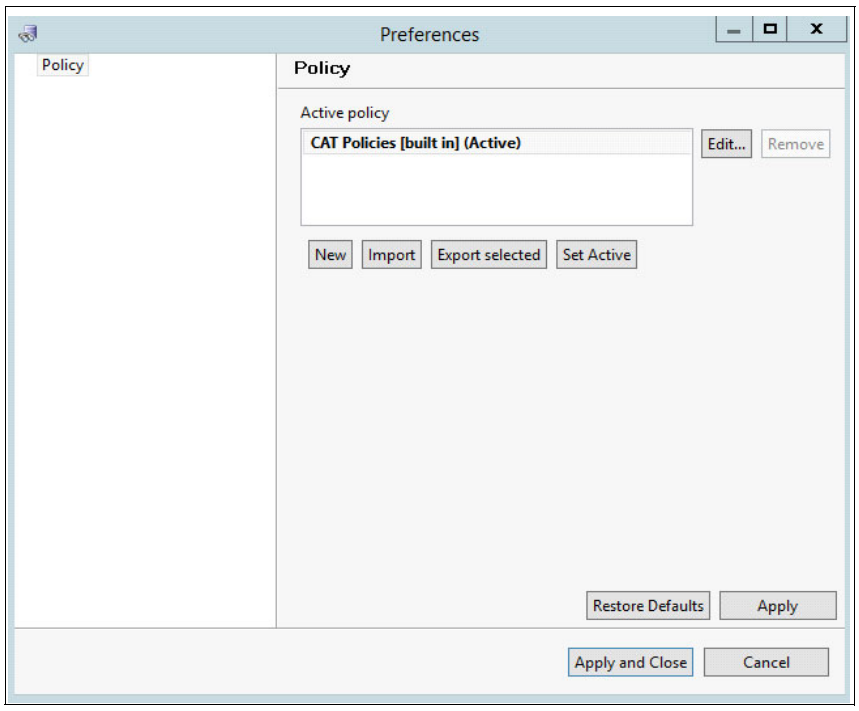



*Figure 5-21 IBM CAT Policy Activation*

## Checking the policy

To verify the content of the policy, select the policy and then, click **Edit**.

The policy editor is displayed (see Figure 5-22), which includes the name of the policy, a description, and a tab for each of the cryptographic elements where a policy can be applied.

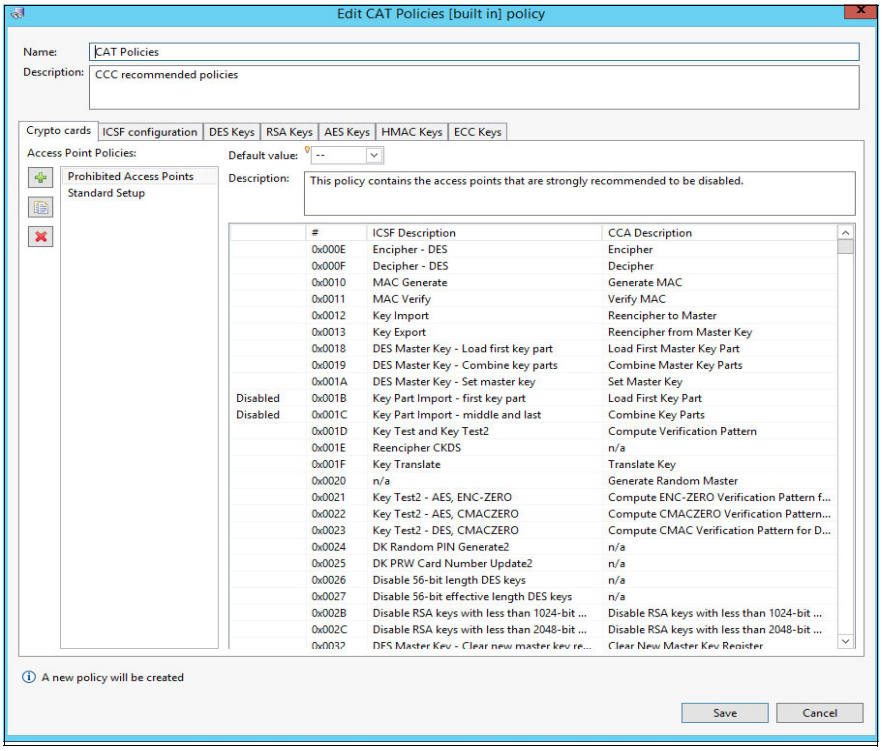



*Figure 5-22 IIBM CAT Policy Editor*

If we select the **AES Keys** tab, we can see that the default policy reports any AES128 key that is created after 2004-01-01, any AES192 key created after 2011-01-01, and so on.

As shown in Figure 5-23, the policy also reports any keys with identical Key Check Value (KCV); that is, those key labels have the same key value.

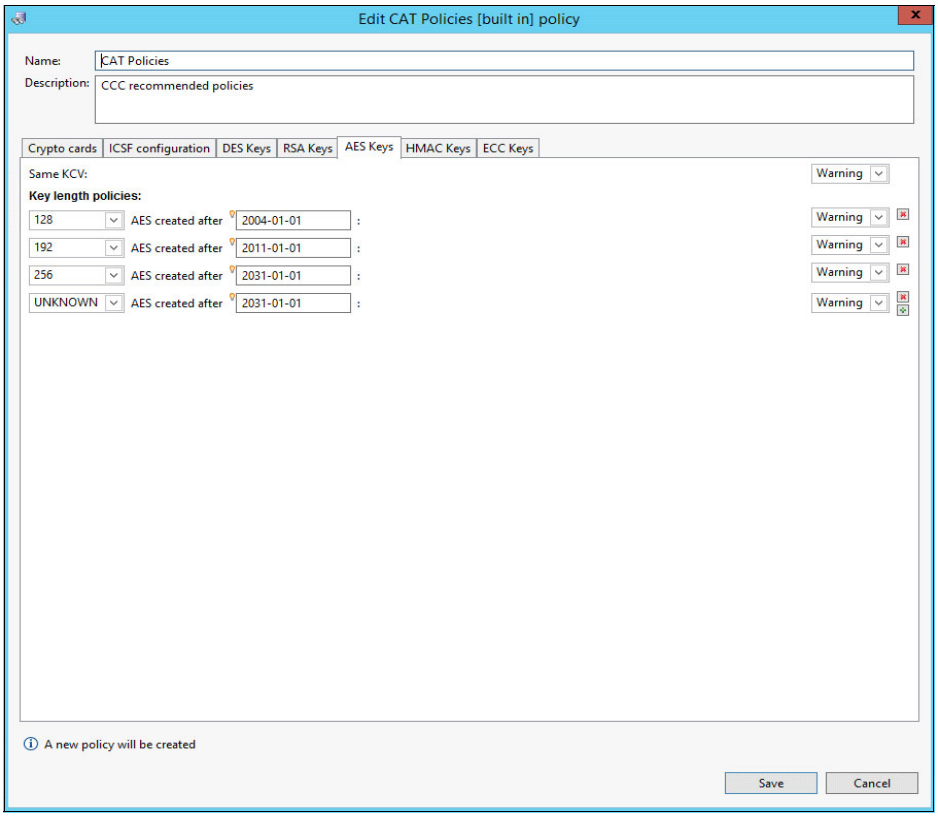



*Figure 5-23 AES Key Length Policies*

Exit the Preference dialog by clicking **Cancel** twice.

## Applying the policy to a snapshot

When back in IBM CAT main window, select and expand a snapshot in the Systems Explorer frame.

In the ICSF section, the types of keys that are present in the key data sets are listed. As shown in Figure 5-24, we right-clicked **AES Keys** and then, selected **Policy** → **Verify AES Keys**.

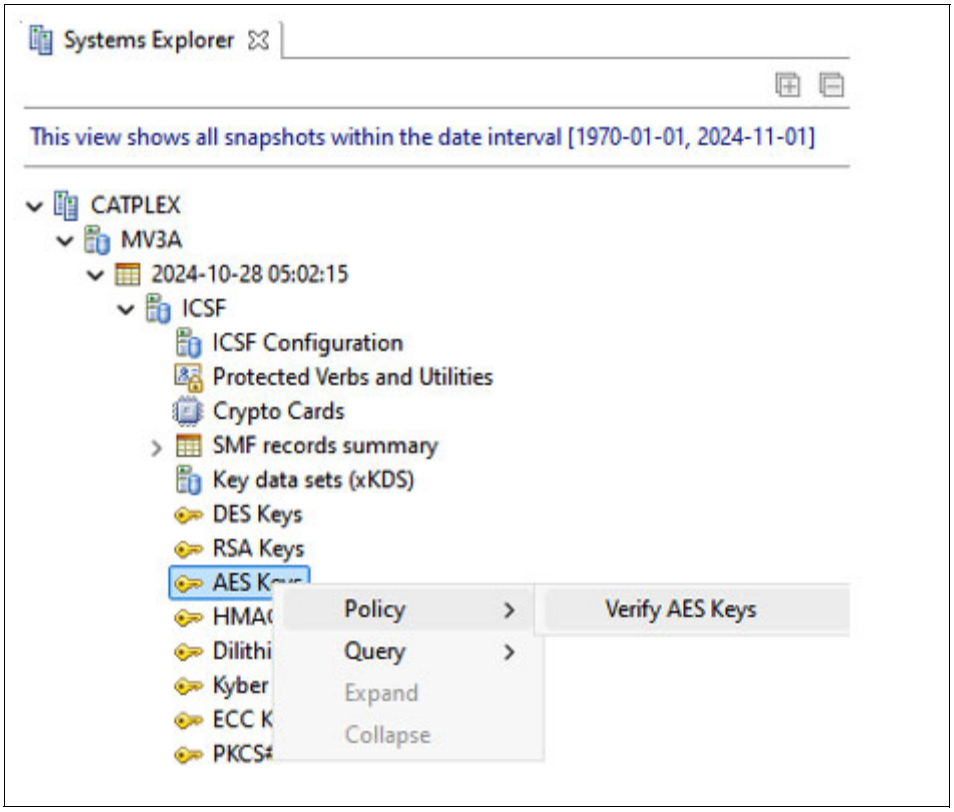



*Figure 5-24 Running the AES Key Policy*

The Policy Check report is displayed. In Figure 5-25 on page 106, the Policy Check Report is highlighting that some noncompliant AES keys were found in our keystore and some keys having the same KCV, such as the same key token in the keys despite they appear under different key labels in the keystore.

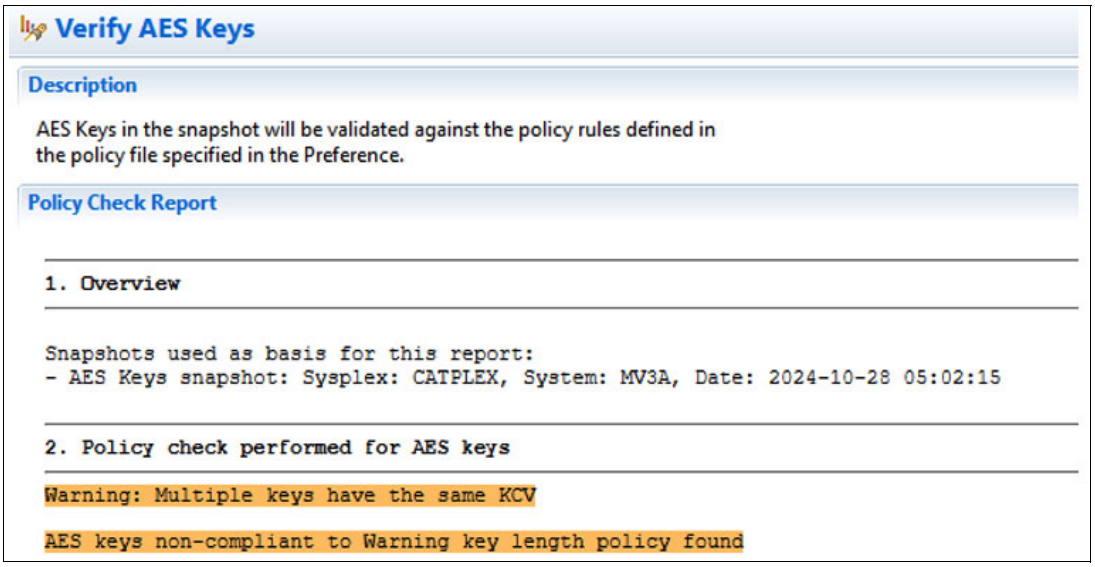



*Figure 5-25 IBM CAT Monitor AES Policy Check Report*

By clicking the highlighted key length policy exception, the list of keys (see Figure 5-26) shows that 211 keys were found as not compliant in the policy check report. When selecting each key, details about the key and the associated metadata are displayed.

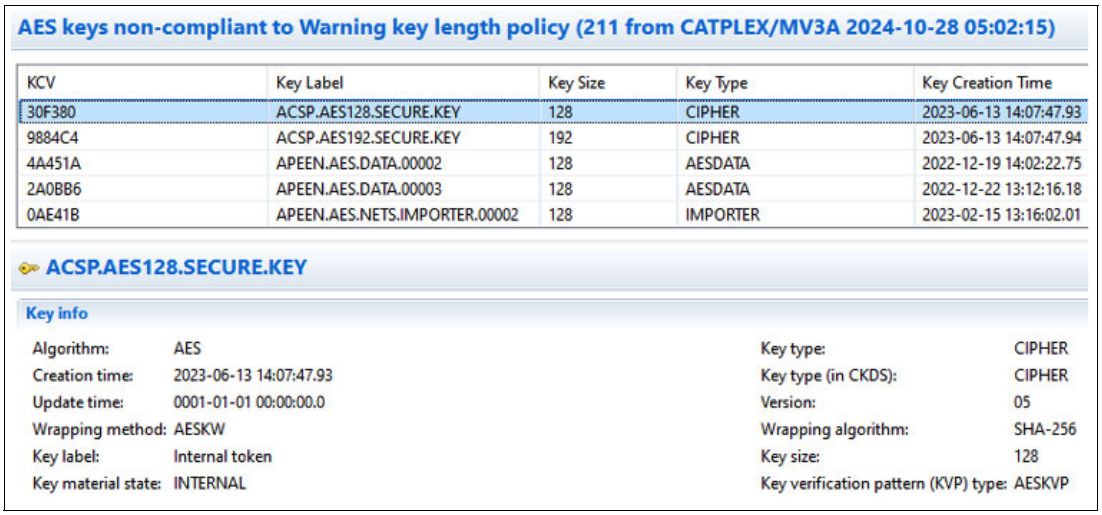
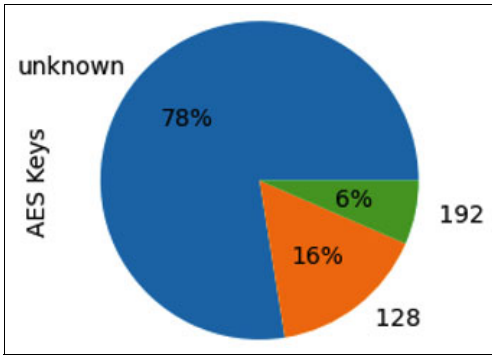



*Figure 5-26 AES non-policy-compliant keys display*

If we query the IBM CAT database and use for example a Jupyter Notebook to work with the query results, we can see how the non-compliant length AES keys and unknown length AES keys are distributed (see Figure 5-27 on page 107).




| CAT_ENV_SYSNAME | CAT_SYM_T4SIZE | SIZECOUNT |
|---|---|---|
| MV3A | 128 | 150 |
| | 192 | 61 |
| | 256 | 20782 |
| | unknown | 729 |

*Figure 5-27 IBM CAT database query on AES keys length distribution*

The 211 AES keys from the policy check as well as the 729 keys that are of unknown length may be subject of further investigation.

By querying further details from the IBM CAT database we can distribute the shorter AES key by for example, type (see Figure 5-28).

| CAT_ENV_SYSNAME | CAT_SYM_T4SIZE | CAT_SYM_TYPE | TYPECOUNT |
|---|---|---|---|
| MV3A | 128 | AESDATA | 31 |
| | | CIPHER | 38 |
| | | EXPORTER | 5 |
| | | IMPORTER | 35 |
| | | MAC | 16 |
| | | PINCALC | 1 |
| | | PINPROT | 24 |
| | 192 | AESDATA | 24 |
| | | CIPHER | 9 |
| | | IMPORTER | 19 |
| | | MAC | 3 |
| | | PINPROT | 6 |

*Figure 5-28 Shorter AES keys type distribution*

## 5.6.5 IBM CAT for cryptographic usage discovery

The examples that are used in this section show how cryptographic attributes can be discovered across systems in an enterprise using IBM CAT.

You can start the discovery process based on the output from any of the three IBM tools shown in Figure 5-29 on page 108.

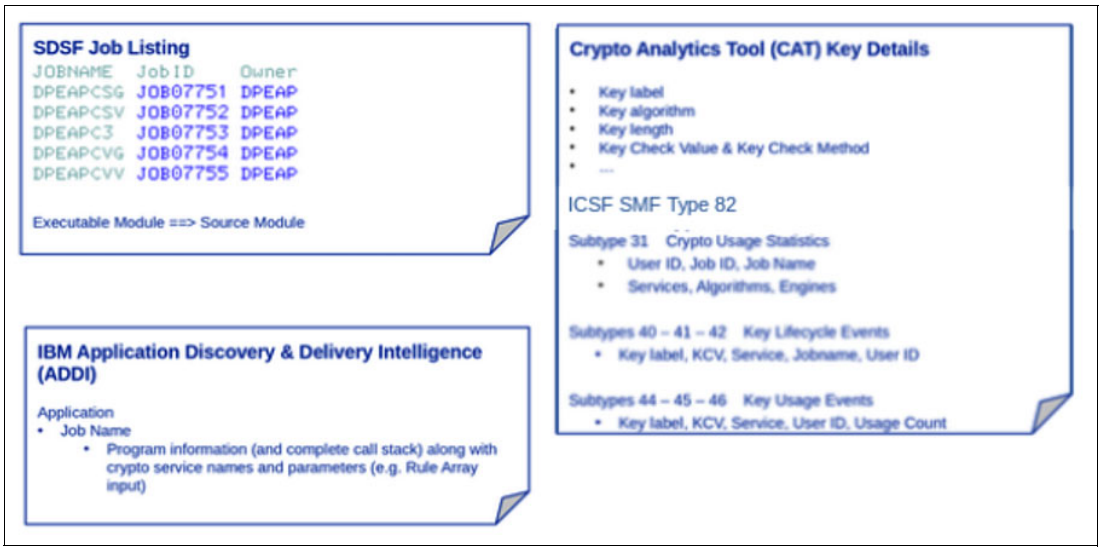



*Figure 5-29 Tools for cryptographic inventory*

## Starting with application source code scan from IBM ADDI

If you are starting with application source code that was scanned by IBM ADDI and uses cryptography, complete the following steps:

1. Select the application source code from the ADDI scan that you want to investigate.
2. Identify the ProgramName in the ADDI scan of the application source code.
3. Determine the job or started task that runs the executable.
4. Create reports from ICSF SMF record type 82:
   - Tailor CSFSMFJ in SYS1.SAMPLIB to fit your environment.
   - Run CSFSMFJ or use the IBM CAT SMF records from an IBM CAT snapshot.
5. Identify the ICSF services that are called by the job or started task and the related key material from the SMF records.
6. Review the key details from the IBM CAT snapshot.

## Starting with an application that you know

If you know the application that uses cryptographic functions, complete the following steps:

1. Select the application jobs or started tasks in the SDSF job listing.
2. Determine the application source code from the executable from the job.
3. Create reports from ICSF SMF record type 82:
   - Tailor CSFSMFJ in SYS1.SAMPLIB to fit your environment.
   - Run CSFSMFJ or use the IBM CAT SMF records from the IBM CAT snapshot.
4. Identify the ICSF service calls that relate to key material from the ICSF SMF records.
5. Verify the ADDI scanning of the application source code.
6. Review the key details from the IBM CAT snapshot.

## Starting with a policy check in the IBM CAT

If you ran a policy check in the IBM CAT component and now have a list of keys from your keystores that are deemed to be weak keys, complete the following steps:

1. Review the key details from an IBM CAT policy scan in the snapshot.
2. Create reports from ICSF SMF record type 82 that reference the key details by key label or by key check value (KCV):
   - Tailor CSFSMFJ in SYS1.SAMPLIB to fit your environment
   - Run CSFSMFJ or use the IBM CAT SMF records from the IBM CAT snapshot
3. Identify the ICSF service calls that relate to the key material from the SMF records.

4. Determine the job or started task from the ICSF SMF records.
5. Identify the executable from the job listing in SDSF and application source code.
6. Review the ADDI scanning of the application source code.

## Starting with SMF record type 82 reports

If SMF record type 82, subtype 31 indicate the use of weak cryptographic algorithms, either by the CSFSMFJ report or by the SMF records collected by IBM CAT, complete the following steps:

1. Identify the jobs or started tasks and users, ICSF service calls, and key material from the SMF 82 records.
2. Find the jobs or started tasks from the SMF records in the SDSF job listing.
3. Identify the executable modules from the SDSF job listing.
4. Determine the application source code from the executable.
5. Verify the ADDI scanning of the application source code.
6. Review the key details from the IBM CAT snapshot.

We will focus on the ICSF SMF record type 82 as captured by IBM CAT (see Figure 5-30 on page 110).

IBM CAT provides information about the cryptographic entities, including the key material, key data sets and the ICSF SMF record type 82 with these subtypes:

- Sub-type 31, which contains cryptographic statistics data for cryptographic engines (ENG), cryptographic services (SRV), and cryptographic algorithms (ALG) for a logical partition (LPAR)
- Sub-types 40 – 42, which contains ICSF key lifecycle events.
- Sub-types 44 – 46, which contains ICSF key usage events.

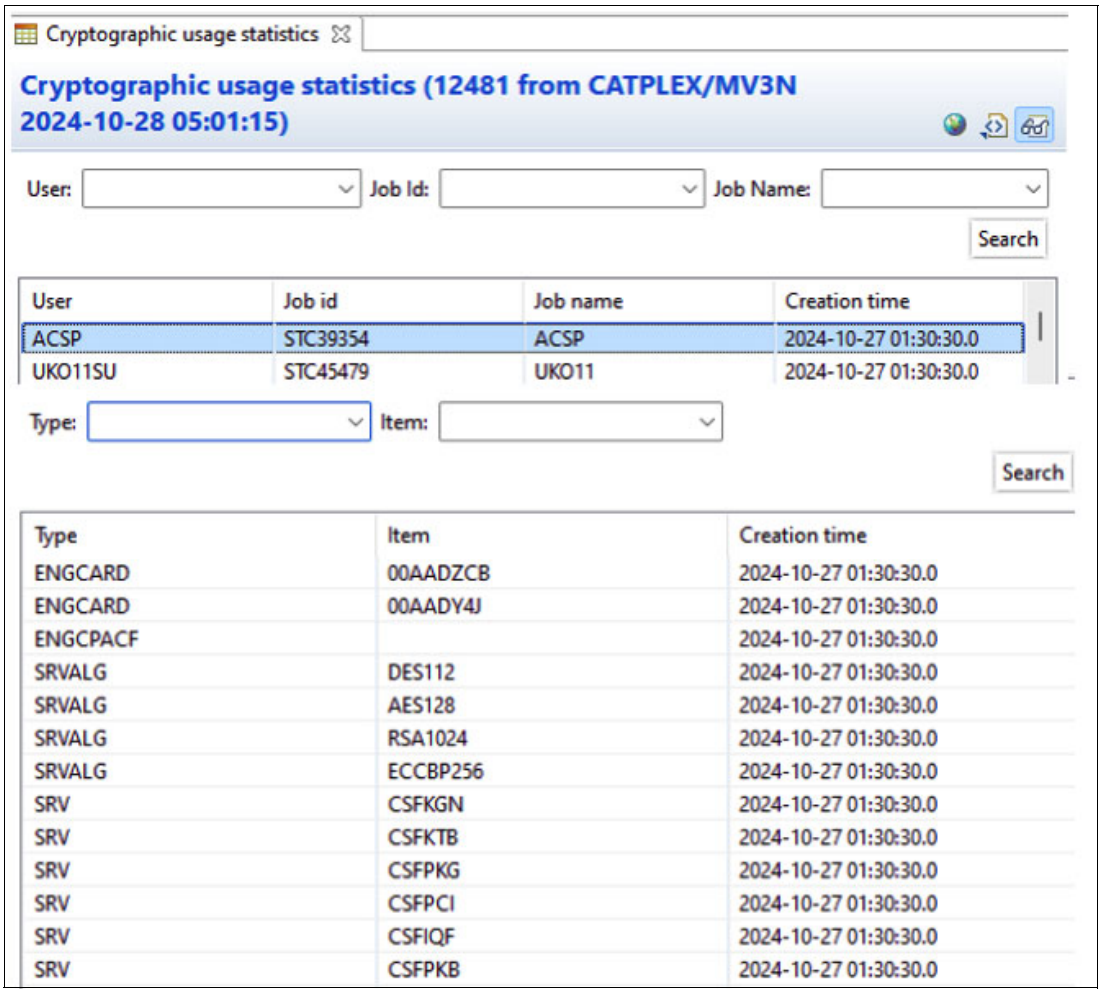



*Figure 5-30 ICSF SMF type 82 subtype 31 records in IBM CAT Monitor.*

Figure 5-30 shows the cryptographic usage statistics in IBM CAT Monitor. There are 12481 ICSF usage statistics records in one system alone. If counted across all systems we have close to 50,000 records.

That is a lot of data to examine and for a large enterprise with extensive cryptographic activity this number of records may very well be much higher.

Even though it is possible to conduct a cryptographic discovery with the IBM CAT Monitor, it is recommended to use Db2 queries and Jupyter Notebook to query and express cryptographic usage within the enterprise.

## Identify cryptographic algorithms across systems and jobs

As the cryptographic search in the database and the display of the results are rather lengthy, we have compiled a Jupyter Notebook. For a sample Jupyter Notebook for Crypto Discovery, go to the IBM Public Repository: *Quantum-Safe Redbook Samples, Chapter 5*

6

# Deploying quantum-safe capabilities

Now that your cryptographic inventory is created (as discussed in Chapter 5, “Creating a cryptographic inventory” on page 77), the preparation and planning can begin for the replacement of weak symmetric keys and public key algorithms. The algorithms and protocols at risk are discussed in “Cryptographic vulnerabilities possible with quantum computers” on page 8.

Adopting and implementing agreed upon quantum-safe standards, algorithms, and protocols in cryptographic systems helps protect against quantum computer and conventional computer attacks.

This chapter includes the following topics:



# 6.1 Quantum-safe algorithm artifacts

As you transition your cryptographic infrastructure to quantum-safe algorithms, it is important to understand some of the differences in artifact sizes between quantum-safe algorithms and traditional public key cryptography. It also is important to understand the supported quantum-safe algorithm object identifiers.

Quantum-safe algorithm (QSA) keys and signatures are much larger and might require changes in your environment. QSA, RSA, and Elliptic Curve Cryptography (ECC) keys are compared in Table 6-1. The private key token size includes the public key. The token size does not include other sections (such as private key name).

*Table 6-1 Asymmetric key token and signature sizes*

| Algorithm | Public key token size (bytes) | Private key token size (bytes) | Signature size (bytes) |
|---|---|---|---|
| RSA CRT 4096 | 1104 | 2504 | 512 |
| ECC Edwards 448 | 79 | 323 | 114 |
| ML-DSA-44 | 1312 | 3872 | 2420 |
| ML-DSA-65 | 1952 | 5984 | 3293 |
| ML-DSA-87 | 2592 | 7488 | 4595 |
| ML-KEM-768 | 1184 | 2400 | N/A |
| ML-KEM-1024 | 1568 | 3168 | N/A |

The supported quantum-safe algorithm object identifiers (OIDs) with Crypto Express7S and Crypto Express8S features are listed in Table 6-2 on page 113. The Module Lattice-based Digital Signature Algorithm (ML-DSA) has its usage domains broken up into two distinct sub-mechanisms: pre-hash and pure digital signatures. Each sub-mechanism has unique OIDs. When generating a pre-hashed digital signature, the user specifies the hash of the message they want to sign, which has been created before calling the signature generation mechanism. When generating a pure digital signature, the user must specify the entire message they want to sign, and the hash (SHAKE-256) is generated during the algorithm itself.

*Table 6-2 Supported quantum-safe algorithm object identifiers*

| Algorithm | Algorithm strength/version | Object identifier |
|---|---|---|
| CRYSTALS-Dilithium | CRYSTALS-Dilithium 6,5 Round 2[a] | 1.3.6.1.4.1.2.267.1.6.5 |
| | CRYSTALS-Dilithium 6,5 Round 3 | 1.3.6.1.4.1.2.267.7.6.5 |
| | CRYSTALS-Dilithium 8,7 Round 2 | 1.3.6.1.4.1.2.267.1.8.7 |
| | CRYSTALS-Dilithium 8,7 Round 3 | 1.3.6.1.4.1.2.267.7.8.7 |
| CRYSTALS-Kyber | CRYSTALS-Kyber 768 Round 2 | 1.3.6.1.4.1.2.267.5.3.3 |
| | CRYSTALS-Kyber 1024 Round 2 | 1.3.6.1.4.1.2.267.5.4.4 |
| Pure ML-DSA | ML-DSA-44 | 2.16.840.1.101.3.4.3.17 |
| | ML-DSA-65 | 2.16.840.1.101.3.4.3.18 |
| | ML-DSA-87 | 2.16.840.1.101.3.4.3.19 |
| Pre-Hash ML-DSA | ML-DSA-44 with SHA-512 | 2.16.840.1.101.3.4.3.32 |
| | ML-DSA-65 with SHA-512 | 2.16.840.1.101.3.4.3.33 |
| | ML-DSA-87 with SHA-512 | 2.16.840.1.101.3.4.3.34 |
| ML-KEM | ML-KEM-768 | 2.16.840.1.101.3.4.4.2 |
| | ML-KEM-1024 | 2.16.840.1.101.3.4.4.3 |

a. IBM z15 and Crypto Express7S features support CRYSTALS-Dilithium 6,5 Round 2 only

For more information about the supported quantum-safe OIDs, see *ICSF Application Programmer's Guide*, SC14-7508.

For more information about each quantum-safe algorithm, see the following NIST publications:

- FIPS 203 - Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)
- FIPS 204 - Module-Lattice-Based Digital Signature Standard (ML-DSA)

## 6.2 Converting your PKDS to KDSRL format

If you plan to store CCA QSA key tokens in your ICSF Public Key Data Set (PKDS), you must be on ICSF FMID HCR77D2 and have a large common record format (KDSRL) PKDS. KDSRL format supports all asymmetric key tokens and metadata. It also allows key usage tracking, if configured. KDSRL format increases the logical record length (LRECL) of the PKDS 3800 to 32756.

ICSF provides a utility to convert a KDSR format PKDS to KDSRL format by using the ICSF panes.

Complete the following steps to convert a PKDS to KDSRL format by using the ICSF panes:

1. On the ICSF Primary menu (see Example 6-1), select option **2, KDS MANAGEMENT** and then, press **Enter**.

*Example 6-1* ICSF Primary menu

```
HCR77D2  -------------- Integrated Cryptographic
 System Name:  SY1                            Crypto Domain: 0
 Enter the number of the desired option.

   1  COPROCESSOR MGMT -  Management of Cryptographic Coprocessors
   2  KDS MANAGEMENT   -  Master key set or change, KDS Processing
   3  OPSTAT           -  Installation options
   4  ADMINCNTL        -  Administrative Control Functions
   5  UTILITY          -  ICSF Utilities
   6  PPINIT           -  Pass Phrase Master Key/KDS Initialization
   7  TKE              -  TKE PKA Direct Key Load
   8  KGUP             -  Key Generator Utility processes
   9  UDX MGMT         -  Management of User Defined Extensions

      Licensed Materials - Property of IBM
      565O-ZOS Copyright IBM Corp. 1989, 2021.
      US Government Users Restricted Rights - Use, duplication or
      disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

 Press ENTER to go to the selected option.
 Press END   to exit to the previous menu.
 OPTION ===>
```

2. On the Key Data Set Management pane (see Example 6-2), select option **2, PKDS MANAGEMENT** and then, press **Enter**.

*Example 6-2 ICSF key data set management*

```
----------------------- ICSF - Key Data Set Management ------------

 Enter the number of the desired option.

   1  CKDS MANAGEMENT -  Perform Cryptographic Key Data Set (CKDS)
                         functions including master key management
   2  PKDS MANAGEMENT -  Perform Public Key Data Set (PKDS)
                         functions including master key management
   3  TKDS MANAGEMENT -  Perform PKCS #11 Token Data Set (TKDS)
                         functions including master key management
   4  SET MK          -  Set master keys

 Press ENTER to go to the selected option.
 Press END   to exit to the previous menu.
```

3. On the PKDS Management pane (see Example 6-3), select option **6, COORDINATED PKDS CONVERSION** and then, press **Enter**.

*Example 6-3 PKDS Management*

```
--------------------------- ICSF - PKDS Management -------------------------

 Enter the number of the desired option.

   1  PKDS OPERATIONS   -  Initialize a PKDS, activate a different PKDS,
                           (Refresh), or update the header of a PKDS and make
                           it active
   2  REENCIPHER PKDS   -  Reencipher the PKDS
   3  CHANGE ASYM MK    -  Change an asymmetric master key and activate the
                           reenciphered PKDS
   4  COORDINATED PKDS REFRESH - Perform a coordinated PKDS refresh
   5  COORDINATED PKDS CHANGE MK - Perform a coordinated PKDS change master key
   6  COORDINATED PKDS CONVERSION - Convert the PKDS to KDSR/L record format
   7  PKDS KEY CHECK    - Check key tokens in the active PKDS for format errors

 Press ENTER to go to the selected option.
 Press END   to exit to the previous menu.
```

4. On the Coordinated KDS conversion pane (see Example 6-4 on page 115), enter the new KDS name and then, press **Enter**.

*Example 6-4 Coordinated KDS conversion*

```
---------------------- ICSF - Coordinated KDS conversion -----------------

 To perform a coordinated KDS conversion, enter the KDS names below
 and optionally select the rename option.

     KDS Type ===> PKDS

   Active KDS ===> 'SYS1.PKDS.KDSR'

     New KDS ===> 'SYS1.PKDS.KDSRL'

           Rename Active to Archived and New to Active (Y/N) ===> N

           Archived KDS ===>

           Create a backup of the converted KDS (Y/N) ===> N

           Backup KDS ===>

 Press ENTER to perform a coordinated KDS conversion.
 Press END to exit to the previous menu.
```

After converting your PKDS to KDSRL format, you can confirm the change by using the **D ICSF,KDS** command. The output is shown in Example 6-5.

*Example 6-5 D ICSF, KDS output*

```
SY1           d icsf,kds
 SY1           CSFM668I 14.28.13 ICSF KDS 875
   CKDS  SYS1.CKDS.KDSR
     FORMAT=KDSR      COMM LVL=3  SYSPLEX=Y  MKVPs=DES AES
     DES MKVP date=Unknown
     AES MKVP date=Unknown
   PKDS  SYS1.PKDS.KDSRL
     FORMAT=KDSRL     COMM LVL=3  SYSPLEX=Y  MKVPs=RSA ECC
     RSA MKVP date=Unknown
     ECC MKVP date=Unknown
   TKDS  SYS1.TKDS.KDSRL
     FORMAT=KDSRL     COMM LVL=3  SYSPLEX=Y  MKVPs=P11
     P11 MKVP date=Unknown
```

## 6.3 Ensuring the environment is ready

The examples in the subsequent sections assume that all the necessary hardware and software is installed. Review the information in this section before deploying the quantum-safe capabilities.

The following general prerequisites must be met:

- ICSF FMID HCR77D2 or later is installed with the latest service level
- z/OS 2.5 or later is installed
- Running on IBM z16 or later
- Crypto Express8S or later is installed and configured
- CPACF feature code 3863 is enabled

For the CCA examples, our environment featured the following components:

- z/OS 2.5 installed and running on an IBM z16.
- ICSF FMID HCR77D2 with the latest service level applied.
- The CKDS and PKDS are allocated and in the correct format:
  - The CKDS is recommended to be in KDSRL format.
  - The PKDS must be in KDSRL format. For more information, see 6.2, “Converting your PKDS to KDSRL format” on page 113.
- Two Crypto Express8S (CEX8C) were installed and configured as CCA coprocessors (for more information, see *IBM z16 Configuration Setup*, SG24-8960).
- Feature Code (FC) 3863 was enabled for CPACF use.
- The AES master key was set and active in each CEX8C (CCA coprocessor) and initialized in the CKDS header. The Master Key Verification Pattern must be the same across your environment.

  If you are converting ciphertext that is encrypted with secure CCA DES keys, you must have the DES master key set and active in each CEX8C coprocessor and initialized in the CKDS header. The Master Key Verification Pattern *must* be the same across your environment.

CCA ML-DSA and ML-KEM key operations require that the ECC master key be set and active in each CEX8C coprocessor and initialized in the PKDS header. The Master Key Verification Pattern must be the same across your environment.

For the PKCS #11 examples, we made the following changes to our environment:

- The TKDS was allocated. A TKDS is not required if session objects are used.
- For secure key operations:
  - Two Crypto Express8 (CEX8P) were installed and configured as Enterprise PKCS #11 (EP11) coprocessor.
  - The EP11 master key was set and active in each CEX8P and initialized in the TKDS header. The Master Key Verification Pattern must be the same across your environment. A TKE is required to set the EP11 master key.

**Note:** PKCS #11 hybrid key-exchange can be done in hardware only.

For more information about allocating ICSF Key Data Sets and entering and activating ICSF master keys, see *z/OS ICSF Administrator's Guide, z/OS*, SC14-7506.

For more information about the IBM Z cryptographic stack, see 4.1, "IBM Z cryptographic components overview" on page 52.

## 6.4 Quantum-safe key generation

This section describes generating cover keys for AES 256, CRYSTALS-Dilithium, CRYSTALS-Kyber, ML-DSA, and ML-KEM algorithms by using ICSF services.

These algorithms are proven to be resistant to attacks from a powerful quantum computer. They also are ideal candidates when transitioning your application that is identified in your cryptographic inventory.

The PKCS #11 sample assumes that your environment is running with a specified ICSF TKDS. For more information about allocating and initializing a TKDS, see *ICSF System's Programmer's Guide*, SC14-7507.

### 6.4.1 Generating an AES 256-bit key by using ICSF CCA services

Generating a secure AES 256-bit CIPHER CCA key token can be done by using the Key Token Build2 (CSNBKTB2 and CSNEKTB2) and Key Generate2 (CSNBKGN2 and CSNEKGN2) ICSF services.

To generate a CCA AES 256-bit key, complete the following steps:

1. Build a skeleton token by using CSNBKTB2 with the correct key usage and key management bits specified. Ensure key type `CIPHER` is specified.
2. Pass the created skeleton token from CSNBKTB2 to CSNBKGN2 and specify the AES and OP rules.

The secure AES 256-bit CCA key token is generated and can be written to the CKDS with the CKDS Key Record Create2 (CSNBKRC2 and CSNEKRC2) service.

For more information about a sample REXX program that showcases steps 1 and 2, see B.1, "CCA AES 256-bit key generation REXX sample" on page 148.

For more information about these services and parameters, see *ICSF Application Programmer's Guide*, SC14-7508.

## 6.4.2 Generating an AES 256-bit key by using ICSF PKCS #11 services

Generating a secure AES 256-bit PKCS #11 key object can be done by using the PKCS #11 Generate Secret Key (CSFPGSK and CSFPGSK6) ICSF service.

To generate a secure PKCS #11 AES 256-bit key, complete the following steps:

1. Initialize a PKCS #11 token by using the PKCS #11 Token Record Create (CSFPTRC and CSFPTRC6) service.
2. Call the CSFPGSK service that passes the token handle that was created in step 1. Specify the CKA_CLASS (with a value of CKO_SECRET_KEY) and CKA_KEY_TYPE (with a value of CKK_AES) object attributes in the attribute list.

For more information about contains a sample REXX program that showcases step 2, see B.2, "PKCS #11 AES 256-bit key generation REXX sample" on page 151.

For more information about key object attributes, see *ICSF Writing PKCS #11 Applications guide*, SC14-7510.

## 6.4.3 Generating a ML-DSA key pair using ICSF CCA services

Generating a secure ML-DSA key pair can be done by using the PKA Key Token Build (CSNDPKB and CSNFPKB) and PKA Key Generate (CSNDPKG and CSNFPKG) ICSF services.

ML-DSA key pairs can be generated with one of two usage mechanisms: Pure and Pre-hash.

- Pure indicates that the message to be signed will be hashed with SHAKE-256 during the signing operation.
- Pre-hash indicates that the message to be signed has already been hashed.

When selecting a key usage, consider that you will not be able to switch once the key has been generated. Additionally, a pure ML-DSA key pair will not be usable for pre-hash signing operations and vice versa.

To generate a ML-DSA key pair, complete the following steps:

1. Build a skeleton token by using CSNDPKB passing the 'QSA-PAIR' and 'U-DIGSIG' rules. In the Key Value Structure (KVS), specify the algorithm ID and algorithm parameters for ML-DSA, as follows:
   - To generate a 'pure' ML-DSA key pair, specify '05'x as the algorithm ID in the KVS.
   - To generate a 'pre-hash' ML-DSA key pair, specify '07'x as the algorithm ID in the KVS.

   The following algorithm parameters are supported for ML-DSA:

   - X'0404' – ML-DSA-44
   - X'0605' – ML-DSA-65
   - X'0807' – ML-DSA-87.
2. Pass the created skeleton token from CSNDPKB to CSNDPKG and specify the 'MASTER' rule.

The secure ML-DSA key pair is generated and can be written to the PKDS with the PKDS Key Record Create (CSNDKRC and CSNFKRC) service if the PKDS is in KDSRL format. For more information, see 6.2, “Converting your PKDS to KDSRL format” on page 113.

The ML-DSA public key optionally can be extracted from the private key token by using the PKA Public Key Extract (CSNDPKX and CSNFPKX) service.

For more information about a sample REXX program that showcases steps 1 and 2, see B.3, “CCA ML-DSA key pair generation REXX sample” on page 153.

For more information about these services and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

### 6.4.4 Generating CRYSTALS-Dilithium key by using ICSF PKCS #11 services

Generating a secure PKCS #11 CRYSTALS-Dilithium key object can be done by using the PKCS #11 Generate Key Pair (CSFPGKP and CSFPGKP6) ICSF service.

To generate a secure PKCS #11 CRYSTALS-Dilithium key object, complete the following steps:

1. Initialize a PKCS #11 token by using the PKCS #11 Token Record Create (CSFPTRC and CSFPTRC6) service.
2. Call the CSFPGKP service that passes the token handle that was created from step 1. Specify the CKA_IBM_DILITHIUM_MODE object attribute with the DER encoded OID that corresponds to the CRYSTALS-Dilithium strength that is wanted in the public key attribute list.

For more information about a sample REXX program that showcases step 2, see B.4, “PKCS #11 CRYSTALS-Dilithium key pair generation REXX sample” on page 156.

For more information about key object attributes, see *ICSF Writing PKCS #11 Applications Guide*, SC14-7510.

### 6.4.5 Generating ML-KEM key pair by using ICSF CCA services

Generating a secure ML-KEM key pair can be done by using the PKA Key Token Build (CSNDPKB and CSNFPKB) and PKA Key Generate (CSNDPKG and CSNFPKG) ICSF services.

To generate a ML-KEM key pair, complete the following steps:

1. Build a skeleton token by using CSNDPKB passing the QSA-PAIR, U-KEYENC, and U-DATENC rules. In the Key Value Structure (KVS), specify the algorithm ID (X‘06’) and algorithm parameter for ML-KEM.

   The following algorithm parameters are supported for ML-KEM:

   – X’0768’ – ML-KEM-768
   – X’1024’ – ML-KEM-1024
2. Pass the created skeleton token from CSNDPKB to CSNDPKG and specify the MASTER rule.

The secure ML-KEM key pair is generated and can be written to the PKDS with the PKDS Key Record Create (CSNDKRC and CSNFKRC) service if the PKDS is in KDSRL format. For more information, see 6.2, “Converting your PKDS to KDSRL format” on page 113.

The ML-KEM public key optionally can be extracted from the private key token by using the PKA Public Key Extract (CSNDPKX and CSNFPKX) service.

For more information about a sample REXX program that showcases steps 1 and 2, see B.5, “CCA ML-KEM key pair generation REXX sample” on page 158.

For more information about these services and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

### 6.4.6 Generating CRYSTALS-Kyber key by using ICSF PKCS #11 services

Generating a secure PKCS #11 CRYSTALS-Kyber key object can be done by using the PKCS #11 Generate Key Pair (CSFPGKP and CSFPGKP6) ICSF service.

To generate a secure PKCS #11 CRYSTALS-Kyber key object, complete the following steps:

1. Initialize a PKCS #11 token by using the PKCS #11 Token Record Create (CSFPTRC and CSFPTRC6) service.
2. Call the CSFPGKP service that passes the token handle that was created from step 1. Specify the CKA_IBM_KYBER_MODE object attribute with the DER encoded OID corresponding to the CRYSTALS-KYBER strength that is wanted in the public key attribute list.

For more information about a sample REXX program that showcases step 2, see B.6, “PKCS #11 CRYSTALS-Kyber key pair generation REXX sample” on page 161.

For more information about key object attributes, see *ICSF Writing PKCS #11 Applications Guide*, SC14-7510.

## 6.5 Quantum-safe encryption

Data must be encrypted with AES 256-bit keys to ensure its protection from quantum computers running Grover’s algorithm. For more information about the effect Grover’s algorithm has on symmetric key cryptography, see 1.3, “Impact of Shor’s and Grover’s algorithms” on page 7.

The following options are available to protect ciphertext:

- Translate it by decrypting it by using the original key and then, encrypting it with an AES 256-bit key. The clear text is visible only for a short time within the secure cryptographic coprocessor.
- Reencrypt the ciphertext by using an AES 256-bit key without decrypting it first.
- Decrypt it with Symmetric Key Decipher (CSNBSYD or CSNBSYD1 and CSNESYD or CSNESYD1) and reencrypt it with Symmetric Key Encipher (CSNBSYE or CSNBSYE1 and CSNESYE or CSNESYE1). The clear text is visible only for a short time on the host system.

For more information about generating AES 256-bit keys, see 6.4.1, “Generating an AES 256-bit key by using ICSF CCA services” on page 117.

For more information about the use of AES 256-bit keys to translate cipher text from a weaker key (such as DES) to AES 256-bit encryption (option 1 in the previous bulleted list), see 6.4.2, “Generating an AES 256-bit key by using ICSF PKCS #11 services” on page 118.

### 6.5.1 Translating ciphertext to AES 256-bit encryption by using ICSF CCA services

Translating ciphertext from a weaker algorithm to AES 256-bit can be done by using the Cipher Text Translate2 (CSNBCTT2, CSNBCTT3, CSNECTT2, CSNECTT3) ICSF service.

To translate ciphertext to AES 256-bit encryption, complete the following steps:

1. Generate an AES 256-bit CIPHER key by using the ENCRYPT and C-XLATE key usage bits enabled. For more information, see B.1, “CCA AES 256-bit key generation REXX sample” on page 148.
2. Call the Cipher Text Translate2 service that passes the key that originally encrypted the ciphertext as the key_identifier_in and the new AES 256-bit key as the key_identifier_out.

The ciphertext is decrypted within the secure cryptographic coprocessor and reencrypted with the AES 256-bit key.

For more information about a sample REXX program that showcases step 2, see C.1, “CCA ciphertext translation REXX sample” on page 166.

For more information about these services and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

### 6.5.2 Translating ciphertext to AES 256-bit encryption by using ICSF PKCS #11 services

Translating ciphertext from a weaker algorithm to AES 256-bit can be done by using the PKCS #11 Secret Key Reencrypt (CSFPSKR and CSFPSKR6) ICSF service.

**Note:** This translate ciphertext service supports secure secret keys only.

To translate ciphertext over to AES 256-bit encryption, complete the following steps:

1. Generate an AES 256-bit PKCS #11 key object by using the CKA_IBM_SECURE key attribute set to TRUE. For more information, see B.2, “PKCS #11 AES 256-bit key generation REXX sample” on page 151.
2. Call the PKCS #11 Secret Key Reencrypt service that passes the key handle that originally encrypted the ciphertext as the decrypt_handle and the new AES 256-bit key handle as the encrypt_handle.

The ciphertext is decrypted within the secure cryptographic coprocessor and reencrypted with the AES 256-bit key.

For more information about a sample REXX program that showcases step 2, see C.2, “PKCS #11 ciphertext translation REXX sample” on page 168.

For for more information about these services and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

# 6.6 Quantum-safe digital signatures

Digital signatures are used to validate the authenticity and integrity of a message. Digital signatures also add nonrepudiation, which provides indisputable proof of origin for the signed data.

Traditional public key cryptography, such as RSA and ECC, are compromised by a sufficiently powerful quantum computer running Shor's algorithm. This can lead to data history manipulation with forged digital signatures. To protect against such issues, it is important to start adopting hybrid signature schemes that combine traditional public key cryptography and quantum-safe algorithms, such as ML-DSA.

ML-DSA can be used to sign data in one of two modes:

- "Pure" indicates that the message to be signed is hashed internally with the SHAKE-256 algorithm.
- "Pre-hash" indicates that the message to be signed has already been hashed. Only SHA-512 message digests are supported currently.

In some instances, signing large messages with ML-DSA "pure" may result in unacceptable performance. This can be addressed by signing a digest of the message (ML-DSA "pre-hash"). In general, ML-DSA "pure" is recommended.

For more information about how organizations can use digital signatures to verify the authenticity of data, see 3.4, "Proof of authorship" on page 46.

This subsequent subsection describes how to digitally sign and verify data by using the ML-DSA quantum-safe algorithm.

## 6.6.1 Generating and verifying a ML-DSA "Pure" digital signature by using ICSF CCA services

To generate and verify a ML-DSA digital signature, use the Digital Signature Generate (CSNDDSG and CSNFDSG) and Digital Signature Verify (CSNDDSV and CSNFDSV) ICSF services.

To generate a ML-DSA "Pure" signature, complete the following steps:

1. Generate a ML-DSA CCA key token by using the CSNDPKB and CSNDPKG ICSF services. For more information, see 6.4.3, "Generating a ML-DSA key pair using ICSF CCA services" on page 118.

   Ensure that the ML-DSA private key has the '05'x algorithm ID.
2. Call the CSNDDSG service and specify the CRDL-DSA, MESSAGE, and CRDLHASH rules. Pass the ML-DSA private key token that was generated in step 1.

   With a Crypto Express8 CCA coprocessor, the message to be signed can be up to 15000 bytes.

The generated signature is created. The signature size depends on the strength of the specified ML-DSA key.

To verify a ML-DSA "pure" signature, complete the following steps:

1. Call the PKA Public Key Extract (CSNDPKX and CSNFPKX) service to extract the ML-DSA public key from the private key token.

2. Call the CSNDDSV service and specify the CRDL-DSA, MESSAGE, and CRDLHASH rules. Pass the ML-DSA public key token that was extracted in step 1.

With a Crypto Express8 CCA coprocessor, the message to be verified can be up to 15000 bytes.

You receive a 0/0 return/reason code for a successful verification.

For more information about a sample REXX program that showcases a ML-DSA digital signature generation and verification, see D.1, “CCA ML-DSA “pure” digital signature generation and verification REXX sample” on page 172.

For more information regarding these services and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

## 6.6.2 Generating and verifying a ML-DSA "Pre-hash" digital signature by using ICSF CCA services

To generate and verify a ML-DSA digital signature, use the Digital Signature Generate (CSNDDSG and CSNFDSG) and Digital Signature Verify (CSNDDSV and CSNFDSV) ICSF services.

To generate a ML-DSA “pre-hash” signature, complete the following steps:

1. Generate a ML-DSA CCA key token by using the CSNDPKB and CSNDPKG ICSF services. For more information, see 6.4.3, “Generating a ML-DSA key pair using ICSF CCA services” on page 118.
   - Ensure that the ML-DSA private key has the ‘07’x algorithm ID.
2. Decide how the message to be signed will be hashed. In both cases, only the SHA-512 hashing algorithm is supported.
   - The message to be signed can be hashed separately with the One-Way Hash callable service (CSNBOWH). See 6.8.1, “Hashing a message with the SHA-512 algorithm by using ICSF CCA services” on page 134. If you choose this option, the SHA-512 message digest is passed to the CSNDDSG service.
   - The message to be signed can be hashed with the CSNDDSG service. If you choose this option, the entire message is passed to the CSNDDSG service.
3. Call the CSNDDSG service and specify the CRDL-DSA, ZERO-PAD, and SHA-512 rules. Pass the ML-DSA private key token that was generated in step 1.
   - If you chose to pass the SHA-512 message digest, additionally specify the HASH rule.
   - If you chose to pass the entire message, additionally specify the MESSAGE rule.

The generated signature is created. The signature size depends on the strength of the specified ML-DSA key.

To verify a ML-DSA “pre-hash” signature, complete the following steps:

1. Call the PKA Public Key Extract (CSNDPKX and CSNFPKX) service to extract the ML-DSA public key from the private key token.
2. Call the CSNDDSV service and specify the CRDL-DSA, ZERO-PAD, and SHA-512 rules. Pass the ML-DSA public key token that was extracted in step 1.
   - If you chose to pass the SHA-512 message digest, additionally specify the HASH rule.
   - If you chose to pass the entire message, additionally specify the MESSAGE rule.

You receive a 0/0 return/reason code for a successful verification.

For more information about a sample REXX program that showcases a ML-DSA “pre-hash” digital signature generation and verification, see D.2, “CCA ML-DSA “pre-hash” digital signature generation and verification REXX sample” on page 175.

For more information regarding these services and parameters, see ICSF Application Programmer’s Guide, SC14-7508.

### 6.6.3 Generating and verifying CRYSTALS-Dilithium digital signature by using ICSF PKCS #11 services

To generate and verify a CRYSTALS-Dilithium digital signature, use the PKCS #11 Private Key Sign (CSFPPKS and CSFPPKS6) and PKCS #11 Public Key Verify (CSFPPKV and CSFPPKV6) ICSF services.

To generate a CRYSTALS-Dilithium signature, complete the following steps:

1. Generate a CRYSTALS-Dilithium PKCS #11 key pair by using the CSFPGKP ICSF service. Ensure that the private key attribute list contains CKA_SIGN set to TRUE, and the public key attribute list contains CKA_VERIFY set to TRUE. For more information, see 6.4.4, “Generating CRYSTALS-Dilithium key by using ICSF PKCS #11 services” on page 119.
2. Call the CSFPPKS service and specify the LI2 rule. Pass the CRYSTALS-Dilithium private key handle that was generated in step 1.
3. The generated signature is created. The signature size depends on the strength of the specified CRYSTALS-Dilithium key.

To verify a CRYSTALS-Dilithium signature, perform the following steps call the CSFPPKV service and specify the LI2 rule. Pass the CRYSTALS-Dilithium public key handle. Pass the signature to verify and the original message.

A 0/0 return/reason code is returned for a successful verification.

D.3, “PKCS #11 CRYSTALS-Dilithium digital signature generation and verification REXX sample” on page 178 contains a sample REXX program that showcases a CRYSTALS-Dilithium digital signature generation and verification.

For more information about these services and parameters, see *ICSF Application Programmer’s Guide,* SC14-7508.

### 6.6.4 Using digital signatures to protect SMF records

SMF records are an important part of the auditability to the z/OS platform. For example, SMF records can contain important security-related information, such as RACF processing records (SMF record type 80).

It is important to ensure that these records are never tempered with. SMF digital signature provides a cryptographic means to verify the integrity of the records when log streams are used as the recording media.

In this section, we demonstrate how SMF records digital signature is used to sign SMF records and how to transition to quantum-safe algorithms for that purpose.

## Configuring SMF digital signature

An SMF digital signature is based on a token and a key (RSA or ECC). The digital signature can be the same for all the records or specific to a log stream.

### *Step 1: Creating the token and the associated key pair*

We created a PKCS#11 token that is called `QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1`. Then, we created a key pair that we bound to this certificate (see Example 6-6).

*Example 6-6 Creating and binding a PKCS#11 token and a key pair (RSA)*

```
RACDCERT ADDTOKEN(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1)
RACDCERT GENCERT ID(STCID) SUBJECTSDN(CN('SMF sign cert')) +
WITHLABEL('SMF sign certificate') SIZE(2048) RSA +
NOTAFTER(DATE(2023/04/18))
RACDCERT BIND(ID(STCID) LABEL('SMF sign certificate') DEFAULT +
TOKEN(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1)
```

### *Step 2: Configuring SMF for digital signature*

Example 6-7 shows an SMFPRMxx parmlib member (truncated), which enables SMF digital signature for a log stream IFASMF.ALLSYS.DATA that stores selected z/OS SMF records. Other SMF records that are directed to the default log stream IFASMF.ALLSYS.DEFAULT do not have SMF digital signature processing enabled.

*Example 6-7 SMFPRMxx parmlib member enabling SMF digital signature for SMF log streams*

```
ACTIVE                      /*ACTIVE SMF RECORDING*/
LISTDSN                     /* LIST DATA SET STATUS AT IPL*/
NOPROMPT                    /*DON'T PROMPT THE OPERATOR  */
DEFAULTLSNAME(IFASMF.ALLSYS.DEFAULT,COMPRESS(PERMFIX(32M)))
LSNAME(IFASMF.ALLSYS.DATA,TYPE(0,23,30,42,70:79,80:83,99,113),
     COMPRESS(PERMFIX(64M))
     RECSIGN(HASH(SHA512),SIGNATURE(RSA),
     TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1))
)
RECORDING(LOGSTREAM)
…
```

Note that the log stream IFASMF.ALLSYS.DATA must be defined with a MAXBUFSIZE of 65532.

### *Step 3: Extracting the data from the log stream*

When extracting the SMF records from the log streams, we want to ensure that the SMF digital signature is available for post-processing. For this purpose, the NOSIGSTRIP parameter *must* be specified (see Example 6-8).

*Example 6-8 JCL for SMF records extraction, preserving SMF digital signature*

```
//IFASMFDL EXEC PGM=IFASMFDL,REGION=0M
//OUTDD1   DD DSN=RBOOK.SMF.LOGS,DISP=(NEW,CATLG,DELETE),
//         SPACE=(CYL,(100,100),RLSE),
//         DCB=(LRECL=32760,RECFM=VBS,BLKSIZE=0)
//SYSPRINT DD  SYSOUT=A
//SYSIN    DD  *
 LSNAME(IFASMF.ALLSYS.DATA,OPTIONS(DUMP))
 OUTDD(OUTDD1,TYPE(0:255),START(1400),END(2000))
 NOSIGSTRIP
```

```
DATE(2022096,2022096)
```

### *Step 4: Validating the SMF signature*

By using the IFASMFDP utility, we can specify the SIGVALIDATE parameter along with the hashing method and the name of the token. The utility validates the records and ensures that they are all signed with the specified token (see Example 6-9).

**Tip:** You do not have write out records to another data set when performing validation, as we show in the example. An alternative is to use a dummy data set, by replacing the DDSMF1 DD statements with DDSMF1 DD DUMMY.

*Example 6-9 JCL to create SMF records signing validation*

```
//SMF      EXEC     PGM=IFASMFDP
//DUMPIN   DD  DISP=SHR,DSN=RBOOK.SMF.LOGS
//DDSMF1  DD  DSN=RBOOK.SMF.RACF,
//             DISP=(NEW,CATLG,DELETE),
//           SPACE=(CYL,(100,100),RLSE),
//           DCB=(RECFM=VBS,BLKSIZE=32748,LRECL=32756)
//SYSPRINT DD SYSOUT=A
//SYSIN    DD  *
 SID(SYSA)
 INDD(DUMPIN,OPTIONS(DUMP))
 OUTDD(DDSMF1,TYPE(30,82))
 NOSIGSTRIP
 DATE(2022096,2022096)
 START(1400) END(1800)
 SIGVALIDATE(HASH(SHA512),
     TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1))
```

Sample output from the SMF records signing validation job is shown in Example 6-10.

*Example 6-10 SMF records signing validation report*

```
IFA010I SMF DUMP PARAMETERS
IFA010I REPORTOPTS(NOSUBTYPE) -- DEFAULT
IFA010I NOASIGVALIDATE -- DEFAULT
IFA010I SIGVALIDATE(HASH(SHA512),
           TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1)) -- SYSIN

IFA010I END(1800) -- SYSIN
IFA010I START(1400) -- SYSIN
IFA010I DATE(2022096,2022096) -- SYSIN
IFA010I NOSIGSTRIP -- SYSIN
IFA010I OUTDD(DDSMF1,TYPE(30,82)) -- SYSIN
IFA010I INDD(DUMPIN,OPTIONS(DUMP)) -- SYSIN
IFA010I SID(SYSA) -- SYSIN
IFA020I DDSMF1   -- RBOOK.SMF.RACF
IFA020I DUMPIN   -- RBOOK.SMF.LOGS

                                    SUMMARY ACTIVITY REPORT
     START DATE-TIME  04/06/2022-14:00:00                    END DATE-TIME  04/06/2022-19:59:58
     RECORD       RECORDS         PERCENT      AVG. RECORD   MIN. RECORD   MAX. RECORD       RECORDS
       TYPE          READ        OF TOTAL          LENGTH        LENGTH        LENGTH       WRITTEN
          2         5,974          4.35 %          347.85            18           356           447
```

```
     3            1          .00 %          18.00          18          18           1
    23           24          .02 %       8,062.00       8,062       8,062           0
    30       20,995        15.29 %       1,529.65         480       6,121       3,244
    42        5,674         4.13 %       2,473.64         176      32,444           0
    70        1,224          .89 %      20,887.45       1,672      32,600           0
    71          288          .21 %       2,596.00       2,596       2,596           0
    72       13,824        10.06 %       1,791.39       1,324      13,328           0
    73          288          .21 %      29,216.00      29,216      29,216           0
    74        5,616         4.09 %      10,474.92         308      32,520           0
    75          576          .42 %         272.00         272         272           0
    77          288          .21 %         585.25         328         832           0
    78          576          .42 %       6,428.00       4,056       9,008           0
    82           47          .03 %         120.00         120         120          31
    99       75,619        55.06 %       2,877.35         419       8,870           0
   113        6,336         4.61 %       1,606.00       1,462       1,834           0
TOTAL       137,350          100 %       2,901.58          18      32,600       3,723
NUMBER OF RECORDS IN ERROR              0

                          RECORD VALIDATION REPORT FOR SYSA
RECORD   RECORD   VALIDATION      VALIDATION START       VALIDATION END       RECORDS       GROUPS     INTERVALS
  TYPE  SUBTYPE      FAILURE            DATE-TIME            DATE-TIME     VALIDATED     VALIDATED     VALIDATED
    30        2            N   04/06/2022-14:00:00  04/06/2022-18:00:00         2,620            73            48
    30        6            N   04/06/2022-14:00:00  04/06/2022-18:00:00           624            48            48
    30        3            N   04/06/2022-14:00:00  04/06/2022-18:00:00             0             0            48
    30        4            N   04/06/2022-14:00:00  04/06/2022-18:00:00             0             0            48
    30        5            N   04/06/2022-14:00:00  04/06/2022-18:00:00             0             0            48
    82       20            N   04/06/2022-14:00:00  04/06/2022-18:00:00            31            31            48
VALIDATION SUCCEEDED
```

## Implementing SMF alternative signatures

As described in “Impact of Shor’s and Grover’s algorithms” on page 7, traditional public key cryptography, such as RSA and ECC, can be compromised by a quantum computer that is running Shor’s algorithm. To maintain a safe way to validate SMF audit data, you can use a function that was introduced with z/OS 2.4 that allows a secondary (or alternative) signature that uses CRYSTALS-Dilithium.

### *Step 1: Generate the PKCS #11 CRYSTALS-Dilithium key pair*

The PKCS #11 CRYSTALS-Dilithium key pair can be clear or secure. For a secure key pair, an Enterprise PKCS #11 coprocessor Crypto Express7S or later must be available with the suitable ICSF minimum service level. The minimum hardware and software levels are listed in Table 4-2 on page 59.

For more information about generating PKCS #11 CRYSTALS-Dilithium key pair services, see 6.4.4, “Generating CRYSTALS-Dilithium key by using ICSF PKCS #11 services” on page 119.

### *Step 2: Configuring SMF for alternative digital signature*

Example 6-11 shows an SMFPRMxx parmlib member (truncated) that enables an SMF digital signature, and alternative digital signature for a log stream that stores z/OS events. The alternative signature is specified in SMFPRMxx by using the ARECSIGN parameter and the RECSIGN parameter. When ARECSIGN is used, RECSIGN also must be specified. The specified TOKENNAME must be the PKCS #11 token that contains the CRYSTALS-Dilithium key object.

*Example 6-11 SMFPRMxx parmlib member that enables SMF digital signature for SMF log streams*

```
ACTIVE                      /*ACTIVE SMF RECORDING*/
LISTDSN                     /* LIST DATA SET STATUS AT IPL*/
NOPROMPT                    /*DON'T PROMPT THE OPERATOR  */
```

```
DEFAULTLSNAME(IFASMF.ALLSYS.DEFAULT,COMPRESS(PERMFIX(32M)))
LSNAME(IFASMF.ALLSYS.DATA,TYPE(0,23,30,42,70:79,80:83,99,113),
     COMPRESS(PERMFIX(64M))
     RECSIGN(HASH(SHA512),SIGNATURE(RSA),
     TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1))
     ARECSIGN(HASH(SHA512),SIGNATURE(LI2),
     TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS2))
)
RECORDING(LOGSTREAM)
```

### *Step 3: Extracting the data from the log stream*

Although the extraction process did not change, we still need to use the NOSIGSTRIP parameter to preserve the record signatures (see Example 6-12).

*Example 6-12 JCL for SMF records extraction, preserving SMF digital signature*

```
//IFASMFDL EXEC PGM=IFASMFDL,REGION=0M
//OUTDD1   DD DSN=RBOOK.SMF.LOGS,DISP=(NEW,CATLG,DELETE),
//         SPACE=(CYL,(100,100),RLSE),
//         DCB=(LRECL=32760,RECFM=VBS,BLKSIZE=0)
//SYSPRINT DD  SYSOUT=A
//SYSIN    DD  *
 LSNAME(IFASMF.ALLSYS.DATA,OPTIONS(DUMP))
 OUTDD(OUTDD1,TYPE(0:255),START(1400),END(2000))
 NOSIGSTRIP
 DATE(2022096,2022096)
```

### *Step 4: Validating the SMF signature*

Here, we want to specify the SIGVALIDATE *and* ASIGVALIDATE parameters (see Example 6-13).

*Example 6-13 JCL to create SMF records signing validation including alternative signature*

```
//SMF      EXEC     PGM=IFASMFDP
//DUMPIN   DD  DISP=SHR,DSN=RBOOK.SMF.LOGS
//DDSMF1  DD  DSN=RBOOK.SMF.RACF,
//             DISP=(NEW,CATLG,DELETE),
//           SPACE=(CYL,(100,100),RLSE),
//           DCB=(RECFM=VBS,BLKSIZE=32748,LRECL=32756)
//SYSPRINT DD SYSOUT=A
//SYSIN    DD  *
 SID(SYSA)
 INDD(DUMPIN,OPTIONS(DUMP))
 OUTDD(DDSMF1,TYPE(30,82))
 NOSIGSTRIP
 DATE(2022096,2022096)
 START(1400) END(1800)
 SIGVALIDATE(HASH(SHA512),
     TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS1))
 ASIGVALIDATE(HASH(SHA512),
     TOKENNAME(QSAFE.REDBOOK.SMF.SIGN.TOKEN.LS2))
```

# 6.7 Quantum-safe key exchange

Key exchange allows two parties to establish a shared secret by using public key cryptography.

With the introduction of the ML-KEM algorithm, it is possible to perform a quantum-safe hybrid key exchange scheme that combines the protection of traditional Elliptic Curve Cryptography (ECC) and the quantum-safe ML-KEM algorithm. This hybrid key exchange scheme provides two layers of protection and ensures that all key exchanges are protected from attacks by traditional and quantum computers.

Additionally, the ML-KEM algorithm can be used to encapsulate and decapsulate a randomly generated 32-byte secret which can be used to establish a new AES 256-bit key.

For more information about how organizations can use secure key exchange to protect their sensitive data, see 3.2, “Sharing keys securely” on page 34.

This section describes how to perform a quantum-safe key exchange by using CCA and PKCS #11 services.

## 6.7.1 Performing a hybrid quantum-safe key exchange scheme by using ICSF CCA services

A hybrid quantum-safe key exchange can be performed with the PKA Encrypt (CSNDPKE and CSNFPKE) and EC Diffie-Hellman (CSNDEDH and CSNFEDH) ICSF services.

The following Access Control Points (ACP) must be enabled:

- PKA Encrypt - Allow ML-KEM, CRYSTALS-Kyber keys
- EC Diffie-Hellman -Allow Hybrid QSA Scheme (035Dx)

Consider the following hybrid quantum-safe key exchange scheme that includes two participants: Alice and Bob are two parties who want securely exchange information. They can be a company and a Business Partner, for example.

### Step 1: Alice

1. Alice creates the following keys:
   - ML-KEM-priv-A, ML-KEM-pub-A: ML-KEM-1024 key pair
   - EC-priv-A, EC-pub-A: ECC key pair for key agreement
   - ML-KEM-cert-A, EC-cert-A: authentication forms of ML-KEM-pub-A and EC-pub-A
2. Alice sends ML-KEM-cert-A and EC-cert-A to Bob.

### Step 2: Bob

1. Bob receives and validates ML-KEM-cert-A and EC-cert-A
2. Bob creates the following keys:
   - AES-ciph-B: AES CIPHER key in a CCA key token

   **Note:** AES-ciph-B must be as strong as the derived shared key (for example, AES 256-bit) and allow encrypt *and* decrypt operations.

   - EC-priv-B, EC-pub-B: ECC key pair for key agreement

- EC-cert-B: authenticated form of EC-pub-B
- ML-KEM-pub-A CCA public key token with public key pulled from ML-KEM-cert-A

3. Bob creates the shared key derivation input by using the CSNDPKE service:
   - RANDOM keyword, AES-ciph-B, ML-KEM-pub-A, AES encryption IV
   - Generates a random 32B value: rand-32
   - AES-CBC encrypts rand-32 by using key AES-ciph-B and the AES encryption IV returning [AES-ciph-B(rand-32)] in the keyvalue parameter.
   - ML-KEM encrypts rand-32 with ML-KEM-pub-A returning [ML-KEM-pub-A(rand-32)] in the PKA_enciphered_keyvalue parameter.
4. Bob completes the shared key derivation by using CSNDEDH

   Bob calls CSNDEDH by using a derivation keyword and wanted key length, [AES-ciph-B(rand-32)], AES-ciph-B, AES encryption IV, EC-priv-B, EC-cert-A, output skeleton token.

   Consider the following points about CSNDEDH:
   - Decrypts rand-32 by using the key AES-ciph-B and the AES encryption IV.
   - Uses EC-priv-B and EC-cert-A with ECDH to generate the Z value.
   - Passes Z and rand-32 to the key derivation function that is indicated by the derivation keyword, rand-32 is the salt or OtherData. The shared key of the requested length is derived.
   - Places the shared key in the provided output skeleton token and then, encrypts the key value.
   - Returns the final CCA shared key token.
5. Bob stores the shared key
6. Bob sends EC-cert-B, [ML-KEM-pub-A(rand-32)] to Alice.

### Step 3: Alice

1. Alice receives and validates EC-cert-B, [ML-KEM-pub-A(rand-32)].
2. Alice completes the shared key derivation by using CSNDEDH

   Alice calls CSNDEDH with a derivation keyword and the wanted key length, [ML-KEM-pub-A (rand-32)], ML-KEM-priv-A, EC-priv-A, EC-cert-B, output skeleton token.

   Consider the following points about CSNDEDH:
   - Decrypts rand-32 by using ML-KEM-priv-A.
   - Uses EC-priv-A and EC-cert-B with ECDH to generate the Z value.
   - Passes Z and rand-32 to the key derivation function that is indicated by the derivation keyword, rand-32 is the salt or OtherData. The shared key of the requested length is derived.
   - Places the shared key in the provided output skeleton token and then, encrypts the key value.
   - Returns the final CCA shared key token.
3. Alice stores the shared key.

The shared key is now established by Alice and Bob.

### Role of CSFNDPKE

The role of the PKA Encrypt (CSNDPKE) service in this scheme is to create the rand-32 derivation input and return rand-32 in the following forms:

- Encrypted by Bob's AES cipher key, AES-ciph-B
- Encrypted by Alice's ML-KEM public key, ML-KEM-pub-A

This process is accomplished in one call to CSNDPKE:

- Inputs:
  - RANDOM rule-array keyword
  - AES-ciph-B: AES-cipher key for Bob

    **Note:** AES-ciph-B must be as strong as the derived shared key.

  - Kyber-pub-A as PKA_key_identifier: ML-KEM key for Alice
- Outputs:
  - keyvalue parameter: [AES-ciph-B(rand-32)]
  - PKA_enciphered_keyvalue parameter: [ML-KEM-pub-A(rand-32)]

Consider the following points:

- Authentication of the public keys that are used in the scheme is the responsibility of the host.

  Currently, ML-KEM keys do not participate in PKI processes. The ML-KEM-cert-A certificate for a ML-KEM public key recognizes that certificate formats are needed for the authentication part of a protocol.

  For the ECC public keys, the CCA internal PKI can be used for authentication if the trust anchor was installed in the adapter.
- A full protocol must include a Key Check Value that is calculated over the shared key that was created by Bob so that Alice can verify the creation of an agreed shared key.

For more information about a REXX sample that shows this end-to-end scheme that uses ICSF CCA services, see E.1, "CCA hybrid quantum-safe key exchange scheme REXX sample" on page 184.

For more information about these services and parameters, see *ICSF Application Programmer's Guide*, SC14-7508.

## 6.7.2 Performing a quantum-safe key encapsulation with ML-KEM using ICSF CCA services

To encapsulate and decapsulate a randomly generated secret using ML-KEM, use the PKA Encrypt (CSNDPKE and CSNFPKE) and PKA Decrypt (CSNDPKD and CSNFPKD) ICSF callable services.

To encapsulate a randomly generated secret, complete the following steps:

1. Generate a ML-KEM CCA key token by using the CSNDPKB and CSNDPKG ICSF services. For more information, see 6.4.3, "Generating a ML-DSA key pair using ICSF CCA services" on page 118.
2. Determine which format you want the secret to be returned. It can be returned in one of three ways:

- In the clear (unencrypted) using rule ‘CLEAR’.
- Encrypted by an AES wrapping key using rule ‘AES-ENC’.
- In an Encrypted CCA AES key token or TR-31 key block using rule ‘AES-KB’.

3. Call the CSNDPKE service passing the ‘ZERO-PAD’ and ‘RANDOM’ rules. Additionally pass the key format rule based on your selection in step 2. Pass the ML-KEM public key in the *PKA_key_identifier* parameter.
   - If you selected ‘AES-ENC’ in step 2, the *sym_key_identifier* parameter should contain a CCA AES CIPHER key or AES TR-31 key block.
   - If you selected ‘AES-KB’ in step 2, the *sym_key_identifier* parameter should contain an AES CCA or TR-31 skeleton.
4. If the call is successful, the ML-KEM wrapped secret will be returned in the *PKA_enciphered_keyvalue* field. Additionally, the secret will be returned in the format selected from step 2 in the *keyvalue* field.

To decapsulate a secret, complete the following steps:

1. Determine which format you want the secret to be returned. It can be returned in one of two ways:
   - In the clear (unencrypted) using rule ‘CLEAR’.
   - In an Encrypted CCA AES key token or TR-31 key block using rule ‘AES-KB’.
2. Call the CSNDPKD service passing the ‘ZERO-PAD’ rule. Additionally pass the key format rule based on your selection in step 2. Pass the ML-KEM private key in the *key_identifier* parameter. The *PKA_enciphered_keyvalue* parameter will contain the ML-KEM enciphered secret produced during the encapsulate step.
   - If you selected ‘AES-KB’ in step 2, the *sym_key_identifier* parameter should contain an AES CCA or TR-31 skeleton.
3. If the call is successful, the secret will be returned in the *target_keyvalue* field according to the format selected in step 1.

For more information about a sample REXX program that showcases a ML-KEM encapsulationand decapsulation, see E.2, “CCA ML-KEM key encapsulation REXX sample” on page 194.

For more information regarding these services and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

## 6.7.3 Performing a hybrid quantum-safe key exchange scheme by using ICSF PKCS #11 services

A hybrid quantum-safe key exchange can be performed by using the PKCS #11 Derive Key (CSFPDVK and CSFPDVK6) ICSF service.

Consider the following hybrid quantum-safe key exchange scheme that includes two participants: Alice and Bob, who are two parties that want to securely exchange information. They can be a company and a Business Partner, for example.

### Step 1: Alice

1. Alice generates an ECC key pair (EC-pub-A, EC-priv-A) for key agreement by using the PKCS #11 Generate Key Pair service.
2. Alice creates EC-cert-A: authenticated form of EC-pub-A
3. Alice sends EC-cert-A to Bob.

### Step 2: Bob

1. Bob receives and validates EC-cert-A.
2. Bob creates the following keys:
   - ECC (EC-pub-B, EC-priv-B) key pair for key agreement by using the PKCS #11 Generate Key Pair service.
   - CRYSTALS-Kyber (Kyb-pub-B, Kyb-priv-B) key pair by using the PKCS #11 Generate Key Pair service.
   - EC-pub-A PKCS #11 public key object that was pulled from EC-cert-A by using the PKCS #11 Token Record Create service.
   - Kyb-cert-B and EC-cert-B: authenticated forms of Kyb-pub-B and EC-pub-B.
3. Bob derives a generic secret key object (GenSec-B) by passing EC-priv-B *and* EC-pub-A to PKCS #11 Derive Key.
4. Bob sends Kyb-cert-B and EC-cert-B to Alice.

### Step 3: Alice

1. Alice receives and validates Kyb-cert-B and EC-cert-B.
2. Alice creates Kyb-pub-B and EC-pub-B PKCS #11 public key objects that were pulled from their respective certificates by using the PKCS #11 Token Record Create service.
3. Alice derives a generic secret key object (GenSec-A) by passing EC-priv-A *and* EC-pub-B to PKCS #11 Derive Key.
4. Alice passes GenSec-A and Kyb-pub-B to PKCS #11 Derive Key to encapsulate random key material (rand-A). The Kyber-encapsulated random key material [Kyb-pub-B(rand-A)] and derived shared key are returned. The shared key is the output in the target_key_handle parameter.
5. Alice sends to Bob the Kyber-encapsulated random key material [Kyb-pub-B(rand-A)].

### Step 4: Bob

Bob passes GenSec-B and Kyb-priv-A to PKCS #11 Derive Key to decapsulate [Kyb-pub-B(rand-A)]. The derived shared key is output in the target_key_handle parameter.

The shared key is now established at Alice and Bob.

Authentication of the public keys that are used in the scheme is the responsibility of the host.

Currently, CRYSTALS-Kyber keys do not participate in PKI processes. The Kyb-cert-B certificate for the CRYSTALS-Kyber public key recognizes that certificate formats are needed for the authentication part of a protocol.

The lack of PKI support for CRYSTALS-Kyber can be circumvented by using a trustworthy public RSA or EC certificate to verify a signed Kyber SPKI.

For more information about a REXX sample that shows this end-to-end scheme that uses ICSF PKCS #11 services, see E.3, “PKCS #11 hybrid quantum-safe key exchange scheme REXX sample” on page 197.

For more information about these services and parameters, see the *ICSF Application Programmer’s Guide*, SC14-7508.

# 6.8 Quantum-safe hashing

*Hashing* is the process of transforming data (for example a key or message) into a shorter, fixed-length message digest by using a cryptographic hash algorithm.

Hashes are used in various cryptographic operations, such as digital signatures, and key derivation functions, such as PBKDF2, Message Authentication Codes (MAC). The message digests that are produced from a hash algorithm ensures the integrity of the data and protects against unauthorized alteration of the source data.

It is important to start transitioning from weaker hash algorithms, such as SHA-1 or MD2, to much stronger hash algorithms, such as SHA-256 or SHA-512.

This section describes how to perform an SHA-512 hash over a message by using CCA and PKCS #11 ICSF services.

## 6.8.1 Hashing a message with the SHA-512 algorithm by using ICSF CCA services

To hash a message by using the SHA-512 algorithm, use the One-Way Hash Generate (CSNBOWH or CSNBOWH1 and CSNEOWH or CSNEOWH1) ICSF service.

To hash a message by using the SHA-512 algorithm, complete the following steps:

1. Call the CSFBOWH ICSF service that passes the SHA-512 and ONLY rules.
2. Pass the message to hashed in the text parameter. (Optionally, the message can be hashed in parts by using the chaining flag rules and the chaining_vector parameter.)

The 64-byte message digest is output in the hash parameter.

For more information about a sample REXX program that showcases this process, see F.1, “CCA SHA-512 one-way hash REXX sample” on page 208.

For more information about this service and parameters, see *ICSF Application Programmer’s Guide*, SC14-7508.

## 6.8.2 Hashing a message with the SHA-512 algorithm by using ICSF PKCS #11 services

To hash a message by using the SHA-512 algorithm, use the PKCS #11 One-Way Hash, Sign, or Verify (CSFPOWH and CSFPOWH6) ICSF service.

To hash a message by using the SHA-512 algorithm, complete the following steps:

1. Initialize a PKCS #11 token by using the PKCS #11 Token Record Create (CSFPTRC and CSFPTRC6) service.
2. Call the CSFPOWH ICSF service that passes the SHA-512 and ONLY rules.
3. Pass the message to hashed in the text parameter.
4. Pass the token handle that was created in step one in handle parameter. (Optionally, the message can be hashed in parts by using the chaining flag rules and the chaining_vector parameter.)

The 64-byte message digest is output in the hash parameter.

For more information about a sample REXX program that showcases step 2, see F.2, “PKCS #11 SHA-512 one-way hash REXX sample” on page 209.

For more information about these services and parameters, see *ICSF Application Programmer’s Guide,* SC14-7508.

## 6.9 Validating your quantum-safe transition

As discussed in 5.1, “Collection tools overview” on page 78, the usage statistics are key to a comprehensive cryptographic inventory.

In Example 6-14, we can see that non-quantum-safe algorithms are used in our DATAOWN job (which is a formatted report of SMF record type 82, subtype 31 [hex ’001F’]). We can identify the use of single DES, AES 128, weak RSA 1024, and so on.

*Example 6-14 Crypto usage statistics:Checking for non-quantum-safe algorithms*

```
Type=82 Subtype=001F Crypto Usage Statistics
Written periodically to record crypto usage counts
22 Feb 2022 15:12:27.73
   TME... 005389D5 DTE... 0122053F SID... SP21    SSI... 00000000 STY... 001F
   INTVAL_START.. 02/22/2022 19:11:30.001815
   INTVAL_END.... 02/22/2022 19:12:27.737573
   USERID_AS.....DATAOWN
   USERID_TK.....
   JOBID.........J0000055
   JOBNAME.......DATAOWN
   JOBNAME2......
   PLEXNAME......SYS1
   DOMAIN........0
   ENG...CARD...8C11/99EA6127...17
   ENG...CPACF...150
   ALG...DES56......2
   ALG...AES128.....2
   ALG...RSA1024....1
   ALG...ECCBP192...1
   ALG...MD5........45
   ALG...RPMD160....15
   ALG...SHA1....... 70
   ALG...SHA3-224... 13
   ALG...SHA3-256... 15
   ALG...SHA3-384... 13
   ALG...SHA3-512... 13
   ALG...SHAKE128... 12
   ALG...SHAKE256... 14
   SRV...CSFKYT..... 2
   SRV...CSFDSG..... 2
   SRV...CSFOWH..... 264
   SRV...CSFOWH1.... 3
   SRV...CSFIQF..... 485
   SRV...CSFIQF2.... 2
```

After identifying the weak algorithms and replacing them with quantum-safe algorithms, ICSF usage statistics can be used to monitor progress with a formatted report of SMF record type 82, subtype 31 (hex ‘001F’).

**Important:** Data that is protected with a retired algorithm must not remain in the system after it is protected by using a quantum-safe algorithm. Removing the data that was encrypted by using a retired algorithm eliminate the risk of an attacker finding that data and breaking the encryption.

In Example 6-15, most algorithms are quantum-safe. For example, we no longer use AES 128; instead, we use AES 256. We can also see that ML-DSA and ML-KEM algorithms are being used.

*Example 6-15 Crypto usage statistics: Checking for quantum-safe algorithms*

```
Type=82 Subtype=001F Crypto Usage Statistics
Written periodically to record crypto usage counts
Mar 2022 15:35:30.00
   TME... 0055A5C8 DTE... 0122070F SID... SP21    SSI... 00000000 STY... 001F
   INTVAL_START.. 03/11/2022 19:33:59.202360
   INTVAL_END.... 03/11/2022 19:35:30.001479
   USERID_AS..... QSAFE
   USERID_TK.....
   JOBID......... T0000046
   JOBNAME....... QSAFE
   JOBNAME2......
   PLEXNAME...... SYS1
   DOMAIN........ 0
   ENG...CARD...8C00/99EA6006... 17
   ENG...CPACF... 5
   ALG...DES112..... 1
   ALG...AES256..... 9
   ALG...ECCP384.... 6
   ALG...MLKE1024... 3
   ALG...MLDU0807... 4
   SRV...CSFDSG..... 2
   SRV...CSFDSV..... 2
   SRV...CSFPKG..... 3
   SRV...CSFPKE..... 1
   SRV...CSFPKX..... 3
   SRV...CSFKYT2.... 2
   SRV...CSFEDH..... 2
   SRV...CSFPKB..... 3
   SRV...CSFCTT2.... 1
```

A

# Finding cryptographic attributes

This appendix describes a process similar to the one in Chapter 5, “Creating a cryptographic inventory” on page 77. However, the SMF record type 82 search made in the text reports generated by running CSFSMFJ is used instead of the IBM CAT database. Also included is an SMF record type 30 sample report that depicts the non-ICSF CPACF usage with indication of job ID and cryptographic algorithm as discussed in 5.4, “IBM z/OS SMF record type 30 cryptographic counters” on page 92.

The result from this suggested process can help you to make qualified choices in terms of where efforts should be allocated in protecting the environment against an attack from a *cryptographically relevant quantum computer* (CRQC).

Some of the steps will require manual analysis to discover which cryptographic algorithms, key lengths, and key labels are used in your programs and applications.

For more information about the creating a cryptographic inventory, see “Establishing a cryptographic inventory” on page 62.

# A.1 Process that was used

In this section, we provide examples of how to proceed with the investigation process by using an ADDI scanning of a set of source modules, as described in “Starting with application source code scan from IBM ADDI” on page 108. Examples of key usage events and a key lifecycle event are provided.

## A.1.1 Examples of finding key usage events

In Figure A-1, we can see that a source module and a few sub modules were ADDI scanned. The program is called DKMSKSA (see the ProgramName column).

| OccurID | APIName | APIDescription | APIM | Or | ParamValue | GroupID | Descriptio | VarName | ProgramName | StartRow |
|---|---|---|---|---|---|---|---|---|---|---|
| 10289 | CSNBENC | Encipher | DES | 9 | 3 | 1 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | 8960 |
| 10289 | CSNBENC | Encipher | DES | 10 | CBC INITIAL DES | 1 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | 8960 |
| 10396 | CSNBENC | Encipher | DES | 9 | 3 | 1 | CRYPTO | CSFSERV-RULE-A-COUNT | DKMSKSA | 9202 |
| 10396 | CSNBENC | Encipher | DES | 10 | CBC INITIAL DES | 1 | CRYPTO | CSFSERV-RULE-A | DKMSKSA | 9202 |

*Figure A-1 Sample1: Source module*

Next, we identify two cryptographic functions that are used by DKMSKSA and map them to the key material.

A call to APIName CSNBENC that is found in DKMSKSA shows the StartRow (8960 and 9202) of the call to CSNBENC.

The value that is shown in the StartRow column is the line number in DKMSKSA where CSNBENC is called.

To map the APIName from CCA to an ICSF service call from the ProgramName and APIDescription, we must take a short detour. In Figure A-2, we can see that the ICSF entry point name (CSFENC) corresponds to the CCA entry point name (CSNBENC).

| **Descriptive service name** | **CCA entry point names** | | **ICSF entry point names** | | **SAF resource name** | **Callable service exit name** |
|---|---|---|---|---|---|---|
| | 31-bit | 64-bit | 31-bit | 64-bit | | |
| Encipher | CSNBENC | CSNEENC | CSFENC | CSFENC6 | CSFENC | CSFENC |

*Figure A-2 Resource names for CCA and ICSF entry points: Encipher*

The CCA and ICSF entry point names tell you which cryptographic operations are being used.

The DKMSRKX source module also does a call to CSNDRKX (see Figure A-3). It can be seen that the CSNDRKX call is made from StartRow 395 in DKMSRKX (ProgramName).

| OccurID | APIName | APIDescription | APIM | Or | ParamValue | GroupID | Descriptio | VarName | ProgramName | StartRow |
|---|---|---|---|---|---|---|---|---|---|---|
| 2518 | CSNDRKX | Remote Key Export | DES | 5 | 1 | 1 | CRYPTO | C-RULE-ARRAY-COUNT | DKMSRKX | 395 |
| 2518 | CSNDRKX | Remote Key Export | DES | 6 | | 1 | CRYPTO | C-RULE-ARRAY | DKMSRKX | 395 |

*Figure A-3 Sample 2: Source module*

Once again to map the APIName from CCA to an ICSF service call from the ProgramName and APIDescription, we must take a short detour. In Figure A-4 we can see that the ICSF entry point name (CSNDRKX) corresponds to the CCA entry point name (CSFRKX).

| Descriptive service name | CCA entry point names | | ICSF entry point names | | SAF resource name | Callable service exit name |
|---|---|---|---|---|---|---|
| | 31-bit | 64-bit | 31-bit | 64-bit | | |
| Remote Key Export | CSNDRKX | CSNFRKX | CSFRKX | CSFRKX6 | CSFRKX | CSFRKX |

*Figure A-4 Resource names for CCA and ICSF entry points - Remote Key Export*

For more information about resource names for CCA and ICSF entry points, see CCA and ICSF entry points, see *ICSF Application Programmer's Guide*, SC14-7508.

Having identified the two ICSF service calls, we now proceed to the next step in the investigation process: identifying the executable module from the source module. This process requires that you understand the steps that are involved in building an executable from application source code and finding the name of the executable.

In our environment, we determined that the DKMSKSA includes the DKMSRKX source code.

Next, we identify the corresponding job execution from the SDSF job listing. At this point, application knowledge is needed in terms of which jobs are involved in executing DKMSKSA and how the program invocation is made. Figure A-5 shows a list of the jobs that execute the DKMSKSA program. The jobs executed on 25 May 2022 (05/25/2022) around 7:00 AM.

```
 Display  Filter  View  Print  Options  Search  Help
-------------------------------------------------------------------------------
SDSF HELD OUTPUT DISPLAY ALL CLASSES LINES 530,551     LINE 24-59 (106)
NP    JOBNAME  JobID    Owner    Prty C ODisp Dest              Tot-Rec  Tot-
      DPEAPCSG JOB07751 DPEAP     144 T HOLD  LOCAL                 194
      DPEAPCSV JOB07752 DPEAP     144 T HOLD  LOCAL                 198
      DPEAPC3  JOB07753 DPEAP     128 T HOLD  LOCAL                 327
      DPEAPCVG JOB07754 DPEAP     128 T HOLD  LOCAL                 213
      DPEAPCVV JOB07755 DPEAP     128 T HOLD  LOCAL                 222
      DPEAPDER JOB07756 DPEAP      96 T HOLD  LOCAL               1,039
      DPEAPGT  JOB07757 DPEAP     128 T HOLD  LOCAL                 223
      DPEAPG   JOB07759 DPEAP     144 T HOLD  LOCAL                 194
      DPEAPPD  JOB07760 DPEAP     128 T HOLD  LOCAL                 258
      DPEAPPE  JOB07761 DPEAP     128 T HOLD  LOCAL                 481
      DPEAPCL  JOB07762 DPEAP     144 T HOLD  LOCAL                 189
      DPEAPPD  JOB07763 DPEAP     144 T HOLD  LOCAL                 187
      DPEAPPG  JOB07764 DPEAP     128 T HOLD  LOCAL                 239
      DPEAPPO  JOB07765 DPEAP     128 T HOLD  LOCAL                 228
      DPEAPRM  JOB07766 DPEAP     128 T HOLD  LOCAL                 229
      DPEAPPV  JOB07767 DPEAP     128 T HOLD  LOCAL                 254
      DPEAPXL  JOB07768 DPEAP     128 T HOLD  LOCAL                 231
      DPEAPOF  JOB07773 DPEAP     128 T HOLD  LOCAL                 238
      DPEAPRM1 JOB07774 DPEAP     112 T HOLD  LOCAL                 812
      DPEAPSDA JOB07775 DPEAP     128 T HOLD  LOCAL                 256
      DPEAPTCV JOB07776 DPEAP     128 T HOLD  LOCAL                 252
```

*Figure A-5 SDSF job listing*

**Important:** The job execution time is needed to identify the relevant SMF records.

For our example, we identified the job IDs that are based on the DKMSKSA program in the JCL libraries. This task also is manual.

Using two jobs as examples for SMF record type 82 subtypes 31 and 44 correlation, we look at job IDs JOB07760 and JOB07761 from Figure A-5 on page 139.

By looking at the JCL library for the two members for DPEAPPD (JOB07760) and DPEADPE (JOB07761), we can determine from the SYSTSIN statement which program is to be run.

In Example A-1, we see job DPEAPPD runs program PANDEC.

*Example A-1 JCL for DPEAPPD*

```
//DPEAPPD JOB (9060-02292-01-33,LU2),'RUN PANDEC ',
   //           MSGCLASS=T,CLASS=G,MSGLEVEL=(1,1)
   /*JOBPARM S=MVSF
   //*--------------------------------------------------------------------
   //RUNAPI   EXEC  PGM=IKJEFT01,REGION=0M
   //STEPLIB  DD DISP=SHR,DSN=DPLMF.KSA0501X.LOAD
   //         DD DISP=SHR,DSN=DPLMF.KSA0501X.LOADD
   //         DD DSN=DB2FSYS.SDSNEXIT,DISP=SHR
   //         DD DSN=DB2FSYS.SDSNLOAD,DISP=SHR
   //SYSTSIN  DD *
   DSN SYST(DB2F)
   RUN PROGRAM(PANDEC) PLAN(KSA0501V)
   END
```

In Example A-2, we see job DPEAPPE runs program PANENC.

*Example A-2 JCL for DPEAPPE*

```
//DPEAPPE JOB (9060-02292-01-33,LU2),'RUN PANENC ',
   //           MSGCLASS=T,CLASS=G,MSGLEVEL=(1,1)
   /*JOBPARM S=MVSF
   //*ABCDE-------------------------------------------------------------
   //RUNAPI   EXEC  PGM=IKJEFT01,REGION=0M
   //STEPLIB  DD DISP=SHR,DSN=DPLMF.KSA0501X.LOAD
   //         DD DISP=SHR,DSN=DPLMF.KSA0501X.LOADD
   //         DD DSN=DB2FSYS.SDSNEXIT,DISP=SHR
   //         DD DSN=DB2FSYS.SDSNLOAD,DISP=SHR
   //SYSTSIN  DD *
   DSN SYST(DB2F)
   RUN PROGRAM(PANENC) PLAN(KSA0501V)
   END
```

Both programs (PANDEC and PANENC) call the DKMSKSA program. Although the input parameters are different, this issue is irrelevant in the context of mapping the key material and ICSF services to DKMSKSA.

In Example A-3, we can see PANENC calls DKMSKSA.

*Example A-3 PANENC calls DKMSKSA*

```
BA-CALL-DKMSKSA SECTION.
      ****************************************************************
      * CALL DKMSKSA                                                 *
      ****************************************************************

           CALL DKMSKSA USING DAPI-KSA-V01
```

```
           DISPLAY 'API RETURN-CODE = ' RETURN-CODE
```

We have two job IDs (JOB07760 and JOB07761) that run program DKMSKSA. Now, we need to identify the ICSF SMF record type 82 entries. We want to search for subtypes 31, 40 – 42, and 44 – 46 in the timeframe in which the job was run. In our environment, SMF records are collected in half-hour intervals.

In Example A-4, the SMF record interval on 05/25/2022 is 07:00:30 - 07:30:30 when JOB07760 (DEAPPD) user DPEAP was run.

*Example A-4 SMF record type 82 subtype 31 (hex '001F')*

```
Subtype=001F Crypto Usage Statistics
    Written periodically to record crypto usage counts
    25 May 2022 7:30:30.11
       TME... 00293EA3 DTE... 0122145F SID... MVSF    SSI... 00000000 STY... 001F
       INTVAL_START.. 05/25/2022 07:00:30.026731
       INTVAL_END.... 05/25/2022 07:30:30.111713
       USERID_AS..... DPEAP
       USERID_TK.....
       JOBID......... JOB07760
       JOBNAME....... DPEAPPD
       JOBNAME2......
       PLEXNAME...... MVSFPLEX
       DOMAIN........ 0
       ENG...CARD...7C01/93AACJPJ... 5
       ENG...CARD...7C03/93AACJN6... 5
       ALG...DES112..... 10
       SRV...CSFENC..... 10
```

In JOB07761 (DEAPPE), user DPEAP was run at the same time as JOB07760 (see Example A-5).

*Example A-5 SMF record type 82 subtype 31 (hex '001F')*

```
Subtype=001F Crypto Usage Statistics
    Written periodically to record crypto usage counts
    25 May 2022 7:30:30.11
       TME... 00293EA3 DTE... 0122145F SID... MVSF    SSI... 00000000 STY... 001F
       INTVAL_START.. 05/25/2022 07:00:30.026731
       INTVAL_END.... 05/25/2022 07:30:30.111713
       USERID_AS..... DPEAP
       USERID_TK.....
       JOBID......... JOB07761
       JOBNAME....... DPEAPPE
       JOBNAME2......
       PLEXNAME...... MVSFPLEX
       DOMAIN........ 0
       ENG...CARD...7C01/93AACJPJ... 5
       ENG...CARD...7C03/93AACJN6... 5
       ALG...DES112..... 10
       SRV...CSFENC..... 10
```

The information in the different subtypes varies. Subtypes 31 and 40 – 42 all contain job name and user ID; sub-types 44 – 46 are accumulated over user ID and key ID. For the ICSF service that is called CSFENC, we must find a subtype 44 (hex ‘002C’) that maps to CSFENC for user DPEAP over the relevant period.

The usage count in subtype 44 records might be lager than identified in the subtype 31 records. In Example A-6, we can see a record that has 20 calls to CSFENC for user DPEAP in the relevant period of our application. This result corresponds with the two usage statistics records in which the two jobs in Example A-4 and Example A-5 had 10 calls to CSFENC each.

In some calls that were made to ICSF, the key label might not be present; rather, the key fingerprint or key check value is present to help identify a key.

*Example A-6 Subtype 44 key usage event*

```
Subtype=002C CCA Symmetric Key Usage Event
    Written for usage events related to symmetric CCA tokens
    25 May 2022 11:02:46.08
       TME... 003CADA0 DTE... 0122145F SID... MVSF    SSI... 00000000 STY... 002C
       STOD.. 05/25/2022 05:00:48.402960
       ETOD.. 05/25/2022 11:02:45.849556
       SRV... CSFENC
       USGC.. 20
       LBL... VDEKDESW3.00.500000.203704.IX0001                              DATA
       TOKFMT Fixed
       KALG.. DES
       KSEC.. Wrapped by MK
       CV.... '00007D0003600081'x (DATA*)
       TIV... 'C97A80A9'x
       KFP... 010105D5B74D
              ENCZ.. 'D5B74D'x
    End User Identity...
       USRI.. DPEAP
```

A key usage event record is available that relates to two job executions. The key label is identified as VDEKDESW3.00.500000.203704.IX0001. The key label and details can also be found in the IBM CAT GUI (see Figure A-6).

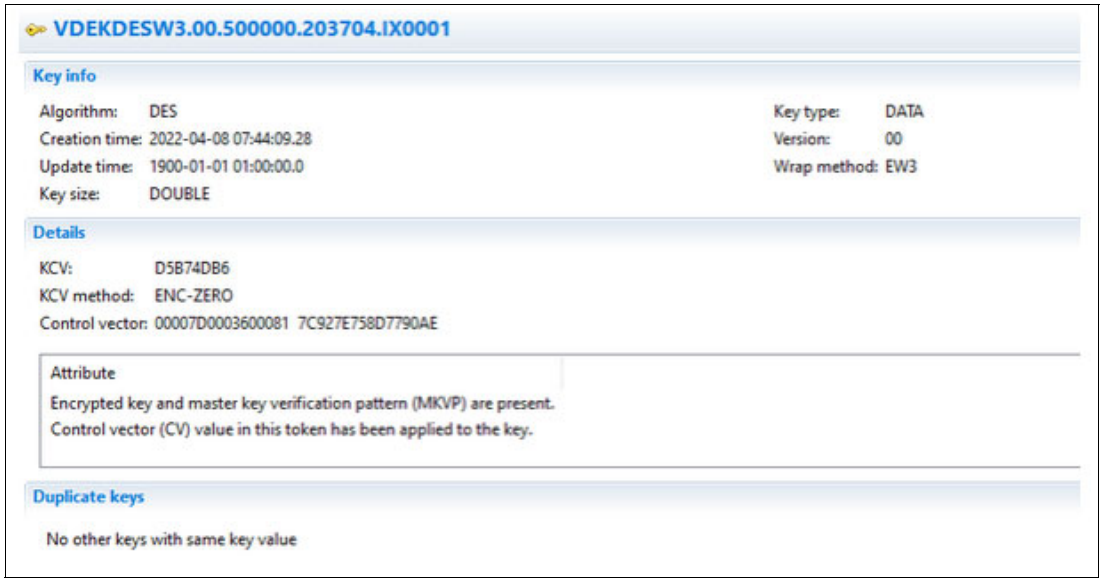



*Figure A-6 Key label and details*

## A.1.2 Examples of finding key lifecycle events

Starting from the SDSF job listing that is shown in Figure A-5 on page 139, we use job ID JOB07756 as an example for cryptographic usage and key lifecycle events. For this example, we correlate SMF record type 82 subtypes 31 and 40.

JOB07756 is run by using the JCL member DPEAPDER (see Example A-7). Here, we find that program DKMSKSA calls DERKEYS.

*Example A-7 JCL for DPEAPDER*

```
//DPEAPDER JOB (9060-02292-01-33,LU2),'TEST DERKEYS',
  //            MSGCLASS=T,CLASS=G,MSGLEVEL=(1,1)
  /*JOBPARM S=MVSF
  //RUNAPI   EXEC  PGM=IKJEFT01,REGION=0M
  //STEPLIB  DD DISP=SHR,DSN=DPLMF.KSA0501X.LOAD
  //         DD DISP=SHR,DSN=DPLMF.KSA0501X.LOADD
  //         DD DSN=DB2FSYS.SDSNEXIT,DISP=SHR
  //         DD DSN=DB2FSYS.SDSNLOAD,DISP=SHR
  //SYSTSIN  DD *
  DSN SYST(DB2F)
  RUN PROGRAM(DERKEYS) PLAN(KSA0501V)
  END
```

For JOB07756 (DEAPDER, user DPEAP), we see in Example A-8 that the record covers the interval of 05/25/2022, 07:00:30 - 07:30:30.

*Example A-8 SMF record type 82 subtype 31 (hex '001F')*

```
Subtype=001F Crypto Usage Statistics
    Written periodically to record crypto usage counts
    25 May 2022 7:30:30.11
       TME... 00293EA3 DTE... 0122145F SID... MVSF    SSI... 00000000 STY... 001F
       INTVAL_START.. 05/25/2022 07:00:30.026731
       INTVAL_END.... 05/25/2022 07:30:30.111713
       USERID_AS..... DPEAP
       USERID_TK.....
       JOBID......... JOB07756
       JOBNAME....... DPEAPDER
       JOBNAME2......
       PLEXNAME...... MVSFPLEX
       DOMAIN........ 0
       ENG...CARD...7C01/93AACJPJ... 20
       ENG...CARD...7C03/93AACJN6... 21
       ENG...CPACF... 1
       ALG...DES112..... 7
       ALG...DES168..... 51
       ALG...SHA1....... 1
       SRV...CSFKTB..... 8
       SRV...CSFOWH..... 1
       SRV...CSFDKG..... 8
       SRV...CSFRKX..... 17
```

Among the ICSF services called in this job, CSFRKX is called 17 times during the execution. CSFRKX triggers a key lifecycle event, in this case subtype 40 (hex '0028') records are generated. These records are written shortly after the call occurs and is not an aggregated record as with the key usage event records. We can then search for a subtype 40 records in the timeframe of the job execution and for the job name DPEAPDER (see Example A-9).

*Example A-9 SMF record type 82 subtype 40 (hex '0028')*

```
Subtype=0028 CCA Symmetric Key Lifecycle Event
    Written for lifecycle events related to symmetric CCA tokens
    25 May 2022 7:01:41.30
       TME... 00269B52 DTE... 0122145F SID... MVSF    SSI... 00000000 STY... 0028
       KEV... Key Exported
       SRV... CSFRKX
       LBL... VZMKDES.00.3000
EXPORTER
       KFP... 01010562C123
              ENCZ.. '62C123'x
       TOKFMT Fixed
       KALG.. DES
       KSEC.. Wrapped by MK
       CV.... '00417E0003600081'x (EXPORTER/OKEYXLAT)
       TIV... '384479E0'x
    ICSF Server Identity...
       USRI.. CSF00000
       GRPN.. CC
       JBN... CSF
```

```
      RST... 7:35:21.23
      RSD... 24 May 2022
      SUID.. 4040404040404040
   End User Identity...
      USRI.. DPEAP
      GRPN.. DP
      JBN... DPEAPDER
      RST... 7:01:41.18
      RSD... 25 May 2022
      SUID.. 4040404040404040
```

The key label in this case is identified as VZMKDES.00.3000 and can be found in IBM CAT, as shown in Figure A-7.

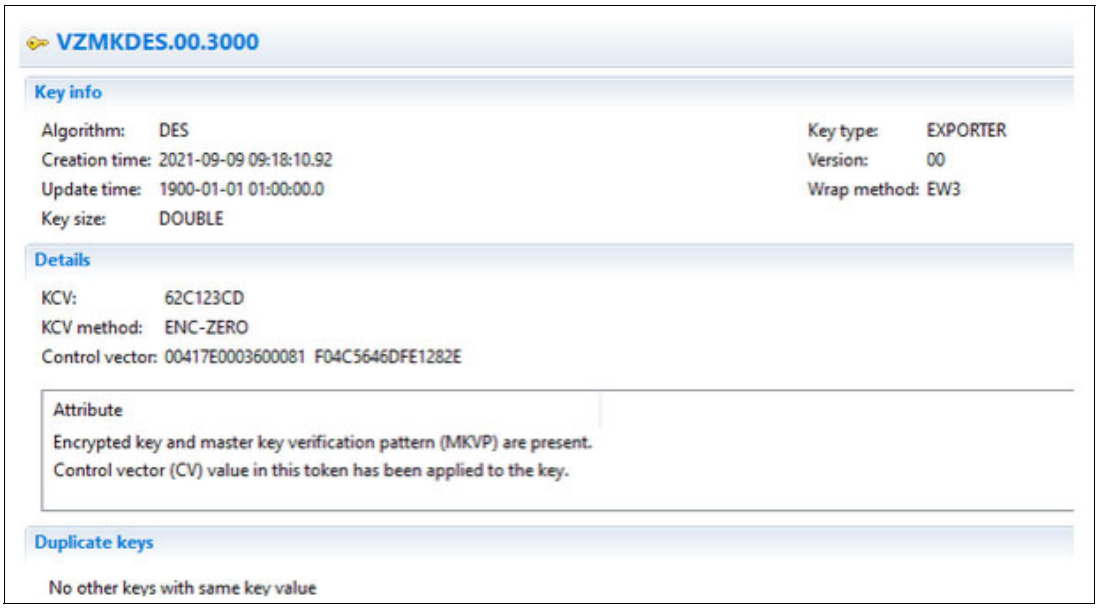



*Figure A-7 IBM CAT - key label and details*

## A.1.3 Summary

We started with an ADDI scan of programs, DKMSKSA and DKMSRKX. Then, we identified two ICSF service calls, the jobs that performed them, and the cryptographic keys that were used in the ICSF service calls and key lifecycle events.

Searching through ICSF SMF records and SDSF job listings can be a significant undertaking and time consuming, depending on the use of cryptographic functions. A deep understanding of your JCL, applications, and programs is essential.

In addition, ICSF SMF records are not always written at the same time. In our environment, SMF type record 82 subtype 31 are written twice every hour (at top and bottom of the hour), and subtypes 40 – 42 are written as they occur. SMF record type 82 subtypes 44 – 46 are written at six-hour intervals, beginning at the time ICSF is started. The intervals at which the SMF records are written must be considered when identifying application and program use of ICSF service calls and their associated cryptographic material.

## A.2 z/OS SMF record type 30 sample output

Using the z/OS SMF record type 30 as outlined in 5.4, “IBM z/OS SMF record type 30 cryptographic counters” on page 92, a report can be generated as shown in Figure A-8 with jobnumber (SMF30JNM), crypto ID (SMF30CID), crypto count (SMF30CCC), and the descriptive value of the crypto ID.

| SMF30SID | SMF30DTE | SMF30TME | SMF30STM | SMF30STM | SMF30PGM | SMF30JNM | SMF30CID | SMF30CCC | SMF30_CrypCtrs |
|---|---|---|---|---|---|---|---|---|---|
| SYSD | 24259 | 6:00:21 | PPBPRPDS | S10PGENC | ZIP390 | J0273607 | 9 | 1470 | KM_AES_256 |
| SYSC | 24259 | 6:00:23 | D21ZIP7 | STEP1 | BPXPRFC | S0272886 | 9 | 1609 | KM_AES_256 |
| SYSE | 24259 | 6:01:58 | WEBAXSPA | STEP1 | BPXPRFC | S0270132 | 54 | 1719 | KMCTR_AES_128 |
| SYSE | 24259 | 6:01:58 | WEBAXSPA | STEP1 | BPXPRFC | S0270132 | 82 | 1767 | KLMD_SHA_1 |
| SYSD | 24259 | 6:03:57 | WEBAXPCP | STEP1 | BPXPRFC | S0273581 | 54 | 209486 | KMCTR_AES_128 |
| SYSD | 24259 | 6:03:57 | WEBAXPCP | STEP1 | BPXPRFC | S0273581 | 72 | 90337 | KIMD_SHA_1 |
| SYSD | 24259 | 6:03:57 | WEBAXPCP | STEP1 | BPXPRFC | S0273581 | 82 | 209555 | KLMD_SHA_1 |
| SYSB | 24259 | 6:05:12 | SAILPTCK | PWDCHG2 | IKJEFT01 | J0273744 | 83 | 90017 | KLMD_SHA_256 |
| SYSC | 24259 | 6:09:21 | EDISDBTG | *OMVSEX | BPXPRECP | S0272886 | 54 | 1603 | KMCTR_AES_128 |
| SYSD | 24259 | 6:09:26 | EDISDBTN | *OMVSEX | BPXPRECP | S0273598 | 54 | 2921 | KMCTR_AES_128 |
| SYSD | 24259 | 6:09:26 | EDISDBTN | *OMVSEX | BPXPRECP | S0273598 | 82 | 1174 | KLMD_SHA_1 |
| SYSE | 24259 | 6:11:09 | FADPR78F | *OMVSEX | BPXPRECP | S0273037 | 54 | 20619 | KMCTR_AES_128 |
| SYSE | 24259 | 6:11:09 | FADPR78F | *OMVSEX | BPXPRECP | S0273037 | 72 | 8340 | KIMD_SHA_1 |
| SYSE | 24259 | 6:11:09 | FADPR78F | *OMVSEX | BPXPRECP | S0273037 | 82 | 13284 | KLMD_SHA_1 |
| SYSD | 24259 | 6:18:31 | FADPR77F | *OMVSEX | BPXPRECP | S0273598 | 54 | 2844 | KMCTR_AES_128 |
| SYSD | 24259 | 6:18:31 | FADPR77F | *OMVSEX | BPXPRECP | S0273598 | 72 | 1051 | KIMD_SHA_1 |
| SYSD | 24259 | 6:18:31 | FADPR77F | *OMVSEX | BPXPRECP | S0273598 | 82 | 1776 | KLMD_SHA_1 |
| SYSK | 24259 | 6:20:06 | SMFFTP | FTP | FTP | S0149269 | 23 | 57291 | KMC_AES_128 |
| SYSK | 24259 | 6:20:06 | SMFFTP | FTP | FTP | S0149269 | 73 | 76379 | KIMD_SHA_256 |
| SYSK | 24259 | 6:20:06 | SMFFTP | FTP | FTP | S0149269 | 82 | 19100 | KLMD_SHA_1 |
| SYSK | 24259 | 6:20:06 | SMFFTP | FTP | FTP | S0149269 | 83 | 38200 | KLMD_SHA_256 |

*Figure A-8 SMF 30 sample output*

This report can be used when identifying non-quantum safe algorithms (also known as, post-quantum cryptography (PQC) weak algorithms), like SHA1.

To extract SMF record type 30 cryptographic counters and create a report, a sample REXX program and JCL to execute it, can be found at:
https://github.com/IBM/ICSF-Education/tree/main/Quantum-Safe%20Redbook%20Samples/SMF30%20samples

B

# Generating quantum-safe keys

The examples in this appendix are REXX executables that can be used to generate quantum-safe keys that use CCA and PKCS #11.

This appendix includes the following topics:

# B.1 CCA AES 256-bit key generation REXX sample

A CCA AES 256-bit key generation REXX sample is shown in Example B-1.

*Example B-1 CCA AES 256-bit key generation REXX sample*

```
/* rexx */
/*--------------------------------------------------------------------*/
/* Generate a secure CCA 256-bit AES CIPHER key                       */
/*--------------------------------------------------------------------*/

/* expected results */
ExpRC = '00000000'x ;
ExpRS = '00000000'x ;


/*--------------------------------------------------------------------*/
/* Build skeleton token with key usage and key management             */
/*--------------------------------------------------------------------*/

KTB2_rule_array = 'INTERNAL' ||,
                  'AES     ' ||,
                  'NO-KEY  ' ||,
                  'CIPHER  ' ||,
                  'ENCRYPT ' ||,
                  'DECRYPT ' ||,
                  'C-XLATE ' ||,
                  'ANY-MODE' ||,
                  'NOEX-SYM' ||,
                  'NOEX-RAW' ||,
                  'NOEXUASY' ||,
                  'NOEXAASY' ||,
                  'NOEX-DES' ||,
                  'NOEX-AES' ||,
                  'NOEX-RSA' ||,
                  'XPRTCPAC'

call CSNBKTB2

/*--------------------------------------------------------------------*/
/* Generate the AES key using the skeleton token from KTB2            */
/*--------------------------------------------------------------------*/
KGN2_Rule_Array = 'AES     ' ||,
                  'OP      '
KGN2_clear_key_Bit_Len = '00000100'x  /* 256-bit */
KGN2_key_Type_1 = 'TOKEN   '
KGN2_key_Type_2 = ''
KGN2_gen_key_1_Len     = '000002D5'x
KGN2_gen_key_1 = left(KTB2_target_key_token,c2d(KGN2_gen_key_1_Len))

call CSNBKGN2


exit
```

```
/* ------------------------------------------------------------ */
/* CSNBKTB2 - Key Token Build2                                  */
/*                                                              */
/* Builds a variable-length AES skeleton token.                 */
/*                                                              */
/* See the ICSF Application Programmer's Guide for more details.  */
/* ------------------------------------------------------------ */
CSNBKTB2:

  KTB2_rc               = 'ffffffff'x ;
  KTB2_rs               = 'ffffffff'x ;
  KTB2_exit_Length      = '00000000'x ;
  KTB2_exit_Data        = '' ;
  KTB2_key_name_len     = '00000000'x ;
  KTB2_key_name         = '';
  KTB2_user_data_Len    = '00000000'x ;
  KTB2_user_data        = '';
  KTB2_token_data_Len   = '00000000'x ;
  KTB2_token_data       = '';
  KTB2_clear_key        = '';
  KTB2_service_data     = '';
  KTB2_service_data_Len  = D2C(length(KTB2_service_data),4) ;
  KTB2_target_key_token_Len = d2c(725,4) ;
  KTB2_target_key_token  = copies('00'x,c2d(KTB2_target_key_token_Len)) ;
  KTB2_clear_key_bit_Len = '00000000'x;
  KTB2_Rule_count       = D2C(length(KTB2_rule_array)/8,4) ;



 address linkpgm 'CSNBKTB2'                                  ,
                 'KTB2_rc'              'KTB2_rs'         ,
                 'KTB2_exit_Length'     'KTB2_exit_Data'  ,
                 'KTB2_rule_count'      'KTB2_rule_array' ,
                 'KTB2_clear_key_bit_Len'                 ,
                 'KTB2_clear_key'                         ,
                 'KTB2_key_name_Len'    'KTB2_key_name'   ,
                 'KTB2_user_data_Len'   'KTB2_user_data'  ,
                 'KTB2_token_data_Len'  'KTB2_token_data' ,
                 'KTB2_service_data_Len' 'KTB2_service_data' ,
                 'KTB2_target_key_token_Len' 'KTB2_target_key_token' ;

  KTB2_target_key_token = ,
     substr(KTB2_target_key_token,1,c2d(KTB2_target_key_token_len))

 If (KTB2_RC <> ExpRC) | (KTB2_RS <> ExpRS) then
   do;
      say 'KTB2 failed : rc =' c2x(KTB2_RC) 'rs =' c2x(KTB2_RS)
   end;
 else
   say 'KTB2 successful : rc =' c2x(KTB2_RC) 'rs =' c2x(KTB2_RS)

return
/* ------------------------------------------------------------ */
/* CSNBKGN - Key Generate                                       */
/*                                                              */
```

```
/* Generates either one or two DES or AES keys encrypted under a   */
/* master key (internal form) or KEK (external form).              */
/*                                                                 */
/* See the ICSF Application Programmer's Guide for more details.   */
/* --------------------------------------------------------------- */
CSNBKGN2:

 KGN2_rc                = 'ffffffff'x ;
 KGN2_rs                = 'ffffffff'x ;
 KGN2_Exit_Length       = '00000000'x ;
 KGN2_Exit_Data         = '' ;
 KGN2_Rule_count        = D2C(length(KGN2_rule_array)/8,4)
 KGN2_key_Name_1_Len    = '00000000'x ;
 KGN2_key_Name_1        = '';
 KGN2_key_Name_2_Len    = '00000000'x ;
 KGN2_key_Name_2        = '';
 KGN2_user_data_1_Len   = '00000000'x ;
 KGN2_user_data_1       = '';
 KGN2_user_data_2_Len   = '00000000'x ;
 KGN2_user_data_2       = '';
 KGN2_KEK_1_Len         = '00000000'x ;
 KGN2_KEK_1             = '';
 KGN2_KEK_2_Len         = '00000000'x;
 KGN2_KEK_2             = '';
 KGN2_gen_key_2_Len     = '00000000'x;
 KGN2_gen_key_2         = '';


  address linkpgm 'CSNBKGN2'                          ,
                 'KGN2_rc'              'KGN2_rs'     ,
                 'KGN2_Exit_Length'     'KGN2_Exit_Data'   ,
                 'KGN2_Rule_Count'      'KGN2_Rule_Array' ,
                 'KGN2_clear_key_Bit_Len'          ,
                 'KGN2_key_Type_1'      'KGN2_key_Type_2' ,
                 'KGN2_key_Name_1_Len'  'KGN2_key_Name_1' ,
                 'KGN2_key_Name_2_Len'  'KGN2_key_Name_2' ,
                 'KGN2_user_data_1_Len' 'KGN2_user_data_1' ,
                 'KGN2_user_data_2_Len' 'KGN2_user_data_2' ,
                 'KGN2_KEK_1_Len'       'KGN2_KEK_1'  ,
                 'KGN2_KEK_2_Len'       'KGN2_KEK_2'  ,
                 'KGN2_gen_key_1_Len'   'KGN2_gen_key_1' ,
                 'KGN2_gen_key_2_Len'   'KGN2_gen_key_2' ;

  If (KGN2_RC <> ExpRC) | (KGN2_RS <> ExpRS) then
   do;
    say 'KGN2 failed: rc =' c2x(KGN2_RC) 'rs =' c2x(KGN2_RS)
   end;
  else
    say 'KGN2 successful: rc =' c2x(KGN2_RC) 'rs =' c2x(KGN2_RS)

 KGN2_gen_key_1 = substr(KGN2_gen_key_1,1,c2d(KGN2_gen_key_1_len))
 KGN2_gen_key_2 = substr(KGN2_gen_key_2,1,c2d(KGN2_gen_key_2_len))

Return
```

## B.2 PKCS #11 AES 256-bit key generation REXX sample

A PKCS #11 AES 256-bit key generation REXX sample is shown in Example B-2.

*Example B-2 PKCS #11 AES 256-bit key generation REXX sample*

```
/*rexx*/
/*--------------------------------------------------------------------*/
/* Generate a secure 256-bit PKCS #11 AES key                         */
/*--------------------------------------------------------------------*/
Call TCSetup

/* expected results */
ExpRC = '00000000'x ;
ExpRS = '00000000'x ;

/*--------------------------------------------------------------------*/
/* Generate the AES key using the attribute list                      */
/*--------------------------------------------------------------------*/

 GSK_Handle      = Left('QSAFE.TEST.TOKEN',44) ;
 GSK_AttrList    = '0007'x ||,                  /* number attributes */
  CKA_CLASS        ||'0004'x || CKO_SECRET_KEY  ||,
  CKA_KEY_TYPE     ||'0004'x || CKK_AES         ||,
  CKA_VALUE_LEN    ||'0004'x || '00000020'x     ||,   /* AES 256-bit */
  CKA_TOKEN        ||'0001'x || CK_TRUE         ||,
  CKA_IBM_SECURE   ||'0001'x || CK_TRUE         ||,
  CKA_ENCRYPT      ||'0001'x || CK_TRUE         ||,
  CKA_DECRYPT      ||'0001'x || CK_TRUE


 Call CSFPGSK;



exit
/* ---------------------------------------------------------------- */
/* PKCS #11 Generate Secret Key                                     */
/* Use the generate secret key callable service to generate a       */
/* secret key or set of domain parameters.                          */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */
 CSFPGSK:
  GSK_RC           = 'FFFFFFFF'x ;
  GSK_RS           = 'FFFFFFFF'x ;
  GSK_Exit_Length  = '00000000'x ;
  GSK_Exit_Data    = '' ;
  GSK_Rule_Count        = '00000001'x;
  GSK_Rule_Array        = 'KEY     ';
  GSK_Parms_List        = ''
  GSK_Parms_List_Length = '00000000'x

  GSK_AttrListLength   = D2C( Length( GSK_AttrList ),4);
```

```
   /* call GSK */
 address linkpgm 'CSFPGSK'                                        ,
                 'GSK_RC'                'GSK_RS'                 ,
                 'GSK_Exit_Length'       'GSK_Exit_Data'          ,
                 'GSK_Handle'                                     ,
                 'GSK_Rule_Count'        'GSK_Rule_Array'         ,
                 'GSK_AttrListLength'    'GSK_AttrList'           ,
                 'GSK_Parms_List_Length' 'GSK_Parms_List'         ;



If (GSK_RC <> ExpRC) | (GSK_RS <> ExpRS) then
  say 'GSK failed: rc =' c2x(GSK_rc) 'rs =' c2x(GSK_rs) ;
else
  say 'GSK successful : rc =' c2x(GSK_rc) 'rs =' c2x(GSK_rs) ;

return

/* ---------------------------------------------------------------- */
/*                                                                  */
/* ---------------------------------------------------------------- */
TCSetup:

CKK_AES               = '0000001F'X

CKO_SECRET_KEY        = '00000004'X

CKA_CLASS             = '00000000'X
CKA_TOKEN             = '00000001'X
CKA_IBM_SECURE        = '80000006'X
CKA_KEY_TYPE          = '00000100'X
CKA_ENCRYPT           = '00000104'X;
CKA_DECRYPT           = '00000105'X;
CKA_VALUE_LEN         = '00000161'X

CK_TRUE               = '01'x
CK_FALSE              = '00'x

Return

EXIT;
```

## B.3 CCA ML-DSA key pair generation REXX sample

A CCA ML-DSA key pair generation REXX sample is shown in Example B-3.

*Example B-3 CCA ML-DSA key pair generation REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* Generate a secure ML-DSA CCA key pair                              */
/*--------------------------------------------------------------------*/

 /* expected results */
 ExpRc = '00000000'x
 ExpRs = '00000000'x

/*--------------------------------------------------------------------*/
/* Build skeleton token with key usage                                */
/*--------------------------------------------------------------------*/
 PKB_Rule_Count   = '00000002'x ;
 PKB_Rule_Array   = 'QSA-PAIR' ||,
                    'U-DIGSIG'

 /* Pure ML-DSA-87 KVS */
 PKB_KVS    = '05'x   ||, /* Alg Id */
              '00'x   ||, /* clear key format */
              '0807'x ||, /* Alg param */
              '0000'x ||, /* clear key len */
              '0000'x     /* Reserved */

 /* Pre-hash ML-DSA-87 KVS */
 /*
 PKB_KVS    = '07'x   ||, /* Alg Id */
              '00'x   ||, /* clear key format */
              '0807'x ||, /* Alg param */
              '0000'x ||, /* clear key len */
              '0000'x     /* Reserved */
 */
 call CSNDPKB

/*--------------------------------------------------------------------*/
/* Generate the ML-DSA key pair using the skeleton token from PKB */
/*--------------------------------------------------------------------*/
 PKG_Rule_Array = 'MASTER  '
 PKG_Skeleton_Key        = PKB_Token;
 PKG_Skeleton_Key_length = PKB_Token_length;

 call CSNDPKG

 Exit
/*--------------------------------------------------------------------*/
/* PKA Key Token Build - used to create PKA key tokens.               */
/*                                                                    */
/* See the ICSF Application Programmer's Guide for more details.      */
/*--------------------------------------------------------------------*/
```

```
CSNDPKB:

/* initialize parameter list */
PKB_Rc           = 'FFFFFFFF'x ;
PKB_Rs           = 'FFFFFFFF'x ;
Exit_Length      = '00000000'x ;
Exit_Data        = '' ;
PKB_KVS_Length   = d2c(length(PKB_KVS),4) ;
PKB_UAD_Length   = '00000000'x ;
PKB_UAD          = ''
PKB_PrivName_Len = '00000000'x ;
PKB_PrivName     = ''
Reserved2_Length = '00000000'x ; Reserved2 = '' ;
Reserved3_Length = '00000000'x ; Reserved3 = '' ;
Reserved4_Length = '00000000'x ; Reserved4 = '' ;
Reserved5_Length = '00000000'x ; Reserved5 = '' ;
PKB_Token_Length = d2c(8000,4) ;
PKB_Token        = copies('00'x,8000) ;

/* call CSNDPKB */
address linkpgm 'CSNDPKB'                              ,
                'PKB_Rc'           'PKB_Rs'         ,
                'Exit_Length'      'Exit_Data'      ,
                'PKB_Rule_Count'   'PKB_Rule_Array' ,
                'PKB_KVS_Length'   'PKB_KVS'        ,
                'PKB_PrivName_Len' 'PKB_PrivName'   ,
                'PKB_UAD_Length'   'PKB_UAD'        ,
                'Reserved2_Length' 'Reserved2'      ,
                'Reserved3_Length' 'Reserved3'      ,
                'Reserved4_Length' 'Reserved4'      ,
                'Reserved5_Length' 'Reserved5'      ,
                'PKB_Token_Length' 'PKB_Token'      ;

if (PKB_Rc \= ExpRc | PKB_Rs \= ExpRs) then
  say 'PKB failed: rc =' c2x(PKB_Rc) 'rs =' c2x(PKB_Rs) ;
else
 do ;
  say 'PKB successful: rc =' c2x(PKB_Rc) 'rs =' c2x(PKB_Rs) ;
  PKB_Token = substr(PKB_Token,1,c2d(PKB_Token_Length)) ;
 end

 return

/* ------------------------------------------------------------- */
/* PKA Key Generate  - Used to generate PKA key pairs            */
/*                                                               */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ------------------------------------------------------------- */
CSNDPKG:

 PKG_rc          = 'FFFFFFFF'x ;
 PKG_rs          = 'FFFFFFFF'x ;
 PKG_Exit_length = '00000000'x ;
 PKG_Exit_Data   = '' ;
 PKG_Rule_count          = d2c( length(PKG_Rule_Array)/8,4 )
```

```
 PKG_Token_length        = '00001F40'x ;
 PKG_Token               = copies('00'x,c2d(PKG_token_length)) ;
 PKG_Regen_data = ''
 PKG_Regen_Data_length = d2c( length(PKG_Regen_data),4 )
 PKG_Transport_Key_Id = ''

  address linkpgm 'CSNDPKG' ,
                 'PKG_rc'                  'PKG_rs' ,
                 'PKG_Exit_length'         'PKG_Exit_Data' ,
                 'PKG_Rule_Count'          'PKG_Rule_Array' ,
                 'PKG_Regen_Data_length'   'PKG_Regen_Data' ,
                 'PKG_Skeleton_Key_length' 'PKG_Skeleton_Key' ,
                 'PKG_Transport_Key_Id'     ,
                 'PKG_Token_length'        'PKG_Token' ;

if (PKG_rc \= ExpRc | PKG_rs \= ExpRs) then
  say 'PKG failed: rc =' c2x(PKG_rc) 'rs =' c2x(PKG_rs)
else
 Do;
  say 'PKG  successful : rc =' c2x(PKG_rc) 'rs =' c2x(PKG_rs) ;
  PKG_Token = substr(PKG_Token,1,c2d(PKG_Token_length)) ;
 End;

return
```

## B.4 PKCS #11 CRYSTALS-Dilithium key pair generation REXX sample

A PKCS #11 CRYSTALS-Dilithium key pair generation REXX sample is shown in Example B-4.

*Example B-4 PKCS #11 CRYSTALS-Dilithium key pair generation REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* Generate a secure PKCS #11 Dilithium key pair                      */
/*--------------------------------------------------------------------*/

/* expected results */
ExpRC = '00000000'x ;
ExpRS = '00000000'x ;

Call TCSetup

GKP_Handle             = Left('QSAFE.TEST.TOKEN',44)

GKP_PrivKey_Attr_List = '0005'x||,
       CKA_CLASS      ||'0004'x|| CKO_PRIVATE_KEY          ||,
       CKA_KEY_TYPE   ||'0004'x|| CKK_IBM_DILITHIUM        ||,
       CKA_TOKEN      ||'0001'x|| CK_TRUE                  ||,
       CKA_SIGN       ||'0001'x|| CK_TRUE                  ||,
       CKA_IBM_SECURE ||'0001'x|| CK_TRUE

GKP_PubKey_Attr_List = '0005'x||,
       CKA_CLASS              ||'0004'x|| CKO_PUBLIC_KEY    ||,
       CKA_KEY_TYPE           ||'0004'x|| CKK_IBM_DILITHIUM ||,
       CKA_IBM_DILITHIUM_MODE ||'000D'x|| DER_OID_8_7_R3    ||,
       CKA_TOKEN              ||'0001'x|| CK_TRUE           ||,
       CKA_VERIFY             ||'0001'x|| CK_TRUE

Call CSFPGKP;

Exit
/* ---------------------------------------------------------------- */
/* PKCS #11 Generate Key Pair                                       */
/* Use the PKCS #11 Generate Key Pair callable service to generate */
/* an RSA, DSA, Elliptic Curve, Diffie-Hellman, Dilithium (LI2) or */
/* Kyber key pair.                                                  */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ---------------------------------------------------------------- */
CSFPGKP:
 GKP_RC = 'FFFFFFFF'x
 GKP_RS = 'FFFFFFFF'x
 GKP_Exit_Length = '00000000'x
 GKP_Exit_Data = ''
 GKP_Rule_Count = '00000000'x
 GKP_Rule_Array = ''
 GKP_PubKey_Handle = copies(' ',44)
```

```
 GKP_PrivKey_Handle = copies(' ',44)

 GKP_PubKey_Attr_List_Length = D2C(Length(GKP_PubKey_Attr_List),4)
 GKP_PrivKey_Attr_List_Length = D2C(Length(GKP_PrivKey_Attr_List),4)

 address linkpgm 'CSFPGKP',
                 'GKP_RC' 'GKP_RS',
                 'GKP_Exit_Length' 'GKP_Exit_Data',
                 'GKP_Handle',
                 'GKP_Rule_Count' 'GKP_Rule_Array',
                 'GKP_PubKey_Attr_List_Length',
                 'GKP_PubKey_Attr_List',
                 'GKP_PubKey_Handle',
                 'GKP_PrivKey_Attr_List_Length',
                 'GKP_PrivKey_Attr_List',
                 'GKP_PrivKey_Handle'

   if (GKP_RC \= ExpRC | GKP_RS \= ExpRS) Then
     say 'GKP failed: rc =' c2x(GKP_rc) 'rs =' c2x(GKP_rs) ;
   else
     say 'GKP successful : rc =' c2x(GKP_rc) 'rs =' c2x(GKP_rs) ;

return;

/* ---------------------------------------------------------------- */
/*                                                                  */
/* ---------------------------------------------------------------- */
TCSetup:

DER_OID_8_7_R3         = '060B2B0601040102820B070807'X

CKK_IBM_DILITHIUM      = '80010023'X

CKO_PUBLIC_KEY         = '00000002'X
CKO_PRIVATE_KEY        = '00000003'X

CKA_IBM_SECURE         = '80000006'X
CKA_KEY_TYPE           = '00000100'X
CKA_CLASS              = '00000000'X
CKA_TOKEN              = '00000001'X
CKA_IBM_DILITHIUM_MODE = '80000010'X
CKA_SIGN               = '00000108'X;
CKA_VERIFY             = '0000010A'X;

CK_TRUE                = '01'x
CK_FALSE               = '00'x
Return

EXIT;
```

## B.5 CCA ML-KEM key pair generation REXX sample

A CCA ML-KEM key pair generation REXX sample is shown in Example B-5.

*Example B-5 CCA ML-KEM key pair generation REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* Generate a secure ML-KEM CCA key pair                              */
/*--------------------------------------------------------------------*/

 /* expected results */
 ExpRc = '00000000'x
 ExpRs = '00000000'x

/*--------------------------------------------------------------------*/
/* Build skeleton token with key usage                                */
/*--------------------------------------------------------------------*/
 PKB_Rule_Count   = '00000003'x ;
 PKB_Rule_Array   = 'QSA-PAIR' ||,
                    'U-KEYENC' ||,
                    'U-DATENC'

 /* ML-KEM-1024 KVS */
 PKB_KVS    = '06'x   ||, /* Alg Id */
              '00'x   ||, /* clear key format */
              '1024'x ||, /* Alg param */
              '0000'x ||, /* clear key len */
              '0000'x     /* Reserved */

 call CSNDPKB

/*--------------------------------------------------------------------*/
/* Generate the ML-KEM key pair using the skeleton token from PKB     */
/*--------------------------------------------------------------------*/
 PKG_Rule_Array = 'MASTER  '
 PKG_Skeleton_Key        = PKB_Token;
 PKG_Skeleton_Key_length = PKB_Token_length;

 call CSNDPKG

 Exit
/*--------------------------------------------------------------------*/
/* PKA Key Token Build - used to create PKA key tokens.               */
/*                                                                    */
/* See the ICSF Application Programmer's Guide for more details.      */
/*--------------------------------------------------------------------*/
CSNDPKB:

/* initialize parameter list */
PKB_Rc           = 'FFFFFFFF'x ;
PKB_Rs           = 'FFFFFFFF'x ;
Exit_Length      = '00000000'x ;
Exit_Data        = '' ;
```

```
PKB_KVS_Length   = d2c(length(PKB_KVS),4) ;
PKB_UAD_Length   = '00000000'x ;
PKB_UAD          = ''
PKB_PrivName_Len = '00000000'x ;
PKB_PrivName     = ''
Reserved2_Length = '00000000'x ; Reserved2 = '' ;
Reserved3_Length = '00000000'x ; Reserved3 = '' ;
Reserved4_Length = '00000000'x ; Reserved4 = '' ;
Reserved5_Length = '00000000'x ; Reserved5 = '' ;
PKB_Token_Length = d2c(8000,4) ;
PKB_Token        = copies('00'x,8000) ;

/* call CSNDPKB */
address linkpgm 'CSNDPKB'                              ,
                'PKB_Rc'           'PKB_Rs'            ,
                'Exit_Length'      'Exit_Data'         ,
                'PKB_Rule_Count'   'PKB_Rule_Array'    ,
                'PKB_KVS_Length'   'PKB_KVS'           ,
                'PKB_PrivName_Len' 'PKB_PrivName'      ,
                'PKB_UAD_Length'   'PKB_UAD'           ,
                'Reserved2_Length' 'Reserved2'         ,
                'Reserved3_Length' 'Reserved3'         ,
                'Reserved4_Length' 'Reserved4'         ,
                'Reserved5_Length' 'Reserved5'         ,
                'PKB_Token_Length' 'PKB_Token'         ;

if (PKB_Rc \= ExpRc | PKB_Rs \= ExpRs) then
  say 'PKB failed: rc =' c2x(PKB_Rc) 'rs =' c2x(PKB_Rs) ;
else
 do ;
  say 'PKB successful: rc =' c2x(PKB_Rc) 'rs =' c2x(PKB_Rs) ;
  PKB_Token = substr(PKB_Token,1,c2d(PKB_Token_Length)) ;
 end

 return

/* ------------------------------------------------------------- */
/* PKA Key Generate  - Used to generate PKA key pairs.           */
/*                                                               */
/* See the ICSF Application Programmer's Guide for more details. */
/* ------------------------------------------------------------- */
CSNDPKG:

 /* initialize parameter list */
 PKG_rc           = 'FFFFFFFF'x ;
 PKG_rs           = 'FFFFFFFF'x ;
 PKG_Exit_length  = '00000000'x ;
 PKG_Exit_Data    = '' ;
 PKG_Rule_count   = d2c( length(PKG_Rule_Array)/8,4 )
 PKG_Token_length = '00001F40'x ;
 PKG_Token        = copies('00'x,c2d(PKG_token_length)) ;
 PKG_Regen_data   = ''
 PKG_Regen_Data_length = d2c( length(PKG_Regen_data),4 )
 PKG_Transport_Key_Id  = ''
```

```
/* call CSNDPKG */
address linkpgm 'CSNDPKG' ,
                'PKG_rc'                  'PKG_rs'          ,
                'PKG_Exit_length'         'PKG_Exit_Data'   ,
                'PKG_Rule_Count'          'PKG_Rule_Array'  ,
                'PKG_Regen_Data_length'   'PKG_Regen_Data'  ,
                'PKG_Skeleton_Key_length' 'PKG_Skeleton_Key' ,
                'PKG_Transport_Key_Id'                      ,
                'PKG_Token_length'        'PKG_Token'       ;

if (PKG_rc \= ExpRc | PKG_rs \= ExpRs) then
 say 'PKG failed: rc =' c2x(PKG_rc) 'rs =' c2x(PKG_rs)
else
 Do;
  say 'PKG  successful : rc =' c2x(PKG_rc) 'rs =' c2x(PKG_rs) ;
  PKG_Token = substr(PKG_Token,1,c2d(PKG_Token_length)) ;
 End;

return
```

## B.6 PKCS #11 CRYSTALS-Kyber key pair generation REXX sample

A PKCS #11 CRYSTALS-Kyber key pair generation REXX sample is shown in Example B-6.

*Example B-6 PKCS #11 CRYSTALS-Kyber key pair generation REXX sample*

```
/* Rexx */

Call TCSetup

/*--------------------------------------------------------------------*/
/* Generate a secure PKCS #11 Kyber key pair                          */
/*--------------------------------------------------------------------*/

/* expected results */
ExpRC = '00000000'x ;
ExpRS = '00000000'x ;

Call TCSETUP

GKP_Handle             = Left('QSAFE.TEST.TOKEN',44)

GKP_PrivKey_Attr_List = '0007'x||,
       CKA_CLASS      ||'0004'x|| CKO_PRIVATE_KEY         ||,
       CKA_KEY_TYPE   ||'0004'x|| CKK_IBM_KYBER           ||,
       CKA_TOKEN      ||'0001'x|| CK_TRUE                 ||,
       CKA_DERIVE     ||'0001'x|| CK_TRUE                 ||,
       CKA_DECRYPT    ||'0001'x|| CK_TRUE                 ||,
       CKA_UNWRAP     ||'0001'x|| CK_TRUE                 ||,
       CKA_IBM_SECURE ||'0001'x|| CK_TRUE

GKP_PubKey_Attr_List = '0007'x||,
       CKA_CLASS              ||'0004'x|| CKO_PUBLIC_KEY    ||,
       CKA_KEY_TYPE           ||'0004'x|| CKK_IBM_KYBER     ||,
       CKA_IBM_KYBER_MODE     ||'000D'x|| DER_OID_KYBER_1024_R2   ||,
       CKA_TOKEN              ||'0001'x|| CK_TRUE           ||,
       CKA_WRAP               ||'0001'x|| CK_TRUE           ||,
       CKA_DERIVE             ||'0001'x|| CK_TRUE           ||,
       CKA_ENCRYPT            ||'0001'x|| CK_TRUE

Call CSFPGKP;

Exit
/* ---------------------------------------------------------------- */
/* PKCS #11 Generate Key Pair                                       */
/* Use the PKCS #11 Generate Key Pair callable service to generate */
/* an RSA, DSA, Elliptic Curve, Diffie-Hellman, Dilithium (LI2) or */
/* Kyber key pair.                                                  */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ---------------------------------------------------------------- */
CSFPGKP:
 GKP_RC = 'FFFFFFFF'x
 GKP_RS = 'FFFFFFFF'x
```

```
 GKP_Exit_Length = '00000000'x
 GKP_Exit_Data = ''
 GKP_Rule_Count = '00000000'x
 GKP_Rule_Array = ''
 GKP_PubKey_Handle = copies(' ',44)
 GKP_PrivKey_Handle = copies(' ',44)

 GKP_PubKey_Attr_List_Length = D2C(Length(GKP_PubKey_Attr_List),4)
 GKP_PrivKey_Attr_List_Length = D2C(Length(GKP_PrivKey_Attr_List),4)

 address linkpgm 'CSFPGKP',
                 'GKP_RC' 'GKP_RS',
                 'GKP_Exit_Length' 'GKP_Exit_Data',
                 'GKP_Handle',
                 'GKP_Rule_Count' 'GKP_Rule_Array',
                 'GKP_PubKey_Attr_List_Length',
                 'GKP_PubKey_Attr_List',
                 'GKP_PubKey_Handle',
                 'GKP_PrivKey_Attr_List_Length',
                 'GKP_PrivKey_Attr_List',
                 'GKP_PrivKey_Handle'

   if (GKP_RC \= ExpRC | GKP_RS \= ExpRS) Then
     say 'GKP failed: rc =' c2x(GKP_rc) 'rs =' c2x(GKP_rs) ;
   else
     say 'GKP successful : rc =' c2x(GKP_rc) 'rs =' c2x(GKP_rs) ;

return;

/* ---------------------------------------------------------------- */
/*                                                                  */
/* ---------------------------------------------------------------- */
TCSetup:

DER_OID_KYBER_1024_R2 = '060B2B0601040102820B050404'X;

CKK_IBM_KYBER         = '80010024'X;

CKO_PUBLIC_KEY        = '00000002'X
CKO_PRIVATE_KEY       = '00000003'X

CKA_IBM_SECURE        = '80000006'X
CKA_KEY_TYPE          = '00000100'X
CKA_CLASS             = '00000000'X
CKA_TOKEN             = '00000001'X
CKA_IBM_KYBER_MODE    = '8000000E'X;
CKA_ENCRYPT           = '00000104'X;
CKA_DECRYPT           = '00000105'X;
CKA_WRAP              = '00000106'X;
CKA_UNWRAP            = '00000107'X;
CKA_DERIVE            = '0000010C'X;

CK_TRUE               = '01'x
CK_FALSE              = '00'x
```

```
Return

EXIT;
```

C

# Translating plain text into cipher text

The examples in this appendix are REXX executables that can be used to translate plain text into cipher text by using CCA and PKCS #11.

This appendix includes the following topics:

# C.1 CCA ciphertext translation REXX sample

A CCA ciphertext translation REXX sample is shown in Example C-1.

*Example C-1 CCA ciphertext translation from DES to AES REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* Translate existing ciphertext to an AES 256-bit key                */
/*--------------------------------------------------------------------*/

 /* expected results */
 ExpRc = '00000000'x
 ExpRs = '00000000'x

 /*--------------------------------------------------------------------*/
 /* Call CSNBCTT2 to translate the existing ciphertext to AES          */
 /*--------------------------------------------------------------------*/
 CTT2_Rule_Count     = '00000004'x
 CTT2_rule_array     = 'I-CBC   '||'O-CBC   '||'IKEY-DES'||'OKEY-AES';
 CTT2_cipher_text_in = 'E7861BBEEA363B3C40168B3174C15D31'x ;

 /* Pass either the tokens or key labels of the encryption keys.      */
 CTT2_key_ID_in   = left('DATAENC#CTT2#DES#CIPHER',64)
 CTT2_key_ID_out  = left('DATAENC#CTT2#AES256#CIPHER',64) ;

 Call CSNBCTT2

 exit
/*--------------------------------------------------------------------*/
/* CipherText Translate2                                              */
/*                                                                    */
/* This callable service deciphers encrypted data (ciphertext) under */
/* one cipher text translation key and reenciphers it under another  */
/* cipher text translation key without having the data appear in the */
/* clear outside the cryptographic coprocessor.                       */
/*                                                                    */
/* See the ICSF Application Programmer's Guide for more details.     */
/*--------------------------------------------------------------------*/
CSNBCTT2:

  CTT2_rc                  = 'FFFFFFFF'x ;
  CTT2_rs                  = 'FFFFFFFF'x ;
  CTT2_Exit_Len            = '00000000'x ;
  CTT2_Exit_Data           = '' ;
  CTT2_IV_in_len           = '00000008'X
  CTT2_IV_in               = '0000000000000000'X
  CTT2_cipher_text_in_len  = d2c(length(CTT2_cipher_text_in),4)
  CTT2_chaining_vector_len = '00000080'X
  CTT2_chaining_vector     = copies('00'x,128)
  CTT2_IV_out_len          = '00000010'X
  CTT2_IV_out              = '0000000000000000'X
  CTT2_rsv1_len            = '00000000'x
  CTT2_rsv1                = ''
```

```
CTT2_rsv2_len             = '00000000'x
CTT2_rsv2                 = ''
CTT2_key_ID_in_len        = '00000040'x
CTT2_key_ID_out_len       = '00000040'x
CTT2_cipher_text_out_len  = d2c(length(CTT2_cipher_text_in),4)
CTT2_cipher_text_out      = copies('00'x,c2d(CTT2_cipher_text_out_len))

address linkpgm 'CSNBCTT2'                                          ,
                'CTT2_rc'                  'CTT2_rs'                ,
                'CTT2_Exit_Len'            'CTT2_Exit_Data'         ,
                'CTT2_Rule_Count'          'CTT2_Rule_array'        ,
                'CTT2_key_ID_in_len'       'CTT2_key_ID_in'         ,
                'CTT2_IV_in_len'           'CTT2_IV_in'             ,
                'CTT2_cipher_text_in_len'  'CTT2_cipher_text_in',
                'CTT2_chaining_vector_len' 'CTT2_chaining_vector',
                'CTT2_key_ID_out_len'      'CTT2_key_ID_out'        ,
                'CTT2_IV_out_len'          'CTT2_IV_out'            ,
                'CTT2_cipher_text_out_len' 'CTT2_cipher_text_out',
                'CTT2_rsv1_len'            'CTT2_rsv1'              ,
                'CTT2_rsv2_len'            'CTT2_rsv2'              ;

if (CTT2_rc \= ExpRc | CTT2_rs \= ExpRs) then
 say 'CTT2 failed: rc=' c2x(CTT2_rc) 'rs =' c2x(CTT2_rs) ;
else
 say 'CTT2 successful: rc=' c2x(CTT2_rc) 'rs =' c2x(CTT2_rs) ;

return;
```

## C.2 PKCS #11 ciphertext translation REXX sample

A PKCS #11 ciphertext translation REXX sample is shown in Example C-2.

*Example C-2 PKCS #11 ciphertext translation from DES to AES REXX sample*

```
/* rexx */

/*-------------------------------------------------------------------*/
/* Translate existing ciphertext to an AES 256-bit key               */
/*-------------------------------------------------------------------*/

 /* expected results */
 ExpRC = '00000000'x ;
 ExpRS = '00000000'x ;

 SKR_Rule_Array = 'D-CBCPAD' || 'E-CBCPAD'

 /*-------------------------------------------------------------------*/
 /* Pass existing ciphertext and set IV according to the decryption   */
 /* key. For DES keys, IV length is 8.                                */
 /*-------------------------------------------------------------------*/
 SKR_dec_iv_length     = '00000008'x;
 SKR_dec_iv            = copies('00'x,c2d(SKR_dec_iv_length) )
 SKR_dec_text   =,
 '3AE0F4D65E911F061FED6FEB0CB84D6996A5623CADED94AEA3B8E2923F04E927'x ||,
 'DADFD96CCDDB5497442F6A75C82041AFE418D930AF4DE8B732A4D86C1D3F60EC'x ||,
 '530BB9336A042B2A398FE650B8E38D2451D2427B904ED7B1'x
 SKR_dec_text_length  = d2c(length(SKR_dec_text),4)

 /*-------------------------------------------------------------------*/
 /* Set encryption IV length to 16 for AES                            */
 /*-------------------------------------------------------------------*/
 SKR_enc_iv_length = '00000010'x
 SKR_enc_iv = copies('00'x,c2d(SKR_enc_iv_length))

 /* Secure DES3 handle */
 SKR_dec_handle = 'QSAFE.TEST.TOKEN                00000001Y'
 /* Secure AES 256 handle */
 SKR_enc_handle = 'QSAFE.TEST.TOKEN                00000002Y'

 call CSFPSKR

 exit
/* ---------------------------------------------------------------- */
/* PKCS #11 Secret Key Reencrypt                                    */
/*                                                                  */
/* Use the PKCS #11 Secret Key Reencrypt callable service to        */
/* decrypt data and then reencrypt the data using secure secret     */
/* keys.                                                            */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */
CSFPSKR:
 SKR_rc            = 'FFFFFFFF'x ;
```

```
SKR_rs            = 'FFFFFFFF'x ;
SKR_Exit_Length   = '00000000'x;
SKR_Exit_Data     = '';
SKR_Rule_Count    = '00000002'x;
SKR_chain_data_length = '00000000'x
SKR_chain_data        = '';
SKR_dec_text_id       = '00000000'x;
SKR_enc_text_length   = D2C(1000,4);
SKR_enc_text          = COPIES('00'x,C2D(SKR_enc_text_length,4));
SKR_enc_text_id       = '00000000'x;

 address linkpgm 'CSFPSKR'                              ,
                 'SKR_rc'                'SKR_rs'       ,
                 'SKR_Exit_Length'       'SKR_Exit_Data' ,
                 'SKR_Rule_Count'        'SKR_Rule_Array',
                 'SKR_dec_handle'        'SKR_enc_handle',
                 'SKR_dec_iv_length'     'SKR_dec_iv'    ,
                 'SKR_enc_iv_length'     'SKR_enc_iv'    ,
                 'SKR_chain_data_length' 'SKR_chain_data',
                 'SKR_dec_text_length'   'SKR_dec_text'  ,
                 'SKR_dec_text_id'                       ,
                 'SKR_enc_text_length'   'SKR_enc_text'  ,
                 'SKR_enc_text_id'                       ;

if (SKR_rc \= ExpRC | SKR_rs \= ExpRS) then
  say 'SKR failed: rc =' c2x(SKR_rc) 'rs =' c2x(SKR_rs)
else
  say 'SKR successful rc =' c2x(SKR_rc) 'rs =' c2x(SKR_rs)
return;
```

D

# Generating and verifying digital signatures

The examples in this appendix are REXX executables that can be used to generate and verify digital signatures using CCA and PKCS #11.

This appendix includes the following topics:

# D.1 CCA ML-DSA “pure” digital signature generation and verification REXX sample

A CCA ML-DSA “pure” digital signature generation and verification REXX sample is shown in Example D-1.

*Example D-1 CCA ML-DSA “pure” digital signature generation and verification REXX sample*

```
/* rexx */

/*--------------------------------------------------------------------*/
/* Pure ML-DSA Digital signature generation and verification         */
/*--------------------------------------------------------------------*/
/* expected results */
ExpRc = '00000000'x ;
ExpRs = '00000000'x ;

/* Message to Sign */
message =  'A9993E364706816ABA3E25717850C26C9CD0D89D'X ||,
           'A9993E364706816ABA3E25717850C26C9CD0D89D'X ||,
           'A9993E364706816ABA3E25717850C26C9CD0D89D'X;

/*--------------------------------------------------------------------*/
/* Call the CSNDDSG service passing the Pure ML-DSA private key.     */
/* With a Crypto Express8S CCA Coprocessor, the message to be        */
/* signed can be up to 15000 bytes.                                  */
/*--------------------------------------------------------------------*/
DSG_Rule_Array = 'CRDL-DSA' ||,
                 'MESSAGE ' ||,
                 'CRDLHASH'

/* Pure ('05'x) ML-DSA private key label  */
DSG_priv_key = left('MLDSA87.PURE.PRV.0001',64)
DSG_data = message

call CSNDDSG

/*--------------------------------------------------------------------*/
/* Call the CSNDDSG service passing the Pure ML-DSA public key.      */
/*--------------------------------------------------------------------*/
DSV_Data = DSG_data
DSV_Sig_Field = DSG_sig_field
DSV_Rule_Array = DSG_Rule_Array

/* Pure ('05'x) ML-DSA public key label  */
DSV_pub_key = left('MLDSA87.PURE.PUB.0001',64)

call CSNDDSV

exit
/* ---------------------------------------------------------------- */
/* Digital Signature Generate                                       */
/*                                                                  */
/* Use the Digital Signature Generate callable service to generate */
/* a digital signature using a PKA private key.                     */
```

```
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ---------------------------------------------------------------- */
CSNDDSG:

DSG_rc                = 'FFFFFFFF'x ;
DSG_rs                = 'FFFFFFFF'x ;
DSG_Exit_Length       = '00000000'x ;
DSG_Exit_Data         = '' ;
DSG_Data_length       = D2C( Length(DSG_Data),4 );
DSG_Sig_Field_Length  = '00001388'x ;
DSG_Sig_Bit_Length    = '00000000'x ;
DSG_Sig_Field         = copies('00'x,c2d(DSG_Sig_field_length))
DSG_rule_count        = d2c( length(DSG_rule_array)/8,4 )
DSG_priv_key_length   = d2c( length(DSG_priv_key),4 )

address linkpgm 'CSNDDSG' ,
                'DSG_rc' 'DSG_rs' ,
                'DSG_Exit_Length' 'DSG_Exit_Data' ,
                'DSG_Rule_Count' 'DSG_Rule_Array' ,
                'DSG_priv_key_length' 'DSG_priv_key' ,
                'DSG_data_length' 'DSG_data' ,
                'DSG_sig_field_length' ,
                'DSG_sig_bit_length' ,
                'DSG_sig_field' ;

DSG_sig_field = substr(DSG_sig_field,1,c2d(DSG_sig_field_length))

if (DSG_rc \= ExpRc | DSG_rs \= ExpRs) then
  say 'DSG: failed: rc =' c2x(DSG_rc) 'rs =' c2x(DSG_rs)
else
  say 'DSG successful : rc =' c2x(DSG_rc) 'rs =' c2x(DSG_rs) ;

return;
/* ---------------------------------------------------------------- */
/* Digital Signature Verify                                         */
/*                                                                  */
/* Use the Digital Signature Verify callable service to verify a    */
/* digital signature using a PKA public key.                        */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */
CSNDDSV:

DSV_rc               = 'FFFFFFFF'x ;
DSV_rs               = 'FFFFFFFF'x ;
DSV_Exit_Length      = '00000000'x ;
DSV_Exit_Data        = '' ;
DSV_Data_length      = D2C( Length(DSV_Data),4 );
DSV_Sig_Field_Length = d2c( length(DSV_sig_field),4 )
DSV_rule_count       = d2c( length(DSV_rule_array)/8,4 )
DSV_pub_key_length   = d2c( length(DSV_pub_key),4 )

address linkpgm 'CSNDDSV' ,
                'DSV_rc' 'DSV_rs' ,
```

```
                'DSV_Exit_Length' 'DSV_Exit_Data' ,
                'DSV_Rule_Count' 'DSV_Rule_Array' ,
                'DSV_pub_key_length' 'DSV_pub_key' ,
                'DSV_data_length' 'DSV_data' ,
                'DSV_sig_field_length' ,
                'DSV_sig_field' ;

if DSV_rc \= ExpRc | DSV_rs \= ExpRs then
  say 'DSV failed: rc =' c2x(DSV_rc) 'rs =' c2x(DSV_rs)
else
  say 'DSV successful : rc =' c2x(DSV_rc) 'rs =' c2x(DSV_rs) ;

return;
```

## D.2 CCA ML-DSA “pre-hash” digital signature generation and verification REXX sample

A CCA ML-DSA “pre-hash” digital signature generation and verification REXX sample is shown in Example D-2.

*Example D-2 CCA ML-DSA “pre-hash” digital signature generation and verification REXX sample*

```
/* rexx */

/*--------------------------------------------------------------------*/
/* Pre-hash ML-DSA Digital signature generation and verification     */
/*--------------------------------------------------------------------*/
/* expected results */
ExpRc = '00000000'x ;
ExpRs = '00000000'x ;

/* Message to Sign */
message =  'A9993E364706816ABA3E25717850C26C9CD0D89D'X ||,
           'A9993E364706816ABA3E25717850C26C9CD0D89D'X ||,
           'A9993E364706816ABA3E25717850C26C9CD0D89D'X;

/*--------------------------------------------------------------------*/
/* Call the CSNDDSG service passing the Pre-hash ML-DSA private key. */
/* With a Crypto Express8S CCA Coprocessor, the message to be        */
/* signed can be up to 15000 bytes.                                  */
/*--------------------------------------------------------------------*/
/* Message is to be hashed with SHA-512 and then signed */
DSG_Rule_Array = 'CRDL-DSA' ||,
                 'MESSAGE ' ||,
                 'SHA-512 '

/* Message has already been hashed and is a SHA-512 digest */
/*
  DSG_Rule_Array = 'CRDL-DSA' ||,
                   'HASH    ' ||,
                   'SHA-512 '
*/

/* Pre-hash ('07'x) ML-DSA private key label  */
DSG_priv_key = left('MLDSA87.PREHASH.PRV.0001',64)

DSG_data = message

call CSNDDSG

/*--------------------------------------------------------------------*/
/* Call the CSNDDSG service passing the Pre-hash ML-DSA public key.  */
/*--------------------------------------------------------------------*/
DSV_Data = DSG_data
DSV_Sig_Field = DSG_sig_field
DSV_Rule_Array = DSG_Rule_Array

/* Pre-hash ('07'x) ML-DSA public key label  */
DSV_pub_key = left('MLDSA87.PREHASH.PUB.0001',64)
```

```
call CSNDDSV

exit
/* ------------------------------------------------------------- */
/* Digital Signature Generate                                    */
/*                                                               */
/* Use the Digital Signature Generate callable service to generate */
/* a digital signature using a PKA private key.                  */
/*                                                               */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ------------------------------------------------------------- */
CSNDDSG:

DSG_rc                = 'FFFFFFFF'x ;
DSG_rs                = 'FFFFFFFF'x ;
DSG_Exit_Length       = '00000000'x ;
DSG_Exit_Data         = '' ;
DSG_Data_length       = D2C( Length(DSG_Data),4 );
DSG_Sig_Field_Length  = '00001388'x ;
DSG_Sig_Bit_Length    = '00000000'x ;
DSG_Sig_Field         = copies('00'x,c2d(DSG_Sig_field_length))
DSG_rule_count        = d2c( length(DSG_rule_array)/8,4 )
DSG_priv_key_length   = d2c( length(DSG_priv_key),4 )

address linkpgm 'CSNDDSG' ,
                'DSG_rc' 'DSG_rs' ,
                'DSG_Exit_Length' 'DSG_Exit_Data' ,
                'DSG_Rule_Count' 'DSG_Rule_Array' ,
                'DSG_priv_key_length' 'DSG_priv_key' ,
                'DSG_data_length' 'DSG_data' ,
                'DSG_sig_field_length' ,
                'DSG_sig_bit_length' ,
                'DSG_sig_field' ;

DSG_sig_field = substr(DSG_sig_field,1,c2d(DSG_sig_field_length))

if (DSG_rc \= ExpRc | DSG_rs \= ExpRs) then
  say 'DSG: failed: rc =' c2x(DSG_rc) 'rs =' c2x(DSG_rs)
else
  say 'DSG successful : rc =' c2x(DSG_rc) 'rs =' c2x(DSG_rs) ;

return;
/* ------------------------------------------------------------- */
/* Digital Signature Verify                                      */
/*                                                               */
/* Use the Digital Signature Verify callable service to verify a   */
/* digital signature using a PKA public key.                     */
/*                                                               */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ------------------------------------------------------------- */
CSNDDSV:

DSV_rc               = 'FFFFFFFF'x ;
DSV_rs               = 'FFFFFFFF'x ;
```

```
DSV_Exit_Length      = '00000000'x ;
DSV_Exit_Data        = '' ;
DSV_Data_length      = D2C( Length(DSV_Data),4 );
DSV_Sig_Field_Length = d2c( length(DSV_sig_field),4 )
DSV_rule_count       = d2c( length(DSV_rule_array)/8,4 )
DSV_pub_key_length   = d2c( length(DSV_pub_key),4 )

address linkpgm 'CSNDDSV' ,
                'DSV_rc' 'DSV_rs' ,
                'DSV_Exit_Length' 'DSV_Exit_Data' ,
                'DSV_Rule_Count' 'DSV_Rule_Array' ,
                'DSV_pub_key_length' 'DSV_pub_key' ,
                'DSV_data_length' 'DSV_data' ,
                'DSV_sig_field_length' ,
                'DSV_sig_field' ;

if DSV_rc \= ExpRc | DSV_rs \= ExpRs then
  say 'DSV failed: rc =' c2x(DSV_rc) 'rs =' c2x(DSV_rs)
else
  say 'DSV successful : rc =' c2x(DSV_rc) 'rs =' c2x(DSV_rs) ;

return;
```

## D.3 PKCS #11 CRYSTALS-Dilithium digital signature generation and verification REXX sample

A PKCS #11 CRYSTALS-Dilithium digital signature generation and verification REXX sample is shown in Example D-2.

*Example D-3 PKCS #11 CRYSTALS-Dilithium digital signature generation and verification REXX*

```
/* rexx */

/*--------------------------------------------------------------------*/
/* CRYSTALS-Dilithium Digital signature generation and verification */
/*--------------------------------------------------------------------*/

 /* expected results */
 ExpRC = '00000000'x ;
 ExpRS = '00000000'x ;

/*--------------------------------------------------------------------*/
/* Call the CSFPPKS service passing the CRYSTALS-Dilithium private  */
/* key handle to generate the digital signature.                     */
/*--------------------------------------------------------------------*/
 PKS_Rule_Array          = 'LI2     '
 PKS_Key_Handle          = 'QSAFE.TEST.TOKEN                00000003Y'
 PKS_Cipher_Value        = Copies('A',128)
 PKS_Cipher_Value_Length = D2C( Length(PKS_Cipher_Value),4 );
 PKS_Clear_Value_length  = D2C(4596,4);
 PKS_Clear_Value         = Copies('OO'x, C2D(PKS_Clear_Value_length) )

 Call CSFPPKS

/*--------------------------------------------------------------------*/
/* Call the CSFPPKV service passing the CRYSTALS-Dilithium public   */
/* key handle to verify the digital signature.                       */
/*--------------------------------------------------------------------*/
 PKV_Key_Handle          = 'QSAFE.TEST.TOKEN                00000002Y'

 Call CSFPPKV

 exit
/* ------------------------------------------------------------------ */
/* PKCS #11 Private Key Sign                                          */
/*                                                                    */
/* Used to sign data using an ECC, RSA, DSA, or CRYSTALS-Dilithium */
/* private key.                                                       */
/* ------------------------------------------------------------------ */
CSFPPKS:

 PKS_RC              = 'FFFFFFFF'x ;
 PKS_RS              = 'FFFFFFFF'x ;
 PKS_Exit_Length     = '00000000'x ;
 PKS_Exit_Data       = '' ;

 PKS_Rule_Count = d2c( length(PKS_Rule_Array)/8,4 )
```

```
 address linkpgm 'CSFPPKS'                 ,
                 'PKS_rc'                  ,
                 'PKS_rs'                  ,
                 'PKS_Exit_Length'         ,
                 'PKS_Exit_Data'           ,
                 'PKS_Rule_Count'          ,
                 'PKS_Rule_Array'          ,
                 'PKS_Cipher_Value_Length' ,
                 'PKS_Cipher_Value'        ,
                 'PKS_Key_Handle'          ,
                 'PKS_Clear_Value_Length'  ,
                 'PKS_Clear_Value'         ;

 PKS_Clear_value = ,
    substr(PKS_clear_value,1,c2d(PKS_Clear_value_length))

 if (PKS_RC \= ExpRC | PKS_RS \= ExpRS) Then
    say 'PKS Failed : rc =' c2x(PKS_RC) 'rs =' c2x(PKS_RS) ;
 else
    say 'PKS Successful : rc =' c2x(PKS_RC) 'rs =' c2x(PKS_RS) ;
return;
/* ---------------------------------------------------------------- */
/* PKCS #11 Public Key Verify                                       */
/*                                                                  */
/* Used to verify a signature using an ECC, RSA, DSA, or            */
/* CRYSTALS-Dilithium public key.                                   */
/* ---------------------------------------------------------------- */
CSFPPKV:

 PKV_RC              = 'FFFFFFFF'x ;
 PKV_RS              = 'FFFFFFFF'x ;
 PKV_Exit_Length     = '00000000'x ;
 PKV_Exit_Data       = '';
 PKV_Cipher_Value_length = PKS_Cipher_Value_length
 PKV_Cipher_Value        = PKS_Cipher_Value
 PKV_Clear_Value         = PKS_Clear_Value
 PKV_Clear_Value_length  = PKS_Clear_Value_length
 PKV_Rule_Array          = PKS_Rule_Array
 PKV_Rule_Count      = d2c( length(PKV_rule_Array)/8,4 )


 address linkpgm 'CSFPPKV'             ,
                 'PKV_RC'              ,
                 'PKV_RS'              ,
                 'PKV_Exit_Length'     ,
                 'PKV_Exit_Data'       ,
                 'PKV_Rule_Count'      ,
                 'PKV_Rule_Array'      ,
                 'PKV_Clear_Value_Length' ,
                 'PKV_Clear_Value'      ,
                 'PKV_Key_Handle'       ,
                 'PKV_Cipher_Value_length' ,
                 'PKV_Cipher_Value'     ;
```

```
 PKV_Cipher_value = ,
    substr(pkv_cipher_value,1,c2d(PKV_Cipher_value_length))

 if (PKV_RC \= ExpRC | PKV_RS \= ExpRS) Then
   say 'PKV Failed : rc =' c2x(PKV_RC) 'rs =' c2x(PKV_RS) ;
 else
   say 'PKV successful : rc =' c2x(PKV_RC) 'rs =' c2x(PKV_RS) ;

return;
```

# D.4 ACSP agile UDF – Crystals-Dilithium digital signature generate and verify

The UDF capability of the ACSP server as described in “Considering cryptographic agility” on page 65, allows to centralize cryptographic operations in the ACSP server and hence, make these available across the enterprise.

## Agile UDF sample code and execution

The ACSP solution has a UDF SDK that includes an agile UDF sample java code that may generate or delete keys, generate a digital signature and verify the digital signature (see Example D-4). All functions can be performed with either an ECC key, a RSA key, or a CRYSTALS-Dilithium key.

*Example D-4 Agile UDF commands*

```
del <use ref> - delete generated key
use <use ref> - set use key for sign/verify
gen <use ref> - generate/store key for use value
check - check and list keys for use values
prop copy <from> <to> - copy use set <from> <to>
prop init - initiate properties with defaults
prop list - list all properties
prop del <name> - delete single property
prop get <name> - get single property
prop set <name> <val> - set property value
sign - sign with current key (call before verify)
ver - verify signature with current key
x,q - exit
```

The commands are used in the Agile UDF test client demo called *agileudftest* to both set the properties and to execute the cryptographic operations. In a real life scenario these will be separated to support the policy updated made by security operators and the cryptographic usage, sign and verify, will be called by the consuming application.

Hence, the security operator will use the property commands to set policy properties, the check command to verify that the keys are available in the keystore. If any of the keys are missing then these may be generated gen command. By using the check command the security operator may verify that the keys are present (see Example D-5)

*Example D-5 .Agile check command*

```
check
List keys referenced by use values - count(3):
dsa.key = GENERATE.UKOOQ.ECSC.DSASIGN.AGILE : key present.
qsa.key = GENERATE.UKOOQ.ECSC.QSASIGN.AGILE : key present.
rsa.key = GENERATE.UKOOQ.ECSC.RSASIGN.AGILE : key present.
```

Listing the properties informs which of the three algorithms are currently defined in the policy. In this context we will use the CRYSTALS-Dilithium key, *GENERATE.UKO0Q.ECSC.QSASIGN.AGILE*, to generate the digital signature.

Enter the sign command and when prompted enter the text to be signed (see Example D-6).

*Example D-6 Agile sign command*

```
Enter Agile command or type 'help' for command list:
sign
Enter text to sign: The quick brown fox jumps over the lazy dog
2024-11-19 08:31:24,742 [INFO ] AZH20317I Connecting to host
[some.acsp.server:9901] using transport [tls] for service [cca]
Signing data with key: qsa
Signature is:
```


The output of Example D-6 is a character representation of the digital signature and the use data information, meaning that one byte output yields two characters.

Changing the use property to dsa, rsa will perform digital signature generate with the ECC, RSA key.

For Comparison setting the use property to dsa end signing the same text yields a much smaller digital signature and use property (see Example D-7).

*Example D-7 Agile use command*

```
use dsa
Getting property key: use.values
  value of property key use.values is 'dsa,rsa,qsa'
Using dsa encryption
Enter Agile command or type 'help' for command list:
sign
Enter text to sign: The quick brown fox jumps over the lazy dog
Signing data with key: dsa
Signature is:

5A154E23180331841E4CFF5CCC2E2BA026D443A3CCA5F37F48C5542C352B37AE
114B802676E57EEC8BD5769A9B044528723E105CE544377C32A8F1B8E3007286
```

E

# Creating a quantum-safe key exchange

The examples in this appendix are REXX executables that can be used to create a quantum-safe key exchange using CCA and PKCS #11.

This appendix includes the following topics:

# E.1 CCA hybrid quantum-safe key exchange scheme REXX sample

A CCA hybrid quantum-safe key exchange scheme REXX sample is shown in Example E-1.

*Example E-1 CCA hybrid quantum-safe key exchange scheme REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* CCA Hybrid Quantum-safe Key exchange scheme                        */
/*--------------------------------------------------------------------*/
/* PKE will require ACP '0083'x                                       */
/* EDH will require ACP '035D'x                                       */
/*--------------------------------------------------------------------*/

CALL INITIALIZE

/* expected results */
Exp_rc = '00000000'x ;
Exp_rs = '00000000'x

/* global parameters */
exit_data_length = '00000000'x
exit_data = ''

/* PKB parameters */
private_name    = ''
user_assoc_data = ''

/* PKE parameters */
PKE_rule_array = 'ZERO-PAD'
PKE_keyvalue       = ''
sym_key_identifier = ''

/* KYT2 parameters */
kek_identifier = ''

/*---------------------*/
/* Create ALICE's keys */
/*---------------------*/

Say "Generating Alice's ML-KEM key pair..."

/*-------------------------------------------------------------*/
/* Build ML-KEM skeleton token with U-DATENC key usage flag  */
/*-------------------------------------------------------------*/
PKB_rule_array = 'QSA-PAIR'||'U-DATENC'
kvs            = '06'x ||,   /* algorithm identifier      */
                 '00'x ||,   /* clear key format skeleton */
                 '1024'x ||, /* algorithm parameter       */
                 '0000'x ||, /* clear key length          */
                 '0000'x     /* reserved                  */
CALL CSNDPKB
```

```
/*-------------------------------------------------------------*/
/* Generate ML-KEM key pair using built skeleton token       */
/*-------------------------------------------------------------*/
PKG_rule_array = 'master  '
CALL CSNDPKG

ALICE_MLKEM_pvt = PKG_token

/*-------------------------------------------------------------*/
/* Extract ML-KEM public key from ML-KEM private key token   */
/*-------------------------------------------------------------*/
PKX_source_key  = PKG_token
CALL CSNDPKX

ALICE_MLKEM_publ = PKX_token

/*-------------------------------------------------------------*/
/* Build ECC skeleton token with KEY-MGMT key usage flag     */
/*-------------------------------------------------------------*/
Say "Generating Alice's ECC key pair..."
PKB_rule_array = 'ECC-PAIR'||'KEY-MGMT'
kvs            = '00'x ||,   /* Prime curve    */
                 '00'x ||,   /* reserved       */
                 '0180'x ||, /* 384 bits       */
                 '0000'x ||, /* pvt key length */
                 '0000'x     /* pub key length */
CALL CSNDPKB

/*-------------------------------------------------------------*/
/* Generate ECC key pair using built skeleton token          */
/*-------------------------------------------------------------*/
PKG_rule_array = 'master  '
CALL CSNDPKG

ALICE_ECC_pvt  = PKG_token

/*-------------------------------------------------------------*/
/* Extract ECC public key from ECC private key token         */
/*-------------------------------------------------------------*/
PKX_source_key     = PKG_token
CALL CSNDPKX

ALICE_ECC_publ = PKX_token

/*-------------------*/
/* Create BOB's keys */
/*-------------------*/

/*-------------------------------------------------------------*/
/* Build ECC skeleton token with KEY-MGMT key usage flag     */
/*-------------------------------------------------------------*/
Say "Generating Bob's ECC key pair..."

PKB_rule_array = 'ECC-PAIR'||'KEY-MGMT'
kvs            = '00'x ||,   /* Prime curve    */
```

```
                '00'x ||,   /* reserved       */
                '0180'x ||, /* 384 bits       */
                '0000'x ||, /* pvt key length */
                '0000'x     /* pub key length */
CALL CSNDPKB

/*-------------------------------------------------------------*/
/* Generate ECC key pair using built skeleton token          */
/*-------------------------------------------------------------*/
PKG_rule_array = 'master  '
CALL CSNDPKG

BOB_ECC_pvt    = PKG_token

/*-------------------------------------------------------------*/
/* Extract ECC public key from ECC private key token         */
/*-------------------------------------------------------------*/
PKX_source_key     = PKG_token
CALL CSNDPKX

BOB_ECC_publ   = PKG_token

/*-------------------------------------------------------------*/
/* BOB creates the shared-key derivation input               */
/*-------------------------------------------------------------*/
PKE_rule_array = 'ZERO-PAD'||'RANDOM  ' || 'AES-ENC '
PKE_keyvalue       = '01010101010101010202020202020202'x||,
                     '00000000000000000000000000000000'x
sym_key_identifier = BOB_AES_CIPHER_key_token
public_key_identifier = ALICE_MLKEM_publ
CALL CSNDPKE

/*-------------------------------------------------------------*/
/* BOB completes the shared-key derivation                   */
/*-------------------------------------------------------------*/
MLKEM_enciphered_PKE_keyvalue = enciphered_PKE_keyvalue
sym_enciphered_PKE_keyvalue   = PKE_keyvalue

EDH_rule_array = 'DERIVO1 '||'KEY-AES '||'QSA-ECDH'||'IHKEYAES'
private_key_identifier = BOB_ECC_pvt
private_kek_identifier = ''
public_key_identifier  = ALICE_ECC_publ
hybrid_key_identifier  = BOB_AES_CIPHER_key_token
party_identifier       = 'Party#Identifier'
key_bit_length         = d2c(192,4)
initialization_vector  = '01010101010101010202020202020202'x
hybrid_ciphertext      = sym_enciphered_PKE_keyvalue
output_kek_identifier  = ''
output_key_identifier  = AES_CIPHER_skeleton
CALL CSNDEDH

/*-------------------------------------------------------------*/
/* A Key check value (KCV) is computed over BOBs shared-key  */
/*-------------------------------------------------------------*/
KYT2_rule_array = 'AES     '||'GENERATE'||'CMACZERO' ;
```

```
key_identifier  = output_key_identifier
CALL CSNBKYT2
KYT2_kcv_BOB = KYT2_kcv

/*-------------------------------------------------------------*/
/* Alice completes the shared-key derivation                   */
/*-------------------------------------------------------------*/
EDH_rule_array = 'DERIVO1 '||'KEY-AES '||'QSA-ECDH'||'IHKEYKYB'
private_key_identifier = ALICE_ECC_pvt
private_kek_identifier = ''
public_key_identifier  = BOB_ECC_publ
hybrid_key_identifier  = ALICE_MLKEM_pvt
party_identifier       = 'Party#Identifier'
key_bit_length         = d2c(192,4)
initialization_vector  = ''
hybrid_ciphertext      = MLKEM_enciphered_PKE_keyvalue
output_kek_identifier  = ''
output_key_identifier  = AES_CIPHER_skeleton
CALL CSNDEDH
/*-------------------------------------------------------------*/
/* A Key check value (KCV) is computed over Alice's            */
/* shared-key                                                  */
/*-------------------------------------------------------------*/
key_identifier  = output_key_identifier
CALL CSNBKYT2
KYT2_kcv_ALICE = KYT2_kcv

/*-------------------------------------------------------------*/
/* Verify that both Alice and Bobs shared-keys are identical */
/*-------------------------------------------------------------*/
IF KYT2_kcv_ALICE = KYT2_kcv_BOB THEN SAY 'TESTCASE SUCCESSFUL'

Exit;
/*-------------------------------------------------------------------*/
/* PKA Key Token Build - used to create PKA key tokens.              */
/*                                                                   */
/* See the ICSF Application Programmer's Guide for more details.     */
/*-------------------------------------------------------------------*/
CSNDPKB:

PKB_rc                 = 'FFFFFFFF'x
PKB_rs                 = 'FFFFFFFF'x
exit_data_length       = '00000000'x
exit_data              = ''
PKB_rule_count         = d2c(length(PKB_rule_array)/8,4)
kvs_length             = d2c(length(kvs),4)
private_name_length    = d2c(length(private_name),4)
user_assoc_data_length = d2c(length(user_assoc_data),4)
key_deriv_data_length  = '00000000'x  /* valid only with ECC-VER1 */
key_deriv_data         = ''
reserved_field3_length = '00000000'x
reserved_field3        = ''
reserved_field4_length = '00000000'x
reserved_field4        = ''
reserved_field5_length = '00000000'x
```

```
reserved_field5        = ''
PKB_token_length       = d2c(6500,4)  /* max */
PKB_token              = d2c(0,6500)

ADDRESS LINKPGM 'CSNDPKB' ,
                'PKB_rc'                'PKB_rs' ,
                'exit_data_length'      'exit_data' ,
                'PKB_rule_count'        'PKB_rule_array' ,
                'kvs_length'            'kvs' ,
                'private_name_length'   'private_name' ,
                'user_assoc_data_length' 'user_assoc_data' ,
                'key_deriv_data_length'  'key_deriv_data' ,
                'reserved_field3_length' 'reserved_field3' ,
                'reserved_field4_length' 'reserved_field4' ,
                'reserved_field5_length' 'reserved_field5' ,
                'PKB_token_length'      'PKB_token'

IF PKB_rc \= Exp_rc | PKB_rs \= Exp_rs THEN
  SAY 'PKB FAILED rc =' c2x(PKB_rc) 'rs =' c2x(PKB_rs)
ELSE
 DO
  SAY 'PKB successful: rc =' c2x(PKB_rc) 'rs =' c2x(PKB_rs)
  PKB_token = SUBSTR(PKB_token,1,c2d(PKB_token_length))
 END

SAY
RETURN

/* ---------------------------------------------------------------- */
/* PKA Key Generate  - Used to generate PKA key pairs.              */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */
CSNDPKG:

PKG_rc                  = 'FFFFFFFF'x ;
PKG_rs                  = 'FFFFFFFF'x ;
PKG_rule_count          = d2c(length(PKG_rule_array)/8,4) ;
regeneration_data_length = '00000000'x ;
regeneration_data       = '' ;
skeleton_key_id_length  = PKB_token_length ;
skeleton_key_id         = PKB_token ;
transport_key_id        = d2c(0,64) ;
PKG_token_length        = d2c(6500,4) ;
PKG_token               = copies('00'x,6500) ;

ADDRESS LINKPGM 'CSNDPKG' ,
                'PKG_rc'                  'PKG_rs' ,
                'exit_data_length'        'exit_data' ,
                'PKG_rule_count'          'PKG_rule_array' ,
                'regeneration_data_length' 'regeneration_data' ,
                'skeleton_key_id_length'  'skeleton_key_id' ,
                'transport_key_id' ,
                'PKG_token_length'        'PKG_token'
```

```
IF PKG_rc \= Exp_rc | PKG_rs \= Exp_rs THEN
  SAY 'PKG FAILED rc =' c2x(PKG_rc) 'rs =' c2x(PKG_rs)
ELSE
 DO
  SAY 'PKG successful: rc =' c2x(PKG_rc) 'rs =' c2x(PKG_rs)
  PKG_token = SUBSTR(PKG_token,1,c2d(PKG_token_length))
 END

SAY
RETURN

/*-------------------------------------------------------------------*/
/* PKA Public Key Extract                                            */
/*                                                                   */
/* Extracts a PKA public key token from a PKA internal (operational)*/
/* or external (importable) private key token.                       */
/*                                                                   */
/* See the ICSF Application Programmer's Guide for more details.     */
/*-------------------------------------------------------------------*/
CSNDPKX:

PKX_rc               = 'FFFFFFFF'x ;
PKX_rs               = 'FFFFFFFF'x ;
PKX_rule_array_count = '00000000'x ;
PKX_rule_array       = '' ;
PKX_source_key_length    = d2c(length(PKX_source_key),4) ;
PKX_token_length     = d2c(6500,4) ;
PKX_token            = copies('00'x,6500) ;

ADDRESS LINKPGM 'CSNDPKX' ,
                'PKX_rc' ,
                'PKX_rs' ,
                'exit_data_length' ,
                'exit_data' ,
                'PKX_rule_array_count' ,
                'PKX_rule_array' ,
                'PKX_source_key_length' ,
                'PKX_source_key' ,
                'PKX_token_length' ,
                'PKX_token'

IF PKX_rc /= Exp_rc | PKX_rs /= Exp_rs THEN
 DO ;
  SAY 'PKX FAILED rc =' c2x(PKX_rc) 'rs =' c2x(PKX_rs)
 END ;
ELSE
 DO ;
   SAY 'PKX successful: rc =' c2x(PKX_rc) 'rs =' c2x(PKX_rs)
   PKX_token = ,
     SUBSTR(PKX_token,1,c2d(PKX_token_length))
 END
SAY
RETURN

/* ---------------------------------------------------------------- */
```

```
/* PKA Encrypt                                                    */
/*                                                                */
/* Creates and encrypts derivation input                          */
/*                                                                */
/* See the ICSF Application Programmer's Guide for more details.  */
/* -------------------------------------------------------------- */
CSNDPKE:

PKE_rc = 'FFFFFFFF'x
PKE_rs = 'FFFFFFFF'x
exit_data_length = '00000000'x
exit_data = ''
PKE_rule_array_count = d2c(length(PKE_rule_array)/8,4)
PKE_keyvalue_length      = d2c(length(PKE_keyvalue),4)
sym_key_identifier_length = d2c(length(sym_key_identifier),4)
public_key_identifier_length = d2c(length(public_key_identifier),4)
enciphered_PKE_keyvalue_length = d2c(1568,4)
enciphered_PKE_keyvalue = d2c(0,1568)

say 'enciphered_PKE_keyvalue_length' c2x(enciphered_PKE_keyvalue_length)
say 'public_key_identifier_length' c2x(public_key_identifier_length)
say 'sym_key_identifier_length' c2x(sym_key_identifier_length)
say 'PKE_keyvalue_length' c2x(PKE_keyvalue_length)

ADDRESS LINKPGM 'CSNDPKE' ,
                'PKE_rc' ,
                'PKE_rs' ,
                'exit_data_length' ,
                'exit_data' ,
                'PKE_rule_array_count' ,
                'PKE_rule_array' ,
                'PKE_keyvalue_length' ,
                'PKE_keyvalue' ,
                'sym_key_identifier_length' ,
                'sym_key_identifier' ,
                'public_key_identifier_length' ,
                'public_key_identifier' ,
                'enciphered_PKE_keyvalue_length' ,
                'enciphered_PKE_keyvalue' ;

IF PKE_rc /= Exp_rc | PKE_rs /= Exp_rs THEN
  SAY 'PKE FAILED rc=' c2x(PKE_rc) 'rs =' c2x(PKE_rs) ;
ELSE
 DO
  enciphered_PKE_keyvalue = ,
     substr(enciphered_PKE_keyvalue,1,c2d(enciphered_PKE_keyvalue_length))
  SAY 'PKE successful rc=' c2x(PKE_rc) 'rs =' c2x(PKE_rs) ;
 END
SAY
RETURN

/* -------------------------------------------------------------- */
/* ECC Diffie-Hellman                                             */
/*                                                                */
/* Generates Z value from D-H process. Derives the shared-key using */
```

```
/* Z and rand-32 from PKE.                                           */
/*                                                                   */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ------------------------------------------------------------------*/
CSNDEDH:

EDH_rc = 'FFFFFFFF'x
EDH_rs = 'FFFFFFFF'x
exit_data_length = '00000000'x
exit_data = ''
EDH_rule_array_count = d2c(length(EDH_rule_array)/8,4)
private_key_identifier_length = d2c(length(private_key_identifier),4)
private_kek_identifier_length = d2c(length(private_kek_identifier),4)
public_key_identifier_length  = d2c(length(public_key_identifier),4)
hybrid_key_identifier_length  = d2c(length(hybrid_key_identifier),4)
party_identifier_length       = d2c(length(party_identifier),4)
initialization_vector_length  = d2c(length(initialization_vector),4)
hybrid_ciphertext_length      = d2c(length(hybrid_ciphertext),4)
reserved3_length = '00000000'x
reserved3 = ''
reserved4_length = '00000000'x
reserved4 = ''
reserved5_length = '00000000'x
reserved5 = ''
output_kek_identifier_length  = d2c(length(output_kek_identifier),4)
output_key_identifier_length  = d2c(900,4)
output_key_identifier         = left(output_key_identifier,900)

ADDRESS LINKPGM 'CSNDEDH' ,
                'EDH_rc' ,
                'EDH_rs' ,
                'exit_data_length' ,
                'exit_data' ,
                'EDH_rule_array_count' ,
                'EDH_rule_array' ,
                'private_key_identifier_length' ,
                'private_key_identifier' ,
                'private_kek_identifier_length' ,
                'private_kek_identifier' ,
                'public_key_identifier_length' ,
                'public_key_identifier' ,
                'hybrid_key_identifier_length' ,
                'hybrid_key_identifier' ,
                'party_identifier_length' ,
                'party_identifier' ,
                'key_bit_length' ,
                'initialization_vector_length' ,
                'initialization_vector' ,
                'hybrid_ciphertext_length' ,
                'hybrid_ciphertext' ,
                'reserved3_length' ,
                'reserved3' ,
                'reserved4_length' ,
                'reserved4' ,
                'reserved5_length' ,
```

```
                'reserved5' ,
                'output_kek_identifier_length' ,
                'output_kek_identifier' ,
                'output_key_identifier_length' ,
                'output_key_identifier' ;

IF EDH_rc /= Exp_rc | EDH_rs /= Exp_rs THEN
  SAY 'EDH FAILED rc =' c2x(EDH_rc) 'rs =' c2x(EDH_rs)
ELSE
 DO
  SAY 'EDH successful: rc =' c2x(EDH_rc) 'rs =' c2x(EDH_rs)
  output_key_identifier = ,
     substr(output_key_identifier,1,c2d(output_key_identifier_length))
 END
SAY
RETURN

/*--------------------------------------------------------------------*/
/* Key Test2                                                          */
/*                                                                    */
/* Generate or verify a secure, cryptographic verification pattern    */
/* (also referred to as a key check value) for AES, DES and HMAC      */
/* keys.                                                              */
/*--------------------------------------------------------------------*/
CSNBKYT2:

KYT2_rc = 'FFFFFFFF'x ;
KYT2_rs = 'FFFFFFFF'x ;
KYT2_rule_array_count = d2c(length(KYT2_rule_array)/8,4) ;
key_identifier_length = d2c(length(key_identifier),4) ;
kek_identifier_length = d2c(length(kek_identifier),4) ;
reserved_length = d2c(0,4) ;
reserved        = '' ;
KYT2_kcv_length      = d2c(8,4) ;
KYT2_kcv             = d2c(0,c2d(KYT2_kcv_length)) ;

ADDRESS LINKPGM 'CSNBKYT2'                                  ,
                'KYT2_rc'               'KYT2_rs'           ,
                'exit_data_length'      'exit_data'         ,
                'KYT2_rule_array_count' 'KYT2_rule_array',
                'key_identifier_length' 'key_identifier' ,
                'kek_identifier_length' 'kek_identifier' ,
                'reserved_length'       'reserved'          ,
                'KYT2_kcv_length'       'KYT2_kcv'          ;

IF KYT2_rc /= Exp_rc | KYT2_rs /= Exp_rs THEN
 SAY 'KYT2 failed: rc =' c2x(KYT2_rc) 'rs =' c2x(KYT2_rs) ;
ELSE
 SAY 'KYT2_kcv:' c2x(KYT2_kcv) ;

RETURN;

/* ---------------------------------------------------------------- */
INITIALIZE:
```

```
BOB_AES_CIPHER_key_token = ,
'010000DA05000000003012058C870E9D3194F0000000000000000020200000100'x||,
'007440001A0002400002000102C000000003E00000000505414553244349504 8'x||,
'4552233139324249544034332E32432E313620202020202020202020202020'x||,
'2020202020202020202020202020202020202020202020202020C1C5E240C3C9D7C8'x||,
'C5D940F1F9F2C2C9E340F4F36DF2C36DF1F6E2219F0ED611C48D338927427F2D'x||,
'141BB9EA9B5B198C98E141BFDD0FFC7B403B8F68620E8744CC92E321354C0707'x||,
'A2CC1E32C835563FDB749C76FF3A0CB32DB0667FA1CA77E8F1B1'x

/* symmetric key skeletons */
AES_CIPHER_SKELETON = ,
'0100003805000000000000000000000000000000000000000000000020200000100'x||,
'001A000000000000000002000102C000000003E00000000000'x

RETURN
```

## E.2 CCA ML-KEM key encapsulation REXX sample

A CCA ML-KEM key encapsulation REXX sample is shown in Example E-2.

*Example E-2 CCA ML-KEM key encapsulation REXX sample*

```
/* rexx */

Exp_rc = '00000000'x
Exp_rs = '00000000'x

/* symmetric key skeletons */
AES_CIPHER_SKELETON = ,
'0100003805000000000000000000000000000000000000000000000020200000100'x||,
'001A0000000000000002000102C000000003E00000000000'x

/* Randomly generate and encrypt a 32b value and return it in an
   encrypted CCA AES key token                                    */
PKE_rule_array = 'ZERO-PAD' ||,
                 'RANDOM  ' ||,
                 'AES-KB  '

/* AES 16b Initialization Vector (IV) left justified in the buffer  */
PKE_keyvalue       = left('01010101010101010202020202020202'x||,
                          '00000000000000000000000000000000'x,256)

/* AES CIPHER key skeleton used to contain the generated key */
PKE_sym_key_identifier = AES_CIPHER_SKELETON

/* ML-KEM Public key label */
PKE_public_key_identifier = left('MLKEM.1024.PUB.0001',64)

call CSNDPKE

say 'PKE_keyvalue' c2x(PKE_keyvalue)

/* Return the 32b value in a CCA AES key token      */
PKD_Rule_Array = 'ZERO-PAD' ||,
                 'AES-KB  '

/* ML-KEM Private key label */
PKD_KeyIdentifier = left('MLKEM.1024.PRV.0001',64)

/* ML-KEM Encrypted value from PKE */
PKD_EncKeyValue_length     = PKE_EncKeyvalue_length
PKD_EncKeyValue            = PKE_EncKeyvalue

/* AES CIPHER key skeleton used to contain the decrypted key */
PKD_sym_key_identifier     = AES_CIPHER_SKELETON ;

call CSNDPKD

say 'PKD_target_Keyvalue' c2x(PKD_target_Keyvalue)

exit
```

```
/* ---------------------------------------------------------------- */
/* PKA Encrypt                                                      */
/*                                                                  */
/* Generates and encrypts a random 32-byte value                    */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */
CSNDPKE:

PKE_rc = 'FFFFFFFF'x
PKE_rs = 'FFFFFFFF'x
exit_data_length = '00000000'x
exit_data = ''
PKE_rule_array_count = d2c(length(PKE_rule_array)/8,4)
PKE_keyvalue_length      = d2c(length(PKE_keyvalue),4)
PKE_sym_key_identifier_length = d2c(length(PKE_sym_key_identifier),4)
PKE_public_key_identifier_length = d2c(length(PKE_public_key_identifier),4)
PKE_EncKeyvalue_length = d2c(1568,4)
PKE_EncKeyvalue = d2c(0,1568)

ADDRESS LINKPGM 'CSNDPKE' ,
                'PKE_rc' ,
                'PKE_rs' ,
                'exit_data_length' ,
                'exit_data' ,
                'PKE_rule_array_count' ,
                'PKE_rule_array' ,
                'PKE_keyvalue_length' ,
                'PKE_keyvalue' ,
                'PKE_sym_key_identifier_length' ,
                'PKE_sym_key_identifier' ,
                'PKE_public_key_identifier_length' ,
                'PKE_public_key_identifier' ,
                'PKE_EncKeyvalue_length' ,
                'PKE_EncKeyvalue' ;

 IF PKE_rc /= Exp_rc | PKE_rs /= Exp_rs THEN
  SAY 'PKE FAILED rc=' c2x(PKE_rc) 'rs =' c2x(PKE_rs) ;
ELSE
 DO
  PKE_EncKeyvalue = ,
     substr(PKE_EncKeyvalue,1,c2d(PKE_EncKeyvalue_length))
  PKE_keyvalue =,
     substr(PKE_keyvalue,1,c2d(PKE_keyvalue_length))
  SAY 'PKE successful'
 END
SAY
RETURN

/* ---------------------------------------------------------------- */
/* PKA Decrypt                                                      */
/*                                                                  */
/* Decrypts the 32-byte value                                       */
/*                                                                  */
```

```
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */

CSNDPKD:

 PKD_rc                     = 'FFFFFFFF'x ;
 PKD_rs                     = 'FFFFFFFF'x ;
 PKD_Exit_length            = '00000000'x ;
 PKD_Exit_Data              = '' ;
 PKD_Rule_Count             =  d2c(length(PKD_Rule_Array)/8,4)
 PKD_sym_key_id_length      = d2c(length(PKD_sym_key_identifier),4)
 PKD_KeyIdentifier_length   = d2c(length(PKD_KeyIdentifier),4)
 PKD_target_Keyvalue        = copies('00'x,256) ;
 PKD_target_Keyvalue_length = d2c(length(PKD_target_Keyvalue),4)

  address linkpgm 'CSNDPKD'                                    ,
                 'PKD_rc'                     'PKD_rs'          ,
                 'PKD_Exit_length'            'PKD_Exit_Data'      ,
                 'PKD_Rule_Count'             'PKD_Rule_Array'     ,
                 'PKD_EncKeyValue_length'     'PKD_EncKeyValue'  ,
                 'PKD_sym_key_id_length'      'PKD_sym_key_identifier' ,
                 'PKD_KeyIdentifier_length'   'PKD_KeyIdentifier'   ,
                 'PKD_target_Keyvalue_length' 'PKD_target_Keyvalue' ;

    PKD_target_keyvalue = ,
     substr(PKD_target_keyvalue,1,c2d(PKD_target_keyvalue_length)) ;
?
  If PKD_rc = Exp_Rc & PKD_rs = Exp_Rc then
   Do;
    say "PKD successful"
    if PKE_Keyvalue /= PKD_target_Keyvalue then
     do
      say '***** Error PKE_keyvalue <> PKD_target_keyvalue *****'
      say ' PKE_keyvalue         : ' PKE_Keyvalue
      say ' PKD_target_Keyvalue  : ' PKD_target_Keyvalue
     end;
   end;
  Else
    say 'PKD : failed rc =' c2x(PKD_rc) 'rs =' c2x(PKD_rs)

say
return
```

## E.3 PKCS #11 hybrid quantum-safe key exchange scheme REXX sample

A PKCS #11 hybrid quantum-safe key exchange scheme REXX sample is shown in Example E-2.

*Example E-3 PKCS #11 Hybrid Quantum-safe key exchange scheme REXX sample*

```
/* REXX */

/*********************************************************************/
/* PKCS #11 Hybrid Quantum-safe Key Exchange Scheme                  */
/*********************************************************************/
SIGNAL ON NOVALUE;

Call TCSETUP

/*********************************************************************/
/* Common test data                                                  */
/*********************************************************************/
/* expected results */
ExpRC = '00000000'x ;
ExpRS = '00000000'x ;

exit_data_length     = '00000000'X;
exit_data            = '';
GKP_EC_pub_attr_list =,
    '0006'X ||,
    CKA_CLASS              || '0004'X || CKO_PUBLIC_KEY           ||,
    CKA_KEY_TYPE           || '0004'X || CKK_EC                   ||,
    CKA_TOKEN              || '0001'X || CK_TRUE                  ||,
    CKA_IBM_SECURE         || '0001'X || CK_TRUE                  ||,
    CKA_EC_PARAMS          || D2C(LENGTH(secp521r1),2) ||,
                                         secp521r1                ||,
    CKA_LABEL            /*|| 'llll'X || 'label'                 */ ;
GKP_EC_prv_attr_list =,
    '0005'X ||,
    CKA_CLASS              || '0004'X || CKO_PRIVATE_KEY          ||,
    CKA_KEY_TYPE           || '0004'X || CKK_EC                   ||,
    CKA_TOKEN              || '0001'X || CK_TRUE                  ||,
    CKA_IBM_SECURE         || '0001'X || CK_TRUE                  ||,
    CKA_LABEL            /*|| 'llll'X || 'label'                 */ ;
GKP_Kyber_pub_attr_list =,
    '0006'X ||,
    CKA_CLASS              || '0004'X || CKO_PUBLIC_KEY           ||,
    CKA_KEY_TYPE           || '0004'X || CKK_IBM_KYBER            ||,
    CKA_TOKEN              || '0001'X || CK_TRUE                  ||,
    CKA_IBM_SECURE         || '0001'X || CK_TRUE                  ||,
    CKA_IBM_KYBER_MODE     || D2C(LENGTH(DER_OID_KYBER_1024_R2),2) ||,
                                         DER_OID_KYBER_1024_R2    ||,
    CKA_LABEL            /*|| 'llll'X || 'label'                 */ ;
GKP_Kyber_prv_attr_list =,
    '0005'X ||,
    CKA_CLASS              || '0004'X || CKO_PRIVATE_KEY          ||,
    CKA_KEY_TYPE           || '0004'X || CKK_IBM_KYBER            ||,
```

```
    CKA_TOKEN              || '0001'X || CK_TRUE                  ||,
    CKA_IBM_SECURE         || '0001'X || CK_TRUE                  ||,
    CKA_LABEL           /*|| 'llll'X || 'label'                 */ ;
DVK_attr_list_ECDH =,
    '0004'X ||,
    CKA_CLASS              || '0004'X || CKO_SECRET_KEY           ||,
    CKA_IBM_SECURE         || '0001'X || CK_TRUE                  ||,
    CKA_KEY_TYPE           || '0004'X || CKK_GENERIC_SECRET       ||,
    CKA_VALUE_LEN          || '0004'X || '00000042'X              ;
DVK_attr_list_Kyber =,
    '0004'X ||,
    CKA_CLASS              || '0004'X || CKO_SECRET_KEY           ||,
    CKA_IBM_SECURE         || '0001'X || CK_TRUE                  ||,
    CKA_KEY_TYPE           || '0004'X || CKK_AES                  ||,
    CKA_VALUE_LEN          || '0004'X || '00000020'X              ;
known_clear_text = COPIES('A',16);

my_token = Left('QSAFE.TEST.TOKEN',44) /* Replace this token handle */


/*********************************************************************/
/* Step 1.1 Generate an ECC key pair for Alice                       */
/*********************************************************************/
testN = 'ECALICE';
pub_key_attr_list = GKP_EC_pub_attr_list||D2C(LENGTH(testN),2)||testN;
prv_key_attr_list = GKP_EC_prv_attr_list||D2C(LENGTH(testN),2)||testN;
CALL CSFPGKP;
handle_EC_Pub_A  = pub_key_object_handle;
handle_EC_Priv_A = prv_key_object_handle;

/*********************************************************************/
/* Step 2.2 Generate an ECC key pair for Bob                         */
/*********************************************************************/
testN = 'ECBOB';
pub_key_attr_list = GKP_EC_pub_attr_list||D2C(LENGTH(testN),2)||testN;
prv_key_attr_list = GKP_EC_prv_attr_list||D2C(LENGTH(testN),2)||testN;
CALL CSFPGKP;
handle_EC_Pub_B  = pub_key_object_handle;
handle_EC_Priv_B = prv_key_object_handle;

/*********************************************************************/
/* Step 2.2 Generate a Kyber key pair for Bob                        */
/*********************************************************************/
testN = 'QSBOB';
pub_key_attr_list=GKP_Kyber_pub_attr_list||D2C(LENGTH(testN),2)||testN;
prv_key_attr_list=GKP_Kyber_prv_attr_list||D2C(LENGTH(testN),2)||testN;
CALL CSFPGKP;
handle_Kyb_Pub_B  = pub_key_object_handle;
handle_Kyb_Priv_B = prv_key_object_handle;


/*********************************************************************/
/* Step 2.3 Derive a key using ECDH(HYBRID_NULL) with Bob's Private  */
/* ECC key and Alice Public ECC key                                  */
/*********************************************************************/
```

```
testN = 'DRVGENSECB';
pub_EC_POINT = CSFPGAV(handle_EC_Pub_A,CKA_EC_POINT);
rule_array               = 'EC-DH   ';
attribute_list           = DVK_attr_list_ECDH;
base_key_handle          = handle_EC_Priv_B;
DVK_ParmsList               =,
       CKD_IBM_HYBRID_NULL          ||, /* KDF function code      */
       '00000000'X                  ||, /* Optional data length   */
       '0000000000000000'X          ||, /* Optional data address  */
       D2C(LENGTH(pub_EC_POINT),4)  ||, /* Public value length    */
       pub_EC_POINT;                   /* Public value           */
CALL CSFPDVK;

handle_GenSec_B = target_key_handle;


/*********************************************************************/
/* Step 3.3 Derive a key using ECDH(HYBRID_NULL) with Alice's Private*/
/* ECC key and Bob's Public ECC key                                  */
/*********************************************************************/
testN = 'DRVGENSECA';
pub_EC_POINT = CSFPGAV(handle_EC_Pub_B,CKA_EC_POINT);
rule_array               = 'EC-DH   ';
attribute_list           = DVK_attr_list_ECDH;
base_key_handle          = handle_EC_Priv_A;
DVK_ParmsList               =,
       CKD_IBM_HYBRID_NULL          ||, /* KDF function code      */
       '00000000'X                  ||, /* Optional data length   */
       '0000000000000000'X          ||, /* Optional data address  */
       D2C(LENGTH(pub_EC_POINT),4)  ||, /* Public value length    */
       pub_EC_POINT;                   /* Public value           */
CALL CSFPDVK;
handle_GenSec_A = target_key_handle;


/*********************************************************************/
/* Step 3.4 Derive key using KYBER(HYBRID_SHA256), then encapsulate  */
/* Bob's Public Kyber key                                            */
/*********************************************************************/
testN = 'DRVSHAREDA';
rule_array               = 'KYBER   ';
attribute_list           = DVK_attr_list_Kyber;
base_key_handle          = handle_Kyb_Pub_B;

DVK_ParmsList               =,
       '00000000'X                  ||, /* version                */
       CK_IBM_KEM_ENCAPSULATE       ||, /* mode                   */
       CKD_IBM_HYBRID_SHA256_KDF    ||, /* kdf                    */
       CK_FALSE                     ||, /* prepend                */
       COPIES('00'X,3)              ||, /* reserved               */
       D2C(0,4)                     ||, /* shared data len        */
       D2C(1600,4)                  ||, /* cipher len (output)    */
       handle_GenSec_A              ||, /* gen secret key handle  */
       COPIES('42'X,1600);              /* buffer for cipher output  */
```

```
CALL CSFPDVK;
CALL parse_Kyber_parmslist;
handle_SharedKey_A = target_key_handle;



/*********************************************************************/
/* Step 4.1 Derive key using KYBER(HYBRID_SHA256) using decapsulate  */
/* with Bob's Private Kyber key                                      */
/*********************************************************************/
testN = 'DRVSHAREDB';
rule_array                = 'KYBER   ';
attribute_list            = DVK_attr_list_Kyber;
base_key_handle           = handle_Kyb_Priv_B;
DVK_ParmsList                =,
       '00000000'X                  ||, /* version                  */
       CK_IBM_KEM_DECAPSULATE       ||, /* mode                     */
       CKD_IBM_HYBRID_SHA256_KDF    ||, /* kdf                      */
       CK_FALSE                     ||, /* prepend                  */
       COPIES('00'X,3)              ||, /* reserved                 */
       D2C(0,4)                     ||, /* shared data len          */
       d2c( length(cphr),4  )       ||, /* cipher len (input)       */
       handle_GenSec_B              ||, /* gen secret key handle    */
       cphr                         ;   /* cipher from previous step */
CALL CSFPDVK;
handle_SharedKey_B = target_key_handle;


/*********************************************************************/
/* Encrypt some data with Alice's SharedKey                          */
/*********************************************************************/
testN = 'ENCSHAREDA';
rule_array                = 'AES     ECB     ONLY    ';
key_handle                = handle_SharedKey_A
init_vector               = '';
clear_text                = known_clear_text;
CALL CSFPSKE;
SAY 'ciphertext('||testN||'): '||C2X(cipher_text);
cipher_text_SharedKey_A = cipher_text;

/*********************************************************************/
/* Encrypt some data with Bob's SharedKey                            */
/*********************************************************************/
 testN = 'ENCSHAREDB';
rule_array                = 'AES     ECB     ONLY    ';
key_handle                = handle_SharedKey_B;
init_vector               = '';
clear_text                = known_clear_text;
CALL CSFPSKE;
SAY 'ciphertext('||testN||'): '||C2X(cipher_text);
cipher_text_SharedKey_B = cipher_text;

/*********************************************************************/
/* Verify cipher text is identical                                   */
/*********************************************************************/
```

```
IF cipher_text_SharedKey_B = cipher_text_SharedKey_A THEN
  SAY 'TESTCASE SUCCESSFUL'


GETOUT: ;
EXIT;
/*********************************************************************/
/* parse_Kyber_parmslist                                             */
/*********************************************************************/
parse_Kyber_parmslist:
    PARSE VALUE DVK_ParmsList WITH ,
              ver              +4  ,
              mode             +4  ,
              kdf              +4  ,
              pre              +1  ,
              rsvd             +3  ,
              shrdlen          +4  ,
              cphrlen          +4  ,
              gskH             +44 ,
              remaining            ;
    shrdlenD = C2D(shrdlen);
    cphrlenD = C2D(cphrlen);
    PARSE VALUE remaining WITH ,
              shrd             +(shrdlenD) ,
              cphr             +(cphrlenD) ,
              extra                        ;
    verP     = "'"||C2X(ver)||"'X (version "||C2D(ver)||")";
    modeP    = "'"||C2X(mode)||"'X";
    SELECT;
      WHEN mode = CK_IBM_KEM_ENCAPSULATE THEN
        modeP = modeP||" (CK_IBM_KEM_ENCAPSULATE)";
      WHEN mode = CK_IBM_KEM_DECAPSULATE THEN
        modeP = modeP||" (CK_IBM_KEM_DECAPSULATE)";
      OTHERWISE
        modeP = modeP||" (unknown)";
    END;
    kdfP     = "'"||C2X(kdf)||"'X";
    SELECT;
      WHEN kdf = CKD_IBM_HYBRID_SHA1_KDF THEN
        kdfP = kdfP||" (CKD_IBM_HYBRID_SHA1_KDF)";
      WHEN kdf = CKD_IBM_HYBRID_SHA224_KDF THEN
        kdfP = kdfP||" (CKD_IBM_HYBRID_SHA224_KDF)";
      WHEN kdf = CKD_IBM_HYBRID_SHA256_KDF THEN
        kdfP = kdfP||" (CKD_IBM_HYBRID_SHA256_KDF)";
      WHEN kdf = CKD_IBM_HYBRID_SHA384_KDF THEN
        kdfP = kdfP||" (CKD_IBM_HYBRID_SHA384_KDF)";
      WHEN kdf = CKD_IBM_HYBRID_SHA512_KDF THEN
        kdfP = kdfP||" (CKD_IBM_HYBRID_SHA512_KDF)";
      OTHERWISE
        kdfP = kdfP||" (unknown)";
    END;
    preP     = "'"||C2X(pre)||"'X";
    SELECT;
      WHEN pre = CK_FALSE THEN
        preP = preP||"       (don't prepend)";
```

```
      WHEN pre = CK_TRUE THEN
        preP = preP||"       (do prepend)";
      OTHERWISE
        preP = preP||"       (unknown)";
    END;
    rsvdP    = "'"||C2X(rsvd)||"'X";
    shrdlenP = "'"||C2X(shrdlen)||"'X ("||shrdlenD||")";
    cphrlenP = "'"||C2X(cphrlen)||"'X ("||cphrlenD||")";
    gskHP    = "'"||gskH||"'";

RETURN;

/* ---------------------------------------------------------------- */
/* PKCS #11 Generate Key Pair                                       */
/*                                                                  */
/* Use the PKCS #11 Generate Key Pair callable service to generate */
/* an RSA, DSA, Elliptic Curve, Diffie-Hellman, Dilithium (LI2) or */
/* Kyber key pair.                                                  */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ---------------------------------------------------------------- */
CSFPGKP:
return_code              = 'FFFFFFFF'X;
reason_code              = 'FFFFFFFF'X;
token_handle             = my_token;
rule_array_count         = '00000000'X;
rule_array               = '';
/* pub_key_attr_list is set by caller */
pub_key_attr_list_length  = D2C(LENGTH(pub_key_attr_list),4);
pub_key_object_handle     = COPIES(' ',44);
/* prv_key_attr_list is set by caller */
prv_key_attr_list_length  = D2C(LENGTH(prv_key_attr_list),4);
prv_key_object_handle     = COPIES(' ',44);
ADDRESS LINKPGM 'CSFPGKP',
                'return_code'              'reason_code'        ,
                'exit_data_length'         'exit_data'          ,
                'token_handle'                                  ,
                'rule_array_count'         'rule_array'         ,
                'pub_key_attr_list_length' 'pub_key_attr_list'  ,
                'pub_key_object_handle'                         ,
                'prv_key_attr_list_length' 'prv_key_attr_list'  ,
                'prv_key_object_handle'                        ;
IF (return_code \= ExpRC) | (reason_code \= ExpRS) THEN
  DO;
    SAY 'GKP('||testN||'): rc/rs='||C2X(return_code)||'/'||,
                                    C2X(reason_code);
    SIGNAL GETOUT;
  END;
Else
  DO;
    SAY 'GKP('||testN||'): successful';
    SAY '  pub_key_object_handle = "'||pub_key_object_handle||'"';
    SAY '  prv_key_object_handle = "'||prv_key_object_handle||'"';
  END;
```

```
RETURN;

/* ------------------------------------------------------------- */
/* PKCS #11 Derive Key                                           */
/*                                                               */
/* Use the PKCS #11 Derive Key callable service to generate a new  */
/* secret key object from an existing key object.                */
/*                                                               */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ------------------------------------------------------------- */
CSFPDVK:
return_code               = 'FFFFFFFF'X;
reason_code               = 'FFFFFFFF'X;
rule_array_count          = D2C(TRUNC((LENGTH(rule_array)+7)/8),4);
/* rule_array (properly padded) is set by caller */
/* attribute_list is set by caller */
attribute_list_length     = D2C(LENGTH(attribute_list),4);
/* base_key_handle is set by caller */
/* DVK_ParmsList is set by caller */
DVK_ParmsList_length         = D2C(LENGTH(DVK_ParmsList),4);
target_key_handle         = COPIES('DD'X,44);
ADDRESS LINKPGM 'CSFPDVK',
                'return_code'               'reason_code'        ,
                'exit_data_length'          'exit_data'          ,
                'rule_array_count'          'rule_array'         ,
                'attribute_list_length'     'attribute_list'     ,
                'base_key_handle'                                ,
                'DVK_ParmsList_length'         'DVK_ParmsList'         ,
                'target_key_handle'                              ;
IF (return_code \= ExpRC) | (reason_code \= ExpRS) THEN
  DO;
    SAY 'DVK('||testN||'): rc/rs='||C2X(return_code)||'/'||,
                                    C2X(reason_code);
    SIGNAL GETOUT;
  END;
Else
  DO;
    SAY 'DVK('||testN||'): successful';
    SAY '  target_key_handle = "'||target_key_handle||'"';
  END;
RETURN;


/* ------------------------------------------------------------- */
/* PKCS #11 Secret Key Encrypt                                   */
/*                                                               */
/* Use the PKCS #11 Secret Key Encrypt callable service to encipher*/
/* data using a symmetric key.                                   */
/*                                                               */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ------------------------------------------------------------- */
CSFPSKE:
return_code               = '99999999'X;
reason_code               = '99999999'X;
rule_array_count          = D2C(TRUNC((LENGTH(rule_array)+7)/8),4);
```

```
/* rule_array (properly padded) is set by caller */
/* key_handle is set by caller */
init_vector_length        = D2C(LENGTH(init_vector),4);
/* init_vector is set by caller */
chain_data_length         = '00000080'X
chain_data                = COPIES('00'X,C2D(chain_data_length));
clear_text_length         = D2C(LENGTH(clear_text),4);
/* clear_text is set by caller */
clear_text_id             = '00000000'X;
cipher_text_length        = D2C(C2D(clear_text_length)+16,4);
cipher_text               = COPIES('00'X,C2D(cipher_text_length));
cipher_text_id            = '00000000'X;
ADDRESS LINKPGM 'CSFPSKE'                                  ,
                'return_code'          'reason_code'       ,
                'exit_data_length'     'exit_data'         ,
                'rule_array_count'     'rule_array'        ,
                'key_handle'                               ,
                'init_vector_length'   'init_vector'       ,
                'chain_data_length'    'chain_data'        ,
                'clear_text_length'    'clear_text'        ,
                'clear_text_id'                            ,
                'cipher_text_length'   'cipher_text'       ,
                'cipher_text_id'                           ;
IF (return_code \= ExpRC) | (reason_code \= ExpRS) THEN
  DO;
    SAY 'SKE('||testN||'): rc/rs='||C2X(return_code)||'/'||,
                                    C2X(reason_code);
    SIGNAL GETOUT;
  END;
Else
  SAY 'SKE('||testN||'): successful';
cipher_text = LEFT(cipher_text,C2D(cipher_text_length));
RETURN;

/* ---------------------------------------------------------------- */
/* PKCS #11 Get Attribute Value                                     */
/*                                                                  */
/* Use the PKCS #11 Get Attribute Value callable service (CSFPGAV) */
/* to retrieve the attributes of an object.                         */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.   */
/* ---------------------------------------------------------------- */
CSFPGAV:
PARSE ARG RATTR.handle,RATTR.attr;
shortHandle = LEFT(RATTR.handle,41);
return_code      = 'FFFFFFFF'X;
reason_code      = 'FFFFFFFF'X;
rule_array_count = '00000000'X;
handle           = RATTR.handle;
rule_array       = '';
attr_list_length = D2C(32000,4);
attr_list        = COPIES('FF'X,32000);
ADDRESS LINKPGM 'CSFPGAV' ,
                'return_code'      'reason_code'   ,
                'exit_data_length' 'exit_data'     ,
```

```
                'handle'                          ,
                'rule_array_count' 'rule_array'   ,
                'attr_list_length' 'attr_list'    ;
IF (return_code \= ExpRC) | (reason_code \= ExpRS) THEN
  DO;
    SAY 'CSFPGAV('||shortHandle||'): rc = '||C2X(return_code)||,
                 ' rs = '||C2X(reason_code);
    SIGNAL GETOUT;
  END;
attr_list = LEFT(attr_list,C2D(attr_list_length));
number_attributes = C2D(LEFT(attr_list,2));
attr_list = SUBSTR(attr_list,3);
DO n = 1 TO number_attributes;
  attr_number  = LEFT(attr_list,4);
  attr_list    = SUBSTR(attr_list,5);
  attr_val_len = C2D(LEFT(attr_list,2));
  attr_list    = SUBSTR(attr_list,3);
  attr_value   = LEFT(attr_list,attr_val_len);
  attr_list    = SUBSTR(attr_list,attr_val_len+1);
  IF (attr_number = RATTR.attr) THEN
    SIGNAL DONE_W_READ_ATTR;
END;
attr_value = 'BADBADBAD';
DONE_W_READ_ATTR: ;
RETURN attr_value;

TCSETUP:

DER_OID_KYBER_1024_R2 = '060B2B0601040102820B050404'X;
secp521r1             = '06052b81040023'x

CKK_IBM_KYBER         = '80010024'X;
CKK_EC                = '00000003'X
CKK_GENERIC_SECRET    = '00000010'X
CKK_AES               = '0000001F'X

CKO_PUBLIC_KEY        = '00000002'X
CKO_PRIVATE_KEY       = '00000003'X
CKO_SECRET_KEY        = '00000004'X

CKA_CLASS             = '00000000'X
CKA_TOKEN             = '00000001'X
CKA_IBM_KYBER_MODE    = '8000000E'X
CKA_LABEL             = '00000003'X
CKA_IBM_SECURE        = '80000006'X
CKA_EC_PARAMS         = '00000180'X
CKA_EC_POINT          = '00000181'X
CKA_VALUE_LEN         = '00000161'X
CKA_KEY_TYPE          = '00000100'X

CKD_IBM_HYBRID_NULL        = '80000001'X;
CKD_IBM_HYBRID_SHA1_KDF    = '80000002'X;
CKD_IBM_HYBRID_SHA224_KDF  = '80000003'X;
CKD_IBM_HYBRID_SHA256_KDF  = '80000004'X;
CKD_IBM_HYBRID_SHA384_KDF  = '80000005'X;
```

```
CKD_IBM_HYBRID_SHA512_KDF  = '80000006'X;

CK_IBM_KEM_ENCAPSULATE    = '00000001'X;
CK_IBM_KEM_DECAPSULATE    = '00000002'X;

CK_TRUE                   = '01'x
CK_FALSE                  = '00'x
return

NOVALUE:
SAY "Condition NOVALUE was raised."
SAY CONDITION("D") "variable was not initialized."
SAY sigl||': '||SOURCELINE(sigl)
EXIT;
```

F

# Generating a one-way hash

The examples in this appendix are REXX executables that can be used to generate an SHA-512 one-way hash, using CCA and PKCS #11.

This appendix includes the following topics:

# F.1 CCA SHA-512 one-way hash REXX sample

A CCA SHA-512 one-way hash REXX sample is shown in Example F-1.

*Example F-1 CCA SHA-512 one-way hash REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* Generate SHA-512 hash using CCA One-Way Hash service               */
/*--------------------------------------------------------------------*/

 /* expected results */
 ExpRc = '00000000'x
 ExpRs = '00000000'x

 BOWH_Rule_Array   = 'SHA-512 ' || 'ONLY    ' ;
 BOWH_Text         = '0123456789ABCDEF';
 BOWH_Hash         = copies('00'x, 64);
 BOWH_Chain_Vector = copies('00'x,128);

 call CSNBOWH

 say 'BOWH Hash: ' c2x(BOWH_Hash)

 Exit

/* ---------------------------------------------------------------- */
/* One-Way Hash Generate                                            */
/*                                                                  */
/* Used to generate a one-way hash                                  */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */

CSNBOWH:

 /* initialize parameter list */
 BOWH_rc                  = 'FFFFFFFF'x ;
 BOWH_rs                  = 'FFFFFFFF'x ;
 BOWH_Exit_Length         = '00000000'x ;
 BOWH_Exit_Data           = '00000000'x ;
 BOWH_Rule_Count          = d2c(length(BOWH_Rule_Array)/8,4);
 BOWH_Text_Length         = d2c(length(BOWH_Text),4);
 BOWH_Chain_Vector_Length = d2c(length(BOWH_Chain_Vector),4);
 BOWH_Hash_Length         = d2c(Length(BOWH_Hash),4);

 /* call CSNBOWH */
 address linkpgm 'CSNBOWH' ,
                 'BOWH_rc'                  'BOWH_rs'          ,
                 'BOWH_Exit_Data_Length'    'BOWH_Exit_Data'   ,
                 'BOWH_Rule_Count'          'BOWH_Rule_Array'  ,
                 'BOWH_Text_Length'         'BOWH_Text'        ,
                 'BOWH_Chain_Vector_Length' 'BOWH_Chain_Vector' ,
                 'BOWH_Hash_Length'         'BOWH_Hash'        ;
```

```
 if (BOWH_rc \= ExpRc | BOWH_rs \= ExpRs) then
  say 'BOWH failed: rc =' c2x(BOWH_rc) 'rs =' c2x(BOWH_rs) ;
 else
  say 'BOWH successful: rc =' c2x(BOWH_rc) 'rs =' c2x(BOWH_rs) ;


return
```

## F.2 PKCS #11 SHA-512 one-way hash REXX sample

A PKCS #11 SHA-512 one-way hash REXX sample is shown in Example F-2.

*Example F-2 PKCS #11 SHA-512 one-way hash REXX sample*

```
/* Rexx */

/*--------------------------------------------------------------------*/
/* Generate SHA-512 hash using PKCS #11 One-Way Hash service          */
/*--------------------------------------------------------------------*/

 /* expected results */
 ExpRc = '00000000'x
 ExpRs = '00000000'x


 /* Call PKCS#11 One-Way Hash with generated token */
 POWH_Rule_Array   = 'SHA-512 ' || 'ONLY    ' ;
 POWH_Text         = '0123456789ABCDEF';
 POWH_Hash         = copies('00'x, 64);
 POWH_Chain_Vector = copies('00'x,128);
 POWH_Handle       = Left('QSAFE.TEST.TOKEN',44)

 call CSNPOWH

 say 'POWH Hash: ' c2x(POWH_Hash)


 Exit
/* ---------------------------------------------------------------- */
/* PKCS #11 One-Way Hash, Sign, or Verify                           */
/*                                                                  */
/* Use the PKCS #11 One-Way Hash, Sign, or Verify callable service */
/* to generate a one-way hash on specified text, sign specified     */
/* text, or verify a signature on specified text.                   */
/*                                                                  */
/* See the ICSF Application Programmer's Guide for more details.    */
/* ---------------------------------------------------------------- */

CSNPOWH:

 /* initialize parameter list */
 POWH_RC                  = 'FFFFFFFF'x ;
 POWH_RS                  = 'FFFFFFFF'x ;
```

```
POWH_Exit_Length         = '00000000'x ;
POWH_Exit_Data           = '' ;
POWH_Rule_Count          = d2c(length(POWH_Rule_Array)/8,4);
POWH_Text_Length         = d2c(length(POWH_Text),4);
POWH_Text_id             = '00000000'x ;
POWH_Chain_Vector_Length = d2c(length(POWH_Chain_Vector),4);
POWH_Hash_Length         = D2C(Length(POWH_Hash),4);


/* call CSNPOWH */
address linkpgm 'CSFPOWH' ,
                'POWH_RC'                     'POWH_RS'           ,
                'POWH_Exit_Length'            'POWH_Exit_Data'    ,
                'POWH_Rule_Count'             'POWH_Rule_Array'   ,
                'POWH_Text_Length'            'POWH_Text'         ,
                'POWH_Text_id'                                    ,
                'POWH_Chain_Vector_Length'    'POWH_Chain_Vector' ,
                'POWH_Handle'                                     ,
                'POWH_Hash_Length'            'POWH_Hash'         ;


if (POWH_rc \= ExpRc | POWH_rs \= ExpRs) then
 say 'POWH failed: rc =' c2x(POWH_rc) 'rs =' c2x(POWH_rs) ;
else
 say 'POWH successful: rc =' c2x(POWH_rc) 'rs =' c2x(POWH_rs) ;


return
```

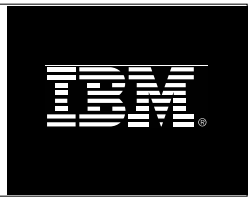
To determine the spine width of a book, you divide the paper PPI into the number of pages in the book. An example is a 250 page book using Plainfield opaque 50# smooth which has a PPI of 526. Divided 250 by 526 which equals a spine width of .4752". In this case, you would use the .5” spine. Now select the Spine width for the book and hide the others: **Special>Conditional Text>Show/Hide>SpineSize(-->Hide:)>Set** . Move the changed Conditional text settings to all files in your book by opening the book file with the spine.fm still open and **File>Import>Formats** the Conditional Text Settings (ONLY!) to the book files.



# Transitioning to Quantum-Safe Cryptography on IBM Z

SG24-8525-01

ISBN



(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

To determine the spine width of a book, you divide the paper PPI into the number of pages in the book. An example is a 250 page book using Plainfield opaque 50# smooth which has a PPI of 526. Divided 250 by 526 which equals a spine width of .4752". In this case, you would use the .5” spine. Now select the Spine width for the book and hide the others: **Special>Conditional Text>Show/Hide>SpineSize(-->Hide:)>Set** . Move the changed Conditional text settings to all files in your book by opening the book file with the spine.fm still open and **File>Import>Formats** the Conditional Text Settings (ONLY!) to the book files.

SG24-8525-01

ISBN

Printed in U.S.A.