

Quantum Protect

Quick Start guide

Author: Richard Williamson

Reviewer: Eric Portrait



utimaco®

Imprint

Copyright 2025	Utimaco IS GmbH Germanusstr. 4 D-52080 Aachen Germany
Phone	+49 (0)241 / 1696-200
Fax	+49 (0)241 / 1696-199
Internet	utimaco.com
e-mail	hsm@utimaco.com
Document Version	1.3
Date	2025-08-20

All Rights reserved	<p>No part of this documentation may be reproduced in any form (printing, photocopy or according to any other process) without the written approval of Utimaco IS GmbH or be processed, reproduced or distributed using electronic systems.</p> <p>Utimaco IS GmbH reserves the right to modify or amend the documentation at any time without prior notice. Utimaco IS GmbH assumes no liability for typographical errors and damages incurred due to them.</p> <p>All trademarks and registered trademarks are the property of their respective owners.</p>
---------------------	---

Contents

1	Introduction	5
2	Prerequisites	6
2.1	Preliminary steps	6
2.2	Required Versions:	6
2.2.1	Using Quantum Protect on a u.trust SE HSM	6
2.2.2	Using Quantum Protect on the simulator	6
3	Simulator installation	8
4	HSM Installation steps:	9
4.1	Step 1 : Basic setup	9
4.1.1	u.trust SE LAN appliance	9
4.1.2	u.trust SE PCIe	9
4.2	Step 2 : Configure the HSM (LAN or PCIe) via GLADM	9
4.3	Step 3 : Connect to the HSM	11
4.4	Step4 : Check the cHSM is available	11
4.5	Step 5 : Check that the cHSM has an MBK	13
4.6	Step 6 : Configure the cHSM for Quantum Protect	13
5	Managing HSM Backups	16
6	Configure Quantum Protect via PKCS11	17
6.1	ML Keys	17
6.2	HBS Keys	17
7	Algorithm Support	18
7.1	PQMI sub-function interface support:	18
7.2	PKCS11 Vendor Defined Mechanism support:	18
7.3	C Typed Interfaces	19
7.4	Java Typed Interfaces	19
8	ML-DSA	22
8.1	Key Types	22
8.2	Key Generation – MLDSA_KeyGen.java and MLDSA_KeyGen.h/_imp.c	22
8.3	Signing – MLDSA_Sign.java, MLDSA_Sign.h/_imp.c	24
8.4	Verifying	25
8.5	Key Wrap – MLXXX_KeyWrap.h/_imp.c	26
8.6	Key Unwrap – MLXXX_KeyWrap.h/_imp.c	27

9	ML-KEM	28
9.1	Key Types	28
9.2	Key Generation – MLKEM_KeyGen.java and MLDSA_KeyGen.h/_imp.c	28
9.3	Key Encapsulation	30
9.3.1	Encapsulate – MLKEM_Encap.java, MLKEM_Encap.h/_imp.c	30
9.3.2	Decapsulate – MLKEM_Decap.java, MLKEM_Decap.h/_imp.c	31
10	LMS and HSS	33
10.1	Key Provisioning/Apportionment	33
10.2	Introduction	33
10.3	Example Source	33
10.4	Key Generation	34
10.5	Public Key retrieval	36
10.6	Sign	36
10.7	Verify	36
11	Create Multiple cHSMs from a snapshot (Non-SHBS algorithms)	38
12	SHBS State Management	39
13	Known Issues	41
14	Release Notes	43

1 Introduction

Quantum Protect (QP) is an extension to the Utimaco u.trust SecurityServer general-purpose Hardware Security Modules (HSMs).

It extends the cryptographic algorithms running inside the tamper-proof environment of the HSMs, adding support for different “Post-Quantum” algorithms.

Version 1.0 (March 2025):

- ML-KEM (FIPS 203)
- ML-DSA (FIPS 204)
- LMS/HSS (FIPS SP800 – 208)
- LMS/HSS(SP800-208)
- XMSS/XMSS-MT (SP800 - 208)

Future versions of QP will include additional algorithms based on market need. Please contact Utimaco ([Appendix A](#)) for more information about upcoming additional supported algorithms.

2 Prerequisites

This QuickStart guide gives guidance to users on how to install Quantum Protect firmware modules into supported Utimaco HSMs, which are currently installed and operational. It also provides guidance on using the sample code supplied with the modules. Sample code currently exists for PKCS11, and for Java via a custom interface. Both of these sample sets use Vendor Defined Mechanisms on top of a standard. In PKCS11 this is a well-defined process.

For Java_UTI, you can look at these as “vendor defined mechanisms for CXI”.

As UCAPI (the improved CXI layer that PKCS11_R3 is based on) is extended to support the PQC algorithms directly, a migration path will be provided from the Java_UTI methodology to the UCAPI/Java methodology for the new algorithms.

2.1 Preliminary steps

QuantumProtect is a set of firmware modules that can be used in either a physical HSM (PCIe card or LAN appliance) or in the HSM simulator. Please ensure that you are using the correct .mtc files, based on the target environment (Windows Simulator, Linux Simulator, or u.trust Anchor GP-HSM). Depending on need, you will need to, or will have already done:

- Install and configure the u.Trust Se HSM (PCIe card or the LAN Appliance)
- Alternatively installed and configured the HSM Simulator
- Alternatively installed the quantum protect CD, which also installs a local copy of the simulator, preloaded with the QP modules and preconfigured for PKCS11 usage on Slot 0.

2.2 Required Versions:

2.2.1 Using Quantum Protect on a u.trust SE HSM

- For the HSM, PCIe or LAN appliance: expected firmware version 4.80 or later
- For the LAN appliance: expected LANOS version 5.8c.0 or later
- For the Quantum Protect firmware module and libraries: versions: 1.1.0.0 or later

2.2.2 Using Quantum Protect on the simulator

You can download and use the standard Utimaco simulator. In this case ensure you download the simulator in version 4.80 or later.

Alternatively, you can download the Quantum Protect CD which includes the simulator in the expected version, with the quantum protect modules preloaded into it.

Note : if you are already using the Utimaco simulator, it uses the default port 3001.

If you want to run the Quantum Protect simulator on the same host, you have to use a different port number. Set the environment variable `SDK_PORT` to the desired port, and the instance of the simulator will listen on it

On Windows: `set SDK_PORT=3301 && cs_sim.bat`

On Linux: `$ SDK_PORT=3301 ./cs_sim.sh`

3 Simulator installation

The Quantum protect product CD comes with a pre-configured simulator. Run the installer and then execute the simulator on a given port number to use it.

Windows: `set SDK_PORT=3301 && cs_sim.bat`

Linux: `SDK_PORT=3301 ./cs_sim.sh`

This allows you to run a standard Simulator on 3001, and a QP-enabled simulator on some other port.

Once the simulator is running you can skip to Section 4.4, [Step4 : Check the cHSM is available](#) .

4 HSM Installation steps:

4.1 Step 1 : Basic setup

Depending on the installation type, Lan appliance, PCIe card, please refer to the corresponding paragraph below

4.1.1 u.trust SE LAN appliance

Installing the u.trust SE Lan appliance require to :

- Setup an IP address, and optionally enabling ssh daemon

Please refer to the standard product documentation located in SecurityServer product CD :
ustrust_Anchor_LAN_V5_QuickStartGuide.pdf

4.1.2 u.trust SE PCIe

Installing the u.trust SE PCIe card require to :

- Setup the card in the computer, and Install the PCIe driver

The driver installation process differs between Linux and Windows systems. Please refer to the standard product documentation located in SecurityServer product CD

- CryptoServer_QuickStartGuide_PcLe_Linux.pdf
- CryptoServer_QuickStartGuide_PcLe_Windows.pdf

4.2 Step 2 : Configure the HSM (LAN or PCIe) via GLADM

This step is the same for both the LAN appliance and the PCIe card. This step is not necessary for the simulator.

After the device is online and visible via gladm's system-get-info command,

- Verify the device is genuine (gladm)
- Create default admins/operators (gladm)
- Claim the device (gladm)
- Create or import an operator secret (gladm)
- Create default admin key ("CAAK" key) for a cHSM (containerized HSM, csadm)
- Instantiate a cHSM with the SecurityServer-SDK template and the public part of the CAAK key generated above (gladm). Important: There are other templates that can work. The template name must contain the symbol "SDK" (SecurityServer-SDK, SecurityServer-FIPS-SDK, etc)
- Generate or import an MBK into the cHSM (csadm)

Please refer to the Utimaco product CD Documentation for available QuickStart Guides covering the initial setup of your HSMs.

Verify that the system is visible using *gladm* or *csadm*, and the correct device locator. Device locator syntax is determined by the HSM form factor and tool.

	gladm	csadm/cxtool/p11tool2/cat
Windows PCIe	-d PCI:<N>	Dev=PCI:<N>.<index>
Linux PCIe	-d /dev/cs2.<N>	Dev=/dev/cs2.<N>.<index>
CSLan or Simulator	-d <IP> or -d <hostname> -p <port>	Dev=<port>@<IP> or Dev=<port>@<hostname>

N: Logical index of PCIe card on PCIe bus

index: Container/Slot ID of the container on the

HSM IP: IP address (IPv4 usually although IPv6

is supported) hostname: Hostname of the

appliance targeted

port: By default, 4000 for gladm and 4001 or higher for a container, or 3001 or as provided by SDK_PORT for the Simulator.

4.3 Step 3 : Connect to the HSM

Check you have cHSMs available in your u.Trust SE device with the following command

```
> gladm -d IP -u admin -k admin.key chsm-list-slots
  1:  f59ac468-ff9b-482f-897e-097ceba0080a  SecurityServer-SDK  [regular]
-
running
  2:
  3:
  4:
[...]
```

Tip: When creating the container via `chsm-create`, the use of a `SecurityServer-SDK` template is required. Without the `-SDK` mark, the template will not allow the PQ modules to be imported.

4.4 Step4 : Check the cHSM is available

The `csadm` tool can be used to gather the state of the cHSM. Each cHSM is referred to by a device locator; the locator string syntax depends on the type of device (PCIe or LAN appliance). This example assumes a hostname of "IP", and that the first container slot is of interest:

```
> csadm dev=4001@IP GetState mode = Operational Mode
state = INITIALIZED (0x00100004)
temp = --- alarm = OFF
bl_ver = 7.00.0.0 (Model: u.trust Anchor cHSM) hw_ver = 7.00.0.0
uid = c9d4f750 d0841c43 | P C |
adm1 = 3d165f39 6bb24ae7 83b4a1f8 d6e86b4d |= _9k J kM|
adm2 = 53656375 72697479 53657276 65722d53 |SecurityServer-S|
adm3 = 342e3830 2e302e30 00000000 00000000 |4.80.0.0 |
```

You can compare line 'adm1' with the UUID output via the `chsm-list-slots` as a check to ensure you are looking at the correct container instance.

And check you have the authentication credential of this default admin.

```
> csadm dev=port@IP ListUsers
```

Name	Permission	Mechanism
ADMIN	22000000	RSA sign

If you plan to use PKCS11, you need to create a PKCS11 User for a PKCS11 Slot.

```
set CRYPTOSERVER=4001@IP
```

```
csadm LogonSign=ADMIN, ./ADMIN_CAAK.key \
    AddUser=SO_0000,00000200{CXI_GROUP=SLOT_0000},hmacpwd,87654321
```

```
csadm LogonPass=SO_0000,87654321 ChangeUser=SO_0000,ask
```

```
Enter new passphrase: ...
```

```
Repeat new passphrase: ...
```

```
csadm LogonSign=ADMIN, ./ADMIN_CAAK.key \
    AddUser=USR_0000,00000002{CXI_GROUP=SLOT_0000},hmacpwd,87654321
```

```
csadm LogonPass=USR_0000,87654321 ChangeUser=USR_0000,ask
```

```
Enter new passphrase: ...
```

```
Repeat new passphrase: ...
```

```
p11tool2 ListSlots=Status
```

Once the 2 pkcs11 are created the output should change to :

```
> csadm dev=port@IP ListUsers
```

Name	Permission	Mechanism	Other
ADMIN	22000000	RSA sign	Z[0]I[0]
SO_0000	00000200	HMAC passwd	Z[0]I[0]A[CXI_GROUP=SLOT_0000]
USR_0000	00000002	HMAC passwd	Z[0]I[0]A[CXI_GROUP=SLOT_0000]

If you plan to use the Java_UTI methodology, you need to create a standard “Cryptographic User” using csadm or CAT.

```
csadm LogonSign=ADMIN, ./ADMIN_CAAK.key \
    AddUser=aCryptoUser,2{CXI_GROUP=myGroup},hmacpwd,87654321
```

```
csadm LogonPass=aCryptoUser,87654321 ChangeUser=aCryptoUser,ask
```

```
Enter new passphrase: ...
```

```
Repeat new passphrase: ...
```

The example source that demonstrates use of the Java_UTI layer, creates an application that uses -p or -s at the command line, to supply one or more users to log in on the session:

```
Java -jar quantumprotect.jar -dev 3301@127.0.0.1 -p aCryptoUser,12345678 -
everything
```

4.5 Step 5 : Check that the cHSM has an MBK

The Master Backup Key (MBK) is the master key used to encrypt-for-export working keys and secrets used by the cHSM, specifically those which should only be available to the cHSM (or to another cHSM with the same MBK). MBKs are usually managed via key ceremony. Ensure you have backups of MBK shares in encrypted keyfiles or on smartcards.

When a cHSM is instantiated, it has a default MBK named "AUTO-GEN". That MBK must be replaced. When an MBK is replaced, if the existing MBK is this "AUTO-GEN" key, it will be moved from MBK slot 3 (the active slot) to slot 7. When done correctly, this shows an example of what should be seen via the MBKListKeys command:

```
> csadm dev=port@IP logonSign=ADMIN,./admin.key MBKListKeys
```

slot	name	len	algo	type	k	generation	date	key check value
3	mbk	32	AES	XOR	2	2023/09/27	07:42:07	e3de62f1bf3e5612:67f8026fee58bb85
7	AUTO-GEN	32	AES	SHARE	1	2024/09/06	14:16:51	f999c685bf57ad75:d9b49364dea6c8fe

Guidance on creating and importing MBKs can be found in:

- [u.trust_Anchor_csadm_Manual.pdf](#)
- [CryptoServer_CAT_Manual.pdf](#)

4.6 Step 6 : Configure the cHSM for Quantum Protect

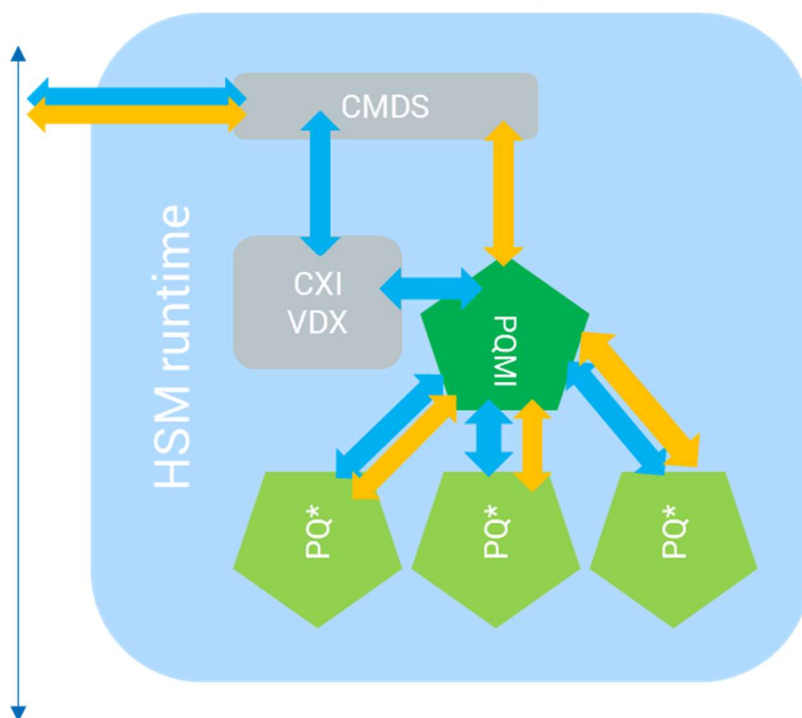
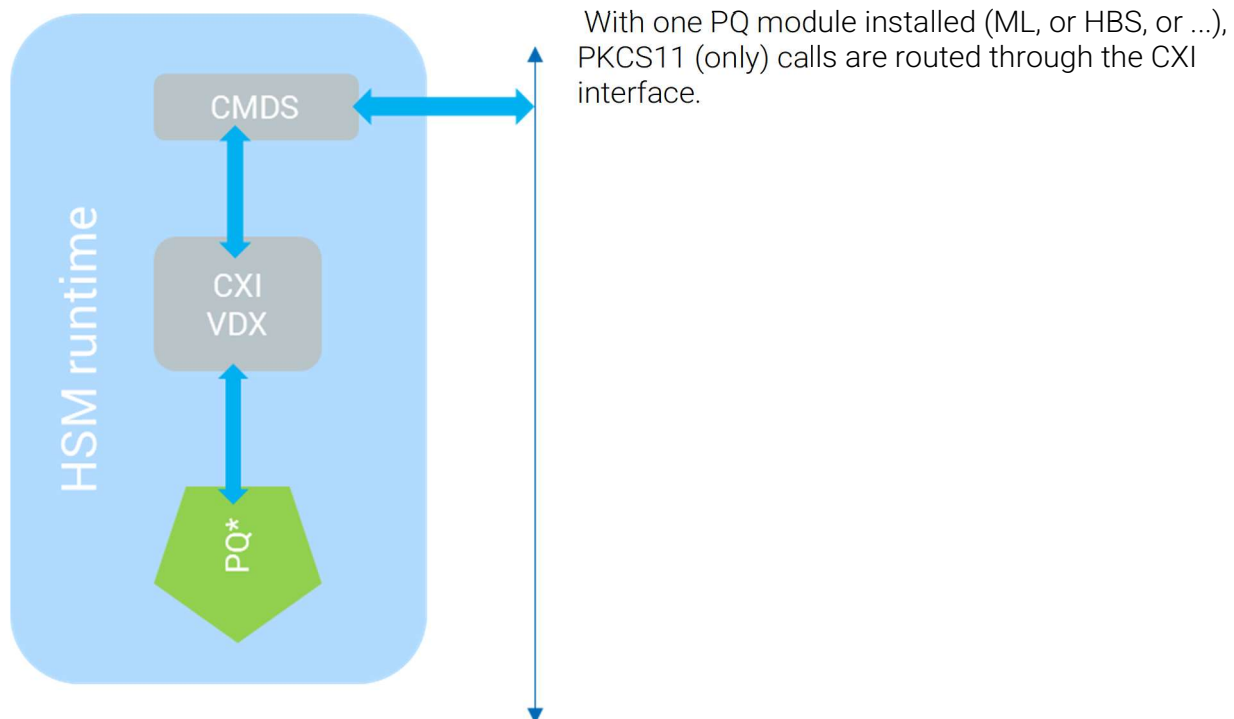
To allow the cHSM or simulator to work with post-quantum algorithms, the runtime must include the supporting modules.

The product currently includes three modules that support PQC.

- PQMI – Post Quantum Module Interface
- ML – Module Lattice (MLDSA, MLKEM)
- HBS – Stateful Hash-based Signatures (LMS, HSS, XMSS, XMSS-MT)

PQMI is different from the algorithm implementations, in that it does not provide any new cryptographic (algorithm) endpoint. Internally, the algorithm modules themselves provide their own PKCS11 vendor-defined mechanism interface, but do not provide an 'external' interface (ie, "sub-function code" interface) as used by the host-HSM protocols.

PQMI provides this external sub-function interface where appropriate, as such it acts as a router between the host and the algorithms.



If you need multiple PQ modules (ML and HBS and ...), you also need to have the PQMI module installed. In this case, normal sub-function access (yellow path) is enabled, as well as multiple PKCS11 access (blue path).

Installing PQMI after the fact is supported. If you start with ML or HBS and later determine a need for the other, you can install both the new algorithm module(s) and PQMI, and restart the cHSM/Simulator.

If you are loading the modules manually, use `csadm LoadFile` to load them. `LoadFile` can be repeated on the same command line, allowing the operator to load the modules in a single command, which reduces the need for repeated administrator logins.

Ensure that you have the correct firmware directory, based on the target runtime (`sim5_windows` for the Windows HSM Simulator, `sim5_linux` for the Linux HSM Simulator, or `uta` for the u.trust Anchor General Purpose HSM).

Files loaded do not get linked into the runtime until reboot, so terminate the command with a Restart. To verify that the modules are loaded and linked into the runtime correctly, see the output from csadm GetBootLog.

```
csadm <auth> LoadFile=hbs.mtc LoadFile=ml.mtc LoadFile=pqmi.mtc \  
Restart  
  
> csadm <dev> GetBootLog  
  
...  
25.02.26 18:35:37 ML: VDM using MUX registry  
25.02.26 18:35:37 module 0xa5 (ML) initialized successfully  
25.02.26 18:35:37 HBS: VDM using MUX registry  
25.02.26 18:35:37 module 0xa2 (HBS) initialized successfully  
25.02.26 18:35:37 module 0xa0 (PQMI) initialized successfully  
...  
>
```

Note: If you see “VMD using **CXI** registry” more than once, you have multiple algorithm modules installed, but PQMI is not installed. You must fix this before trying to use the HSM. “VDM using **MUX** registry” multiple times is not an issue.

WARNING: If there are other modules installed which use the CXI entry point for registration of PKCS11 VDMs (oscca, some others), they are unaware of the PQMI module and so will not use it. In this case there will be a conflict for the CXI entry point, and the system will not work. Remove oscca and any other custom firmware that uses CXI for VDMs. Alternately, if you are the author of the custom firmware, contact Utimaco for guidance on how to check for, and register with PQMI if it is there, and then your module will coexist cleanly.

Note: The firmware modules are signed using the Utimaco production signing key. This is different from the pre-release versions. The pre-release versions required that the public part of the module signing key be manually pre-loaded into the HSM, in order for it to validate the signature on the module transport containers (.mtc files).

As the production public key is preloaded on the HSM during manufacture, this additional step of loading the signing public key is not needed.

5 Managing HSM Backups

There are different methods for backing up the contents of a cHSM.

- (Container Admin) Use `csadm` to backup the various databases, encrypted under the MBK,
- (Operator) Use `gladm` to snapshot the cHSM (not for SHBS containers), encrypted under the operator secret

As noted throughout this document, the standard “backup and restore” methodology used for both classical algorithm keys (RSA, AES, ECC) and stateless PQ keys (MLDSA, MLKEM) is not suitable for use with stateful keys (LMS, XMSS and their multi-tree variants).

Please refer to recommendations and commands described in the regular user documentation. The targets for backup and restore are:

- `csadm BackupUserDatabase (csadm RestoreUserDatabase)`
- `csadm BackupDatabase=pqmi.db (csadm RestoreDatabase=pqmi.db)`
- `csadm BackupDatabase=CXIKEY.db (csadm RestoreDatabase=CXIKEY.db)`

You can not use backup/restore on the `hbs.db`, as it is blocked. It contains stateful information that must not be replicated across cHSMs, except by use of the state management tool, *sfhsmgr* (Stateful-Hash State Manager).

6 Configure Quantum Protect via PKCS11

Quantum protect support the pkcs#11 interface and therefore support internal and external keystore as per Utimaco standard product.

Please refer to regular user documentation to configure the pkcs#11 keystore (internal or external accordingly)

6.1 ML Keys

ML keys (MLDSA, MLKEM) are created using `C_GenerateKeyPair`. `C_FindObjects` can retrieve object handles for both/either of the public part, or private part.

6.2 HBS Keys

HBS keys (LMS, HSS, XMSS, XMSS-MT) are created using `C_GenerateKey`. `C_FindObjects` is used to retrieve an object handle for the key. Here, however, the handle points to a reference index, which will be used to load the actual key from an internal database. In this way, accidental proliferation of the key material is avoided.

This process still works whether the PKCS11 provider is configured for internal or external keystore – what is held is a reference to the actual key. The reference itself is transferable, however it does not take the key information along with it. See the section on [SHBS State Management](#) for how to safely and securely replicate key material for HBS keys.

Using External Keystores (Typed Interfaces)

The Java-UTI infrastructure includes a dedicated internal keystore. A key can be created with the 'External' flag, however. In this case, the entire key is returned as a Key Blob to the caller. A Key Blob is encrypted using the cHSM's Master Backup Key, which allows the key to be used on any cHSM with the MBK available to it, assuming it also has the ML module installed.

These blobs can be stored in any way, suitable to your use case.

7 Algorithm Support

This section lists the supported algorithms, and includes implementation details and supported APIs.

For PKCS11 and C usage, please see the various source and headers in `.../Crypto_APIS/PKCS11_R3/samples/qptool2/src` and `../include`.

For Java_UTI usage, please see the various java files in `.../Crypto_APIS/Java_UTI/samples/typed_interfaces/src/com/utimaco/cs2/mdl/share` and `.../pqmi`.

7.1 PQMI sub-function interface support:

- MLDSA – module ML: Generate, sign, verify
 - Limitations: Payloads are limited to 256Kb for sign/verify
 - Pre-hash is supported.
- MLKEM – module ML: Generate, key encapsulate, key decapsulate
- LMS – module HBS: Generate, sign, verify
 - Limitations: Keys with $h > 15$ can result in timeouts. See below.
- HSS – module HBS: Generate, sign, verify
 - Limitations: Keys with $h > 15$ can result in timeouts. See below.
- XMSS, XMSS-MT: Generate, sign, verify
 - Limitations: Keys with $h > 15$ can result in timeouts. See below.

7.2 PKCS11 Vendor Defined Mechanism support:

- MLDSA – module ML: Generate, sign, verify, private key export (AES-KWP wrapped), public key export
 - Limitations: Payloads are limited to 256Kb for sign/verify
 - Pre-hash is supported.
- MLKEM – module ML: Generate, key encapsulate, key decapsulate, private key export (AES-KWP wrapped), public key export
- LMS – module HBS: Generate, sign, verify, public key export
 - Limitations: Keys with $h > 15$ can result in timeouts. See below.
- HSS – module HBS: Generate, sign, verify, public key export
 - Limitations: Keys with $h > 15$ can result in timeouts. See below.

- XMSS, XMSS-MT: Generate, sign, verify, public key export
 - Limitations: Keys with $h > 15$ can result in timeouts. See below.

Any key generation with larger keys may take longer than 15 minutes, and for very large keys this can durations best measured in days.

There is a hardwired timeout system wide at 15 minutes that will result in cancellation of all sessions on the cHSM. Note that the key is still being generated, even though the cHSM is not responding. It is safe to ignore the timeout, and just let the cHSM run to completion.

Because stateful keys may only be internal, the key generation finishes and the keys are loaded into the internal keystore, and so are visible using PKCS_11 C_FindObjects, or for keys generated using the Java_UTI interface, via the Java_UTI keystore list function.

7.3 C Typed Interfaces

Typed interface implementation files (“_imp.c files”) and headers provide a simple and convenient way to serialize structures into byte buffers for consumption by the cHSM.

The relevant fields are supplied via a typedef’d struct definition, and methods are then provided to compute the serialized length of the data on the wire, and various ways to serialize the data for the wire, depending on allocation strategies.

This version of the implementation files does not support deserialization; in all cases described the return value from the call is either an object handle, a raw byte array (signature, etc) or a CKE error value.

7.4 Java Typed Interfaces

Typed interface classes are provided to simplify the serialization and deserialization of byte buffers which need to be sent to, or received from the cHSM. Each distinct interface required by a sub- function has a matching typed interface.

The fields in the byte buffer are expressed as instance variables, and are supported by constructors, getters and setters. Java code is expected to populate the instance variables with relevant data, and then the instance is ‘executed’, to serialize and send the data in the correct format, as expected by the receiver.

When the cHSM returns from a call, the serialized data may also be represented by a typed interface, in which case a new instance can be created by passing in the serialized data, returned by a previous call, to the constructor of the relevant typed interface.

The Typed Interfaces are of the pattern, *instantiate, populate, execute*.

Instantiate:

Done using the standard Java process. Constructors are supplied for empty instances (the default constructor), and with parameters that match to the fields of the instance. There are also variants that allow for storing a cxi object with the instance, for later use via the `exec` method.

- `<TI> ()`
 - Generic constructor.
- `<TI> (CryptoServerCXI _cxi)`
 - Stores a copy of a `_cxi` instance for later use.
 - Allows you to target a specific cluster with this instance.
- `<TI> (CryptoServerCXI _cxi, int mid, int sfc)`
 - Stores a copy of a `_cxi` instance for later use.
 - Also provides a way to link this instance to a specific module ID and sub-function code (it's possible that a single interface can work with multiple SFCs)
- `<TI> (CryptoServerCXI _cxi, byte[] _in)`
 - Stores a copy of a `_cxi` instance for later use.
 - Takes as input a serialized instance of this `<TI>`. The input is deserialized and the instance is populated with the extracted data.
 - Useful when the return information from the cHSM is a typed interface.
- `<TI> (byte[] _in)`
 - Takes as input a serialized instance of this `<TI>`. The input is deserialized and the instance is populated with the extracted data.
 - Useful when the return information from the cHSM is a typed interface.
- `<TI> (byte[] _in, boolean tf)`
 - Takes as input a serialized instance of this `<TI>`. This constructor assumes that the data provided may extend beyond the requirements of the deserialized instance.
 - The input is deserialized and the instance is populated with the extracted data, ignoring any bytes past what is needed for this instance.
 - Useful when the byte array is a *sequence* of the TI.
- `<TI> (parameters by field list)`
 - These will be `<TI>` specific.
 - In the case where a Typed Interface description lists a "packed" instance of another TI, the parameters-by-field-list also accept the TI object instance in that location.
For example, `MLDSA_KeyGen`'s `3d` field is a packed `CxiKeyAttributes` instance, so a constructor exists for `(int, int, byte[], byte[])`, and `(int, int, CxiKeyAttributes, byte[])`. On use, the constructor will serialize the parameter and store a local (serialized) copy of the instance.
-

Populate:

The instance fields can be populated during construction, deserialization or in code.

```
MyTypedInterface mti = new MyTypedInterface(0,5,aString);
MyTypedInterface mti2 = new MyTypedInterface();
mti2.flags(0);
mti2.spacer(5);
mti2.info(aString);
```

If the response from the CHSM is also a typed interface, you can deserialize it manually using the `<TI>.deserialize(...)` method, or you can pass the TI expected in the `exec` command.

```
FooRespTI oof = new FooRespTI();
boolean opr2 = foo.exec(_cxi, mid, sfc, oof);
// if unsuccessful, throws
// if successful, the 'oof' instance is populated.
// Alternately, this works too:

FooRespTI oof2 = new FooRespTI(foo.resp);
// oof and oof2 are equivalent.
```

Execute:

A populated instance can be executed against a `CryptoServerCXI` instance or cluster:

```
MyTypedInterface mti = new MyTypedInterface(0,5,aString);

try {
    mti.exec(<cxi_instance>, <module_id>, <subfunction code>);
} catch (IOException io) {
    ...
} catch (CryptoServerException cs) {
    ...
}
```

8 ML-DSA

ML-DSA is based on the FIPS 204 requirements. If you need the “Round 3” (IPD) parameter set implementation of Lattice (CRYSTALS-Dilithium or CRYSTALS-Kyber) for a legacy project, please contact Utimaco.

This MLDSA implementation supports key sizes of “security level” 2, 3 and 5. Canonically, these are 4x4, 6x5 and 8x7 matrices.

The implementation does not support ‘chunking’ of data to be signed. For data over 256Kb (with protocol overhead), use the ‘ph’ (pre-hash) variation.

8.1 Key Types

For PKCS11 use, two representations of key-type are supported in this implementation. Type may be expressed as a simple index (1, 2, 3) for key types 4x4, 6x5 and 8x7, or by using the KT mask for ML keys and a security strength level:

```
#define ML_DSA_KT_FLAG      0x030
#define ML_DSA_44           1
#define ML_DSA_KT_44       ML_DSA_KT_FLAG | 0x02 // 0x32
#define ML_DSA_65           2
#define ML_DSA_KT_65       ML_DSA_KT_FLAG | 0x03 // 0x33
#define ML_DSA_87           3
#define ML_DSA_KT_87       ML_DSA_KT_FLAG | 0x05 // 0x35
```

For Java, only the extended “KT_FLAG” (KeyType) forms are defined:

8.2 Key Generation – MLDSA_KeyGen.java and MLDSA_KeyGen.h/_imp.c

PKCS11 VDM: C_GenerateKeyPair

Mechanism ID: UTL_MECH_MLDSA_GENKEY

- Parameter: struct MLDSA_KEYGEN (MLDSA_KeyGen.h/_imp.c)
 - Set flags to 0 (DRBG) or 1 (PRNG) for source of random
 - Set type to one of ML_DSA_## or ML_DSA_KT_##
 - Remaining fields in MLDSA_KEYGEN are ignored.
- Parameter (alternate form):
 - 1 byte: keytype (uint8)
 - This form assumes PRNG

MLDSA_KeyGen has four instance variables.

flags	int	<p>PKCS11: 0 – Use the DRBG, 1 – Use the PRNG.</p> <p>Java_UTI: Bitwise OR of random number generator, internal/external key store flag, and the internal keystore overwrite flag: Pqmi.MODE_REAL_RND or Pqmi.MODE_PSEUDO_RND and optionally Pqmi.KEY_EXTERNAL If not KEY_EXTERNAL, optionally Pqmi.KEY_OVERWRITE. Defaults are Pqmi.MODE_PSEUDO_RND, and internal storage (without overwrite).</p>
type	int	<p>PKCS11: 1 or 0x32 (4x4), 2 or 0x33 (6x5), or 3 or 0x35 (8x7),</p> <p>Java_UTI: Bitwise OR of key type and size. Pqmi.ML_DSA_KT_FLAG Pqmi.ML_DSA_KT_[44 65 87]. There is no default, type must be set accordingly.</p>
		Continued next page

attributes	byte []	<p>PKCS11: Not used</p> <p>Java_UTI: Expected by the implementation to be a packed CxiKeyAttributes typed interface. Fields of interest are name, group and specifier, which are needed to determine which CXI Key Group the generated key belongs to, for authorized use.</p> <p>The CxiKeyAttributes includes a field mdata, which is (arbitrary) meta data to store with the key. If you need more attributes than what are currently considered, use a CryptoServerCXI.KeyAttributes object, which can be serialized and stored in the mdata field for later host-side interpretation. This field (in CxiKeyAttributes) is otherwise ignored in this release.</p>
seed	byte []	Ignored.

8.3 Signing – MLDSA_Sign.java, MLDSA_Sign.h/_imp.c

Limitations in QP 1.0:

- Signing does not support message sizes over 256Kb.

PKCS11:

PKCS11 VDM: C_Sign

PKCS11 MechanismID: CKR_MECH_MLDSA_SIGN, CKR_MECH_MLDSA_EXTMU_SIGN

Parameter:

- 8 byte array (two uint32 big endian values)
 - 4 bytes: Flags
 - 0 – DRBG, 1 – PRNG (source of random)
 - 4 bytes: Keytype
 - One of ML_DSA_## or ML_DSA_KT_##

Typed Interface:

MLDSA_Sign has six instance variables.

Fieldname	Type	Usage
flags	uint32	Bitwise OR. MLDSA_Sign.FLAG_HAS_CTXT: The signature should include the ctxt (additional authenticated data) field, MLDSA_Sign.FLAG_PRE_HASH: The msg field is a digest of the data, not the data itself, MLDSA_Sign.FLAG_SIG_PACKED: The signature should be returned in a “packed” format, Pqmi.MODE_EXTERNAL_MU: The msg field is calculated mu.
type	uint32	Bitwise OR of key type and size. Pqmi.ML_DSA_KT_FLAG Pqmi.ML_DSA_KT_[44 65 87]. There is no default, type must be set accordingly.
key	byte []	A key handle (internal keystore) or key blob (external keystore) for the signing key.
msg	byte []	The data to be included in the signature
ctxt	byte []	Additional authentication data added to the generation of the signature. If chunking, the ctxt should only be included on the FLAG_INIT payload – this limits the ctxt data to ~256Kb.
state	byte []	If the amount of data to be signed is > ~256Kb, it must be chunked. In this case. Must be NULL with FLAG_INIT, populated with the response from the previous call on FLAG_CONT or FLAG_FINAL.

8.4 Verifying

Limitations in QP 1.0:

- Verifying does not support message sizes over 256Kb.

PKCS11:

PKCS11 VDM: C_Verify

Mechanism ID:

CKR_MECH_MLDSA_VERIFY,

CKM_MECH_MLDSA_VERIFY

Parameter:

- 8 byte array (two uint32 big endian values)
 - 4 bytes: Flags
 - 0 – DRBG, 1 – PRNG (source of random)
 - (unimplemented) Pre-hash flag.
 - 4 bytes: Keytype.
 - One of ML_DSA_## or ML_DSA_KT_##

Typed interface:

MLDSA_Verify has seven instance variables.

Fieldname	Type	Usage
flags	uint32	Bitwise OR. MLDSA_Sign.FLAG_HAS_CTXT: The signature should include the ctxt (additional authenticated data) field, MLDSA_Sign.FLAG_PRE_HASH: The msg field is a digest of the data, not the data itself, MLDSA_Sign.FLAG_SIG_PACKED: The signature should be returned in a “packed” format, Pqmi.MODE_EXTERNAL_MU: The msg field is calculated mu.
type	uint32	Bitwise OR of key type and size. Pqmi.ML_DSA_KT_FLAG Pqmi.ML_DSA_KT_[44 65 87]. There is no default, type must be set accordingly.
key	byte []	A key handle (internal keystore) or key blob (external keystore) for the public key
		Continued next page
msg	byte []	The data to be included in the verification

ctxt	byte []	Additional authentication data needed for verification
state	byte []	NOT SUPPORTED in 1.0. Set to NULL. If the amount of data to be signed is > ~256Kb, it must be chunked. In this case. Must be NULL with (flags) FLAG_INIT, populated with the response from the previous call on FLAG_CONT or FLAG_FINAL.
sig	byte []	The signature to be verified.

8.5 Key Wrap – MLXXX_KeyWrap.h/_imp.c

PKCS11 VDM: C_WrapKey

Mechanism ID: CKM_MECH_MLDSA_WRAP_AESKWP

- Parameter: struct MLXXX_KEYGEN (MLXXX_KeyWrap.h/_imp.c)
 - Set flags to ML_MODE_STORE_SEED or ML_MODE_STORE_SK
 - Set type to one of ML_DSA_##

Used to export private key or key generation seed. Only available for PKCS11 VDM.

Only AES wrapping keys are supported in this version.

Field name	Type	Usage
flags	int	PKCS11: ML_MODE_STORE_SEED – export seed used for key generation (key can be recreated from seed) ML_MODE_STORE_SK – export private key
type	int	PKCS11: 1 or 0x32 (4x4), 2 or 0x33 (6x5), or 3 or 0x35 (8x7),

Result of exporting

DB content	Export flag	Status
SEED	any	OK
SK	ML_MODE_STORE_SK	OK
SK	ML_MODE_STORE_SEED	Not available

8.6 Key Unwrap – MLXXX_KeyWrap.h/_imp.c

PKCS11 VDM: C_UnwrapKey

Mechanism ID: CKM_MECH_MLDSA_UWRAP_AESKWP

- Parameter: struct MLXXX_KEYGEN (MLXXX_KeyWrap.h/_imp.c)
 - Set flags to ML_MODE_STORE_SEED or ML_MODE_STORE_SK
 - Set type to one of ML_DSA_##

Used to import private key or key generation seed. Only available for PKCS11 VDM.

When private key (SK) is imported, public key must be imported separately.

Only AES wrapping keys are supported in this version.

Field name	Type	Usage
flags	int	PKCS11: ML_MODE_STORE_SEED – store seed to key database ML_MODE_STORE_SK – store private key to key database ML_MODE_STORE_SEED ML_MODE_STORE_SK – store both
type	int	PKCS11: 1 or 0x32 (4x4), 2 or 0x33 (6x5), or 3 or 0x35 (8x7),

Result of importing

Wrapped key content	Import flag	Status
SEED	any, including ML_MODE_STORE_SK ML_MODE_STORE_SEED	OK
SK	ML_MODE_STORE_SK	OK
SK	ML_MODE_STORE_SEED	Not available

9 ML-KEM

ML-KEM is based on the FIPS 203 requirements.

This MLKEM implementation supports key sizes of “security level” 2, 3 and 5. Canonically, these are key sizes 512, 768 and 1024.

9.1 Key Types

Two representations of key-type are supported in this implementation. Type may be expressed as a simple index (1, 2, 3) for key types 4x4, 6x5 and 8x7, or by using the KT mask for ML keys and a security strength level:

```
#define ML_KEM_KT          0x40

#define ML_KEM_512          1
#define ML_KEM_KT_512      ML_KEM_KT | 0x02      // 0x42

#define ML_KEM_768          2
#define ML_KEM_KT_65        ML_KEM_KT | 0x03      // 0x43

#define ML_KEM_1024         3
#define ML_KEM_KT_87        ML_KEM_KT | 0x05      // 0x45
```

9.2 Key Generation – MLKEM_KeyGen.java and MLDSA_KeyGen.h/_imp.c

PKCS11

PKCS11 VDM: C_GenerateKeyPair

Mechanism ID: UTI_MECH_MLKEM_GENKEY

Parameter:

struct MLKEM_KEYGEN (MLKEM_KeyGen.h/_imp.c)

- Set flags to 0 (DRBG) or 1 (PRNG) for source of random
- Set type to one of ML_KEM_## or ML_KEM_KT_##
- Remaining fields in MLKEM_KEYGEN are ignored.
- Parameter (alternate forms):
 - 8 byte array:
 - uint32 set to source of random, as above

- uint32 set to keytype, as above

Java Typed Interfaces:

Key sizes are defined as public static final integers. The 2, 3, and 5 are security strength levels:

```
public static final int    ML_KEM_KT_FLAG        = 0x040;
public static final int    ML_KEM_KT_512        = ML_KEM_KT_FLAG | 0x2;
public static final int    ML_KEM_KT_768        = ML_KEM_KT_FLAG | 0x3;
public static final int    ML_KEM_KT_1024       = ML_KEM_KT_FLAG | 0x5;
```

A complete Key Type value is ML_KEM_KT_FLAG | ML_KEM_KT_##, giving one of 0x42, 0x43 or 0x45.

MLKEM_KeyGen has four instance variables.

Fieldname	Type	Usage
flags	uint32	<p>PKCS11: 0 – Use the DRBG, 1 – Use the PRNG.</p> <p>Java_UTI: Bitwise OR of random number generator, internal/external key store, and the internal keystore overwrite flag. Pqmi.MODE_REAL_RND or Pqmi.MODE_PSEUDO_RND and optionally Pqmi.KEY_EXTERNAL If not KEY_EXTERNAL, optionally Pqmi.KEY_OVERWRITE. Defaults are Pqmi.MODE_PSEUDO_RND, and internal storage (without overwrite).</p>
type	uint32	<p>PKCS11: 1 or 0x42 (512), 2 or 0x43 (768), or 3 or 0x45 (1024), Java_UTI: Bitwise OR of key type and size. Pqmi.ML_KEM_KT_FLAG Pqmi.ML_KEM_KT_[512 768 1024]. There is no default, type must be set accordingly.</p>

attributes	byte []	<p>PKCS11: Not used</p> <p>Java_UTI: Expected by the implementation to be a packed CxiKeyAttributes typed interface. Fields of interest are name, group and specifier, which are needed to determine which CXI Key Group the generated key belongs to, for authorized use.</p> <p>The CxiKeyAttributes includes a field mdata, which is (arbitrary) meta data to store with the key. If you need more attributes than what are currently considered, use a CryptoServerCXI.KeyAttributes object, which can be serialized and stored in the mdata field for later usage by the host side. This field (in CxiKeyAttributes) is otherwise ignored in this release.</p>
seed	byte []	Ignored.

9.3 Key Encapsulation

Key encapsulation is a three-stage process: Generate, Encapsulate, Decapsulate.

Assume two trading partners, Alice and Bob. They wish to securely exchange an encrypted message, and to do so, will use MLKEM to generate each locally an arbitrary secret.

Alice and Bob both generate their own MLKEM ephemeral

keys. Alice sends her public key to Bob.

Bob uses “Key Encapsulate”, supplying his private key, and Alice’s public key.

The return value of the encapsulation step includes two parts, a generic secret, and an encapsulated message (encapsulated using Alice’s public key).

Bob sends the encapsulated message to Alice.

Alice uses “Key Decapsulate”, supplying her private key, and the encapsulated message.

The return value of the decap step is the same generic secret as received by Bob.

9.3.1 Encapsulate – MLKEM_Encap.java, MLKEM_Encap.h/_imp.c

PKCS11 VDM: C_DeriveKey

PKCS11 Mechanism ID: UTI_MECH_MLKEM_ENCAP

Parameter:

- flags – uint32
 - 0 use DRBG, 1 use PRNG
- type – uint32

- An MLKEM keytype (see below)
- public_key – unsigned char []
 - The target public key

MLKEM_Encap has three instance variables.

Fieldname	Type	Usage
flags	uint4	0 – Use DRBG 1 – Use PRNG
type	uint4	Either index (1, 2, or 3) OR Bitwise OR of key type and size. Pqmi.ML_KEM_KT_FLAG Pqmi.ML_KEM_KT_[512 768 1024]. There is no default, type must be set accordingly.
public_key	byte []	MLKEM Public key of same type as entered in key_type

Return Values:

There are two return values from the Encap step. First, the generic secret is returned as an object handle to the derived key. Second, in the CKA_UTILITY_CUSTOM_DATA (virtual) attribute, you will find the cyphertext. For how to access/use the CKA_UTILITY_CUSTOM_DATA, see the sample code provided in Crypto_APIS/PKCS11_R3/samples.

9.3.2 Decapsulate – MLKEM_Decap.java, MLKEM_Decap.h/_imp.c

PKCS11 VDM: C_DeriveKey

PKCS11 Mechanism ID: UTI_MECH_MLKEM_DECAP

Parameter:

- flags – uint32
 - Not used. Set to 0
- type – uint32
 - Keytype (see below)
- cyphertext – unsigned char []
 - See below

MLKEM_Decap has three instance variables.

Fieldname	Type	Usage
flags	uint4	Not used. Set to 0.
type	uint4	Either index (1, 2, or 3) OR Bitwise OR of key type and size. Pqmi.ML_KEM_KT_FLAG Pqmi.ML_KEM_KT_[512 768 1024]. There is no default, type must be set accordingly.
cyphertext	byte []	Cyphertext returned from an encapsulate step.

Return value: The generic secret is returned via the object handle of the derived key.

10 LMS and HSS

Implementation based on the SP800-208 specification, which follows RFC8554 with some narrowing limitations.

With regards to key exportability, Version 1.0 of Quantum Protect **does** follow the current SP800-208 specification, which it does not permit.

10.1 Key Provisioning/Apportionment

The SP800-208 specification will be adjusted to allow key provisioning.

In a trust relationship between multiple cHSMs, it will be possible to apportion LM-OTS key blocks between the cHSMs, providing for high-availability, fault-tolerant and performance improvement by allowing multiple HSMs to sign in the name of the LMS/HSS key. This also provides a mitigation against the loss of a cHSM; instead of losing the entire key, only the LM-OTS keys apportioned to that cHSM are lost.

The provisioning available in QP 1.0 can be provided on request, however the current implementation is being reworked to match the upcoming draft of the to-be-reissued SP800-208 specification. The reworked implementation is expected to be released in QP 1.1

10.2 Introduction

LMS (Leighton-Micali Hash-Based Signatures), and its multi-tree variant, HSS (Hash-based Signature Scheme) are stateful. In either form, there is one (LMS) or more (HSS) Merkel Trees, and each leaf node is a Leighton-Micali scheme One-Time-Signature (LM-OTS) signing key. The Merkel tree nodes are intermediate values used for signature validation.

Every LM-OTS key may be used exactly once. Re-use of any key invalidates the signature guarantees for the tree itself; the module must take extra care to ensure that the cHSM implementation does not re-use any one LM-OTS key.

Support for LMS/HSS key generation is available via PKCS11 Vendor Defined Mechanisms, and via the Java_UTI typed interface methodologies.

Vendor defined mechanisms are an extension to standard cryptographic functionality of HSM. A host application can use the standard cryptographic API calls (CXI, PKCS#11) to access functions, which were not available when API was designed.

10.3 Example Source

Example source code for PKCS11 is found in `.../Crypto_APIs/PKCS11_R3/samples`.

Example source code for typed interfaces is found in `.../Crypto_APIs/Java_UTI/samples/typed_interfaces`.

In both cases, the example source may be compiled and run to demonstrate the techniques.

10.4 Key Generation

This mechanism is used to create a top level LMS, HSS, XMSS or XMSS-MT private key.

For PKCS11, a persistent keypair *reference* is stored in the CXI database and can be referenced normally, ie via C_FindObjects.

For the typed interfaces, keys are accessed via Name, Group and Specifier, and internal storage is provided by the pqmi.db database. The pqmi.db holds the key reference (similar to the keypair reference used by PKCS11).

The actual private key and its associated state are stored in the hbs.db database, internally on the HSM.

External key storage is not available for stateful keys.

Host application cryptographic API calls should use these parameters:

Cryptographic operation	Generate key (C_GenerateKey)
Key algorithm	CUSTOM
Key type	SECRET
Mechanism	HBS_MECH_GENKEY
Mechanism parameters	See below

Mechanism parameters (LMS/HSS)

RNG type 1 byte	LMS/OTS levels 1 byte	Array of level parameters 2 bytes each level	Auxiliary data size 2 bytes
--------------------	--------------------------	---	--------------------------------

Field	Length	Description
RNG type	1 byte	Type of random number generator <ul style="list-style-type: none"> - HBS_RNG_TYPE_PSEUDO - HBS_RNG_TYPE_REAL
Levels	1 byte	LMS/OTS: Number of LMS/OTS levels. Valid values are 1 – 8. Each level will have an LMS and an OTS parameter in the Level Parameters field.

		XMSS/XMSS-MT: Set to 0 if XMSS OID follows. Set to 1 if XMSS-MT OID follows..
Level Parameters	2 bytes each level	<p>LMS/OTS parameters pairs. The first byte or the pair is the LMS parameter and the second byte is OTS. There must be one pair per level (1 – 8)</p> <p>LMS options:</p> <ul style="list-style-type: none"> - LMS_SHA256_N32_H5 - LMS_SHA256_N32_H10 - LMS_SHA256_N32_H15 - LMS_SHA256_N32_H20 - LMS_SHA256_N32_H25 - LMS_SHA256_N24_H5 - LMS_SHA256_N24_H10 - LMS_SHA256_N24_H15 - LMS_SHA256_N24_H20 - LMS_SHA256_N24_H25 <p>OTS options</p> <ul style="list-style-type: none"> - LMOTS_SHA256_N32_W1 - LMOTS_SHA256_N32_W2 - LMOTS_SHA256_N32_W4 - LMOTS_SHA256_N32_W8 - LMOTS_SHA256_N24_W1 - LMOTS_SHA256_N24_W2 - LMOTS_SHA256_N24_W4 - LMOTS_SHA256_N24_W8 <p>XMSS/XMSS-MT: The OID (1 byte) of the desired scheme. SHAKE OIDs are not supported in this release.</p>
Auxiliary data size	2 bytes	<p>LMS/HBS: Size of the auxiliary data space. Valid numbers are 1 – 16384. The auxiliary data will be allocated at the key generation and filled with the precalculated values. Generating signatures will be significantly faster with an appropriate size of this value. On the other hand, this reduces the space in the key database.</p> <p>The optimal value for LMS_SHA256_N32_H10 / LMOTS_SHA256_N32_W4 is 10916 bytes.</p> <p>XMSS/XMSS-MT: Not configurable (optimum size is provided internally)</p>

10.5 Public Key retrieval

This mechanism exports the public part of LMS/OTS or XMSS keypair. Export is done in two steps. First, the new key object is derived from the existing key. Then CKA_UTI_CUSTOM_DATA attribute is retrieved from the new object, which contains the public key.

Host application cryptographic API calls should use these parameters:

Cryptographic operation	Derive key (C_DeriveKey)
Mechanism	HBS_MECH_GET_PUBKEY
Mechanism parameters	None

10.6 Sign

This mechanism is used to create an LMS/OTS signature. It supports both signature methods – single block of data (C_SignInit, C_Sign) and multiple data blocks (C_SignInit, C_SignUpdate, C_SignFinal). Each signature updates the private key.

Note: Do not use concurrent signatures calls because the result is unpredictable.

Host application cryptographic API calls should use these parameters:

Cryptographic operation	Sign (C_SignInit, C_Sign,...)
Mechanism	HBS_MECH_GET_PUBKEY
Mechanism parameters	None

10.7 Verify

This mechanism is used to verify an LMS/OTS signature. It supports both verification methods – single block of data (C_VerifyInit, C_Verify) and multiple data blocks (C_VerifyInit, C_VerifyUpdate, C_VerifyFinal).

Host application cryptographic API calls should use these parameters:

Cryptographic operation	Verify (C_VerifyInit, C_Verify,...)
Mechanism	HBS_MECH_GET_PUBKEY
Mechanism parameters	None

11 Create Multiple cHSMs from a snapshot (Non-SHBS algorithms)

The u.trust Anchor Se GP HSMs, depending on licensed version, multiple containers, each a separate cHSM runtime. These cHSMs can be run in 'regular' mode (mutable/modifiable) or 'cluster' mode (immutable).

1. Create a cHSM in 'regular' mode, using the SecurityServer-SDK template:
 - a. `gladm chsm-create -h`
2. Populate the cHSM with an MBK, users, the QP firmware:
 - a. `csadm Help=MBKGenerateKey`
 - b. `csadm Help=MBKImportKey`
 - c. `csadm Help=AddUser`
 - d. `csadm Help=LoadFile`
3. If using internal keys, generate the keys (PKCS11 or Typed Interfaces)
 - a. `.../Crypto_APis/PKCS11_R3/samples/qptool2`
 - b. `.../Crypto_APis/Java_UTI/samples/typed_interfaces`
4. Snapshot the container:
 - a. `gladm chsm-snapshot -h`
5. Clone the container:
 - a. `gladm chsm-clone -h`

WARNING: This methodology must NOT be used for stateful algorithms. Stateful algorithms must NOT be run on containers in 'cluster' mode, as each instance will have the same state at the beginning, invalidating the security guarantees of the algorithm.

12 SHBS State Management

Stateful algorithms do not benefit from the standard methodologies for backup/restore, or for container snapshot/clustering.

Using those techniques would remove the security guarantees around the algorithms, due to the copy/proliferation of the unique state of the key at the time of the backup/snapshot. All restored versions, whether backup/restore or snapshot/cluster or snapshot/restore, have that same initial state, and so any signatures generated from that point forward, on any of the cHSMs, would start from that same state.

Withing a trust relationship between multiple cHSMs, however, LM/OTS key state can be provisioned safely.

Key provisioning is the process of replicating the Private and Public key information *without* any state to the devices in the trust relationship. As each of the recipients of the key information has no state, they can't sign anything, protecting the security guarantees.

Once the private key information is available on the receiver(s), the generator host can be instructed to create a second state instance, with a range of LM/OTS keys. This new state is effectively *subtracted* from the state on the provider instance. The new state is then exported from the HSM as an encrypted blob with a defined destination device.

The destination device loads the new state into its own environment and can then use it to sign. Because the LM/OTS range was removed from the provider instance, there is no chance that the same LM/OTS keys will be used by different devices.

Note that a private key may hold multiple instances of LM/OTS state objects, which it will use in turn until each is exhausted.

Take as an example an H=5 LMS key (32 signatures). On key generation, the generator instance has a state that is aware of 32 unused LM/OTS keys, and that the next key to use is at index '0' (LM/OTS indexes 0-31). The generator is then instructed to provide 8 LM/OTS keys to a recipient device, by trust-relationship name. The generator creates a second state object, that is aware of 8 unused LM/OTS keys, and that the next key to use is at index '0' (LM/OTS indexes 0-7). As part of this step, the generator advances its index to the next LM/OTS index after the apportionment. Its own index now provides for indexes 8-31.

The second state object is encrypted using a key based partly on the recipient's trust relationship identifier.

Note that with this process, the recipient can now also be a generator; it's key space (0-7) can be further separated for re-apportionment to some third device, or back to the first device. This latter case may be because a certain number of keys were expected, but only some were used. The unused portion may be reassigned back to the original generator.

Version 1.0 of QP does support key provisioning, however with upcoming changes to SP800-208, as signaled by NIST, the version we provide may not correctly map to the drafted changes, likewise the drafted changes may themselves be altered over time.

Utimaco can provide a state management tool based on the current implementation, on request, however it can not be guaranteed as acceptable in a FIPS or SP800-208 sense. For this reason, it should only be used in a QA or development environment while building out infrastructure, it should not be used in a production environment.

13 Known Issues

Quantum Protect 1.0

Code Support/Basis of the implementation

Quantum Protect includes operational code that works around design limitations in the base firmware. When those limitations are removed from the GA code base, Quantum Protect will be modified to make use of those updates.

UCAPI

UCAPI is the underlying native cryptographic library between host code and what the cHSM uses. It was first released as part of the PKCS11_R3 release (4.40). Over time, the different providers (EKM, JCE, etc) have been migrated away from CXI to UCAPI. Both PKCS11_R3 and UCAPI, however, must be modified to accept the PQ algorithms, keys and techniques, which is ongoing. When Oasis releases PKCS11 v3.2 (support for PQ), and when UCAPI is updated, the following will change:

- PKCS11_R3 Vendor defined mechanisms change to the PKCS11 v3.2 standardized mechanisms. As of QP 1.0, this standard is not yet published. This change will entail both CKM value, as well as the mechanism parameter changes. It will, however, be a simplification for user code, as the user code will no longer need to manually serialize the mechanism information.
- Java_UTI support will be deprecated, in place of standard UCAPI/CXI calls. For example, instead of MLDSA_KeyGen, the code will call the standard CXI "<cx>.generate_key(...)" command (both C++ and Java CXI), with the necessary flags and information to route it to the same ml_dsa_gen_key() method internally on the HSM. This change will entail a migration and so will be disruptive, however a migration plan/procedure will be provided, detailing the differences. Java/C++ UTI-based implementations should use an existing or new abstraction layer now, rather than having local applications call the typed-interfaces directly. This will facilitate the future migration. The UTI interfaces will be supported for at least one year past the cutover.

PKCS11 and PCIe (Windows) Support

A bug in the device software (hypervisor layer) prevents Vendor Defined Mechanisms from being correctly routed, when using the Windows PCIe driver and a u.trust Anchor PCIe card on a Windows system. Windows PCIe is not supported as a target for PKCS11 in current releases.

PKCS11 and multiple 'vendor-defined mechanism'-using modules

The PKCS11 VDM support on the cHSM is designed for only one single module making use of the VDM registration process. If multiple modules attempt to register with the runtime, 'last one wins'.

To work around this, if multiple algorithm modules (currently ML, HBS) are installed at the same time, the PQMI module must *also* be installed. When PQMI is available, it registers with the runtime, and the algorithm modules will register their VDMs via PQMI. If PQMI is not available, they will each register normally – leading to a conflict.

Note that this also means the existence of other VDM-using modules (BRICKS, OSCCA, 3d party modules that use VDM) on the cHSM will conflict, and so should be removed from the cHSM/Simulator.

If you are a vendor of PKCS11 VDM custom firmware, and expect your module to run next to the Quantum Protect modules, please contact Utimaco Professional Services. PS will provide guidance on migrating your module to use of PQMI. Alternately, wait until the underlying issue is addressed in the base firmware.

14 Release Notes

2025-08-20 (QP 1.3)

Changes and Additions:

Support for External Mu

ML-DSA signature has the ability to perform a client-side hash function. A calculated "external mu" is sent to the device instead of the entire message.

Key Export and Import for ML-DSA and ML-KEM

From version 1.3 onwards, the seed from which the key was generated is also written to the key database. When exporting (Key Wrap) and importing (Key Unwrap), it is possible to select the key component to be exported or imported. Key Wrap and Key Unwrap are only supported with the PKCS#11 interface. With Java UTI, export is currently only possible in the form of a Backup Blob, which contains both the seed and the raw private key.

2025-04-30 (QP 1.2)

Changes and Additions:

In general:

A documentation set is supplied as a separate download from the portal. It will in the future be included in the QP download.

Error codes are harmonized. In general, error codes in the 0xB1A0 and 0xB1A6 ranges have been combined and moved into the 0xB0A6 range. For example, 0xB1A60A01 is now 0xB0A60A01. This change was required due to the same module having its ID changed from 0x1A0 to 0x1A6 during development, and then dropped to 0x0A6 for production.

LMS vs HSS:

The HBS module produces HSS structured artifacts, not LMS.

When using the HSS mechanisms in HBS, you get HSS-formatted artifacts.

There are now dedicated "LMS" mechanisms in the PKCS11 VDM implementation.

When used, artifacts are presented in LMS format rather than HSS format.

See `.../Crypto_APIS/PKCS11_R3/samples/qptool2/src/test_case_lms.c` .

XMSS:

Public keys are now output in RFC-correct format, when extracted as raw bytes.

See `.../Crypto_APIS/PKCS11_R3/samples/qptool2/src/test_case_xmss.c` .

MLDSA:

(PKCS11) VDM mechanisms are now available for sign and verify in MLDSA that carry with them the key type:

ML_DSA_MECH_SIGN_4X4
ML_DSA_MECH_SIGN_6X5
ML_DSA_MECH_SIGN_8X7
ML_DSA_MECH_VERIFY_4X4
ML_DSA_MECH_VERIFY_6X5
ML_DSA_MECH_VERIFY_8X7

When you use one of the above, internally the mechanism parameter is assumed to want the pseudo-RNG if needed, and the keytype. As such, the mechanism parameter can be sent in empty. For a 4x4 key:

```
CK_MECHANISM    mechanism = { ML_DSA_MECH_SIGN_4X4, NULL, 0 };
```

is sufficient, instead of

```
unsigned char p_mech[8];  
store_int4(0, p_mech);  
store_int4(ML_DSA_44, p_mech + 4);
```

```
CK_MECHANISM mechanism;  
mechanism.mechanism = CKM_MECH_MLDSA_SIGN;  
mechanism.ulParameterLen = sizeof(p_mech);  
mechanism.pParameter = p_mech;
```

ML Module:

(PKCS11) It is now possible to extract a wrapped private key using C_WrapKey, and import a wrapped secret key using C_UnwrapKey.

The encryption algorithm is AES_KWP and requires an AES256 key that PKCS11_R3 is aware of/previously loaded.

Database listing

(Java_UTI)

It is now possible to list the keys created by the Java_UTI interface layer (and the PQMI module).

In ../Crypto_APIS/Java_UTI/samples/typed_interfaces/src/com/utimaco,
see ../aut/KeyStoreTest.java

Fixed Bugs:

(PKCS11) ML-DSA Verify no longer returns "Invalid Key".

(PKCS11) XMSS now returns an RFC-correct public key when requested.

Known Bugs:

(PKCS11) (LMS) When using a short-hash scheme (those using M24/N24 instead of M32/N32), the output artifacts are still the length of the M32/N32 artifact. The artifact (less the zeroized excess bytes at the end) is correct, just 8 bytes too long.

2025-02-20 (QP 1.0/1.1)

qptool2 test application code:

xmss-test: Added loop counter support for 'sign'

lms-test: Added loop counter support for 'sign'

The goal of the two changes above is to loop on sign, so that the user can watch the 'q' counter value in the signature as it is changed for each signature.

2025-02-20 lms:

Firmware: stray output to bootlog by release version, removed

Appendix A

To Contact Utimaco:

1-800-UTIMACO (US)

Sales: sales@utimaco.com

Support: support@utimaco.com, and <https://support.hsm.utimaco.com>

Contact

Utimaco IS GmbH
Germanusstr. 4
D-52080 Aachen
Germany

Phone: +49 241 1696 – 200

Fax: +49 241 1696 – 199

Web: utimaco.com

E-mail: hsm@utimaco.com