



OPEN

Compute Project

OCP Layered Open-Source Cryptographic Key-management (L.O.C.K.)

Revision 0.5

Version 0.5

CONTRIBUTORS:

Andrés Lagar-Cavilla
(Google)
Amber Huffman (Google)
Charles Kuzman (Google)
Jeff Anderson (Google)
Chris Sabol (Google)
Srini Narayanamurthy (Google)
Lee Prewitt (Microsoft)
Michael Norris (Microsoft)
Eric Eilertson (Microsoft)
Bryan Kelly (Microsoft)
Anjana Parthasarathy (Microsoft)
Ben Keen (Microsoft)
Jisoo Kim (Samsung)
Gwangbae Chio (Samsung)
Eric Hibbard (Samsung)
Mike Allison (Samsung)
Scott Shadley (Solidigm)
Gamil Cain (Solidigm)
Festus Hategekimana (Solidigm)
John Geldman (Kioxia)
Fred Knight (Kioxia)
Paul Suhler (Kioxia)
James Borden (Kioxia)

Revision Table

Date	Revision #	Author	Description
September 2024	0.5	Authoring companies	Initial proposal draft based on work from the list of contributors

License

Open Web Foundation (OWF) CLA

Contributions to this Specification are made under the terms and conditions set forth in Open Web Foundation Modified Contributor License Agreement (“OWF CLA 1.0”) (“Contribution License”) by:

Google, Microsoft, Samsung, Solidigm, Kioxia

Usage of this Specification is governed by the terms and conditions set forth in **Open Web Foundation Modified Final Specification Agreement (“OWFa 1.0”) (“Specification License”)**.

You can review the applicable OWFa1.0 Specification License(s) referenced above by the contributors to this Specification on the OCP website at. For actual executed copies of either agreement, please contact OCP directly.

Notes:

- 1) The above license does not apply to the Appendix or Appendices. The information in the Appendix or Appendices is for reference only and non-normative in nature.

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Acknowledgements

The Contributors of this Specification would like to acknowledge the following companies for their feedback:

Compliance with OCP Tenets

Please describe how this Specification complies to the following OCP tenets. Compliance is required for at least three of the four tenets. The ideals behind open sourcing stipulate that everyone benefits when we share and work together. Any open source project is designed to promote sharing of design elements with peers and to help them understand and adopt those contributions. There is no purpose in sharing if all parties aren't aligned with that philosophy. The IC will look beyond the contribution for evidence that the contributor is aligned with this philosophy. The contributor actions, past and present, are evidence of alignment and conviction to all the tenets.

Openness

OCP L.O.C.K source for RTL and firmware will be licensed using the Apache 2.0 license. The specific mechanics and hosting of the code are work in progress due to CHIPS alliance timelines. Future versions of this spec will point to the relevant resources.

Efficiency

OCP L.O.C.K. is used generate and load keys for use of encrypting user data prior to storing data at rest and decrypting stored user data at rest when read. So, it cannot yield a measurable impact on system efficiency.

Impact

OCP L.O.C.K. enables consistency and transparency to a foundational area of security of media encryption keys such that no firmware in the device ever has access to a media encryption key. Furthermore, no decrypted media encryption key exists in the device when power is removed from the device.

Scale

OCP L.O.C.K. is a committed intercept for Cloud silicon for Google and Microsoft. This scale covers both a significant portion of the Cloud market in hyperscale and enterprise.

Sustainability

The goal of OCP L.O.C.K. is to eliminate the need to destroy storage devices (e.g., SSDs) in the Cloud market by providing a mechanism that increases the confidence that a media encryption key within the device is deleted in a crypto-erase. This enables repurposing the device and or components on the device at end of use or end of life. Given the size of the Cloud market this provides a significant reduction of e-waste.

Table of Contents

This specification is to become a subdivision of the set of separate specifications covering the span of the Caliptra project

Introduction	8
Background	9
OCP L.O.C.K. goals.....	9
Threat model.....	10
High level architecture	11
MEK derivation.....	13
PMEK lifecycle	16
PMEK Generation.....	17
PMEK unlock	17
PMEK access key rotation	17
Transport encryption for PMEK access keys	17
Access key rotation flows.....	18
Algorithm support.....	18
Storage root key rotation.....	18
Reporting of erasure state	18
Externally-injected entropy	18
Cryptography helper commands	18
Post-Quantum Cryptography (PQC) requirements.....	19
Hardware.....	20
KMB-Encryption Engine Interface.....	21
Overview	21
KMB Command Sequence.....	26
Encryption Engine Requirements.....	26
Storage Root Key Fuse requirements.....	26
Storage Root Key Fuse programming	27
Error reporting and handling.....	27
Fatal errors	27
Non-fatal errors.....	27
Runtime Firmware environment	28
Boot and initialization	28
Fault handling.....	28
Mailbox commands.....	28

GET_STATUS.....	29
GET_CAPABILITIES.....	29
GET_ALGORITHMS	30
CLEAR_SECRETS	30
MIX_EXTERNAL_ENTROPY	31
GET_ENCAPSULATION_PUB_KEY.....	31
ENDORSE_ENCAPSULATION_PUB_KEY.....	31
ROTATE_ENCAPSULATION_KEY	32
GENERATE_PMEK.....	32
REWRAP_PMEK.....	33
UNLOCK_PMEK	33
INITIALIZE_MEK.....	34
MIX_PMEK.....	34
LOAD_MEK.....	35
UNLOAD_MEK.....	36
ENUMERATE_HANDLES	36
INVALIDATE_HANDLE.....	37
GET_RANDOM.....	37
RAW_DECRYPT_AES256_GCM.....	37
RAW_KDF_HMAC384.....	38
Terminology.....	39
References.....	40
Acknowledgments	41
Footnotes	42

Introduction

OCP L.O.C.K. (Layered Open-source Cryptographic Key management) is a feature set conditionally compiled into Caliptra, which provides key management for Data At Rest protection in self-encrypting storage devices.

OCP L.O.C.K. was originally created as part of the Open Compute Project ([OCP](#)). The major revisions of the OCP L.O.C.K. specifications are published as part of Caliptra at OCP as OCP L.O.C.K. is an optional extension to Caliptra. The evolving source code and documentation for Caliptra are in the repository within the [CHIPS Alliance Project](#), a Series of LF Projects, LLC.

Background

OCP L.O.C.K is being defined to improve drive security. The life of a storage device in a datacenter is that the device leaves the supplier, a customer writes user data to the device, and then the device is decommissioned. The problem is that customer data is not allowed to leave the data center. There needs to be a high confidence that and storage device leaving the datacenter is secure. The current default cloud service provider (CSP) policy to ensure this level of security is to destroy the drive. Other policies may exist that leverage drive capabilities (e.g., Sanitize), but are deemed insufficient by these CSPs¹. This produces significant e-waste and inhibits any re-use/recycling.

OCP L.O.C.K. is solving this security issue with data encryption by defining entropy used to create a media encryption key that is able to encrypt all data on the storage device. If that entropy is deleted, then the media encryption key is unable to be generated to decrypt the data on that storage device (i.e., no access to the plaintext behind the ciphertext).

OCP LOCK is removing the need to destroy the storage device and providing the security to address these issues by:

- Preventing leakage of media keys via firmware vulnerabilities or side channels;
- Binding media keys to a set of securely-provisioned access keys; and
- Enabling attestable “hard” purge via erasure of fuse secrets.

OCP L.O.C.K. goals

The goal of OCP L.O.C.K. is to define a Key Management Block (KMB) that:

- Isolates storage keys to a trusted hardware block
- Binds storage keys to a given set of externally-supplied access keys
- Provides replay-resistant transport security for these access keys
- Allows access key injection into KMB without trusting the host
- able to be used in conjunction with the storage device support Opal and Key per I/O

¹ Refer to the [Self-encrypting deception: weaknesses in the encryption of solid state drives](#) by Carlo Meijer and Bernard van Gastel

Threat model

The protected asset is the user data stored at rest on the drive. The adversary profile extends up to nation-states in terms of capabilities.

Adversary capabilities include:

- Interception of a storage device in the supply chain.
- Theft of a storage device from a data center.
- Destructively inspecting a stolen device.
- Running arbitrary firmware on a stolen device.
- Attempting to glitch execution of code running on general-purpose cores.
- Stealing debug core dumps or UART/serial logs from a device while it is operating in a data center, and later stealing the device.
- Gaining access to any class secrets, global secrets, or symmetric secrets shared between the device and an external entity.
- Executing code within a virtual machine on a multi-tenant host offered by the cloud service provider which manages an attached storage device.
- Accessing all device design documents, code, and RTL.

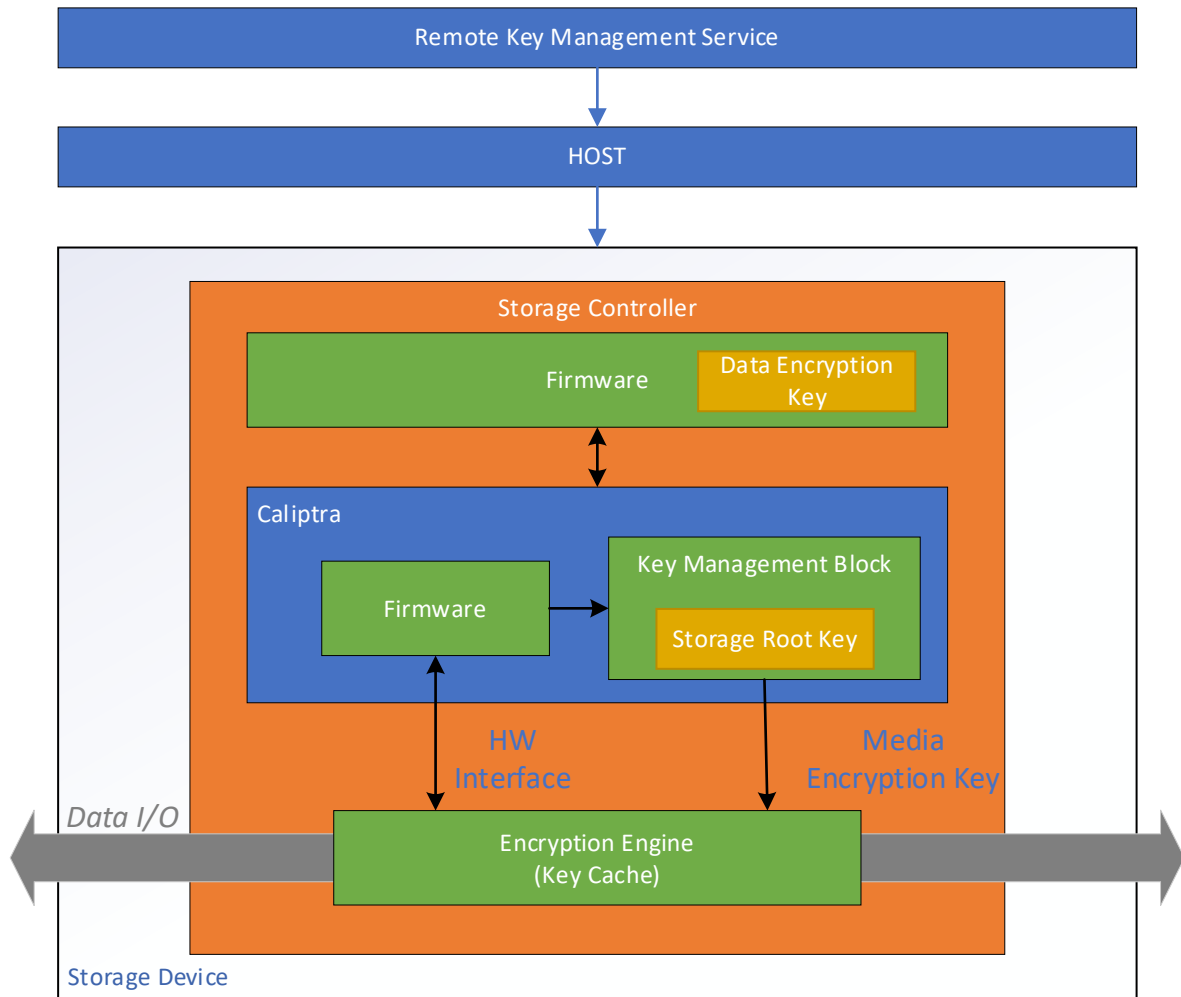
Given the above adversary profile, the following are a list of vulnerabilities that OCP L.O.C.K. is designed to mitigate.

- Keys managed by storage controller firmware are compromised due to implementation bugs or side channels.
- Keys erased by storage controller firmware are recoverable via invasive techniques.
- MEKs are not fully bound to user credentials due to implementation bugs.
- MEKs are bound to user credentials which are compromised by a vulnerable host.
- Cryptographic erasure was not performed properly due to a buggy host.

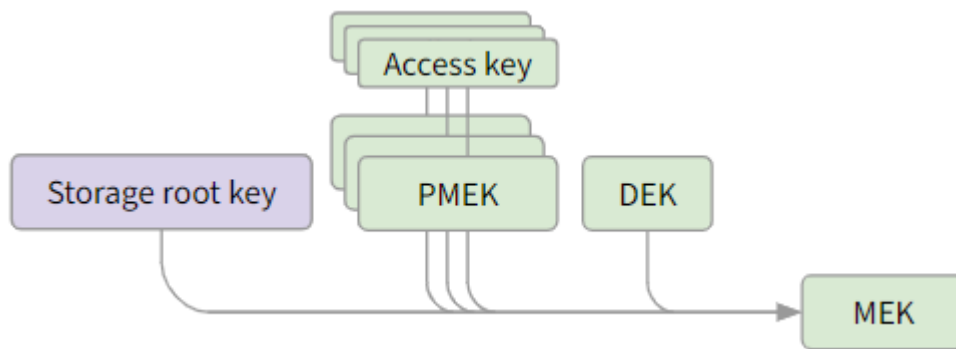
High level architecture

The following figure shows the basic high-level blocks of OCP L.O.C.K.

Figure 1 OCP L.O.C.K high level blocks



OCP L.O.C.K. is an optional feature to Caliptra use to generate Media Encryption Keys (MEKs) and load them into the Key Cache of the encryption engine used to encrypt and decrypt user data at rest. MEKs are never visible to any firmware. To load a MEK into the Key Cache of the Encryption Engine, Firmware interfaces to the Key Management Block to generate or retrieve a previously generated MEK that is created using the Storage Root Key, optional external Partial Media Encryption Keys (PMEK), and a Data Encryption Key (DEK) and then cause hardware to load the MEK into the Encryption Engine. Each MEK has associated metadata, to identify the namespace, LBA range, and how to load the MEK in the Encryption Engine.

Figure 2 MEK Generation

The Remote Key Management Service is utilized by the host to allow the host access to information for the PMEKS. OPAL or Key per I/O are example Remote Key Management Services. Each PMEK is encrypted as rest using an externally-injected access key where that access key is not stored persistently on the storage device. If there is more than one PMEK used to generate a MEK, then it requires multiple authorities to unlock a given range of user data.

Each MEK is bound for its lifetime to the Storage Root Key, the list of PMEKS, and the DEK. To generate a MEK, the access key for each PMEK must be provided. All MEKs are removed from the Encryption Engine on a power cycle or during sanitization on the storage device.

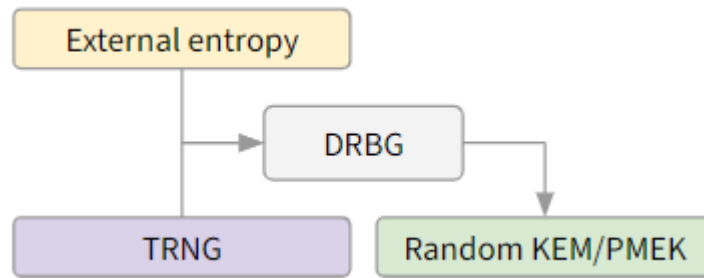
The DEK may be derived from or decrypted by a users C-pin to support legacy OPAL. The DEK may be the imported key associated with a Key per I/O key tag.

KMB generates Key Encapsulation Mechanism (KEM) keypairs and stores them in internal volatile memory using a KEM algorithm such as ECDH or MLKEM/Kyber. KMB can issue endorsements of KEM public keys allowing a Remote Key Management Services to ensure they only release access keys to authentic devices. PMEK access keys are encrypted in transit using these KEM keypairs. Upon drive reset, the access key must be re-encrypted to a new KEM for its associated PMEK to be usable.

KMB can maintain multiple active KEM keypairs. Nominally, one for each supported algorithm. KMB will automatically initialize a KEM for each supported algorithm and this may be done lazily. KMB will allow the controller to trigger KEM rotation. This can be done as part of zeroization.

KMB supports PMEK access key rotation where the storage controller must replace an old encrypted PMEK with a new encrypted PMEK. The end user must prove to KMB that they control both old and new access key. This is done by encrypting the new access key with the old access key. The new access key is therefore double-encrypted when provided to KMB with the old access key and with an ephemeral transport encryption key.

KMB is able to generate two kinds of keys: KEMs (used for access key transport encryption) and PMEKS (used for MEK derivation). A host is able to external inject entropy into KMB where it is held in internal volatile memory. Subsequent keys are randomly generated using both KMB's TRNG and the host's entropy as shown in figure x.

Figure 3 KEM and PMEK Key Generation

MEK derivation

When controller firmware wishes to program an MEK to the hardware cryptographic engine, it performs the following steps:

1. Instruct Caliptra to initialize its MEK seed buffer.
 - Caliptra initializes the buffer using a value derived from the storage root key as shown in Figure 4.
2. Provide zero or more PMEKs to Caliptra.
 - Caliptra extends the MEK seed buffer using each given PMEK as shown in Figure 5.
3. Provide a DEK to Caliptra.
 - Caliptra derives the MEK using the given DEK and the contents of the MEK seed buffer as shown in Figure 6.
4. Provide MEK metadata to Caliptra, such as the MEK's associated namespace and logical block address range.
 - Caliptra programs the derived MEK and its metadata to the hardware cryptographic engine as shown in Figure 6 and Figure 7.

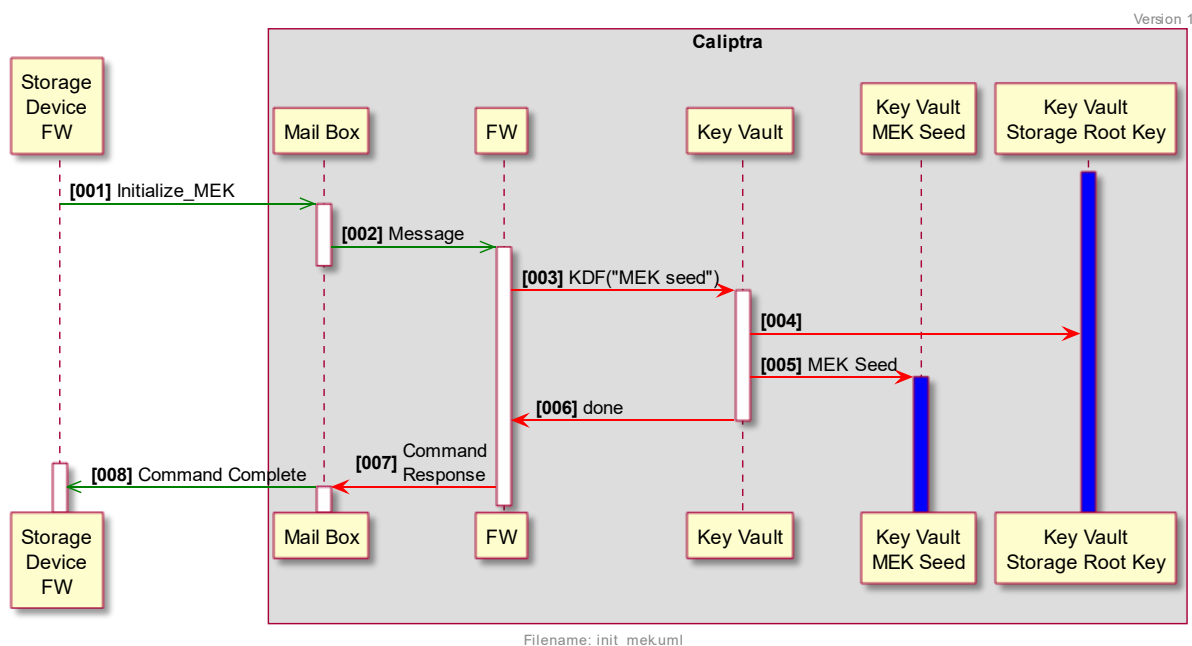
Figure 4 Sequence to Initialize the MEK

Figure 5 Sequence to mix PEM into MEK seed

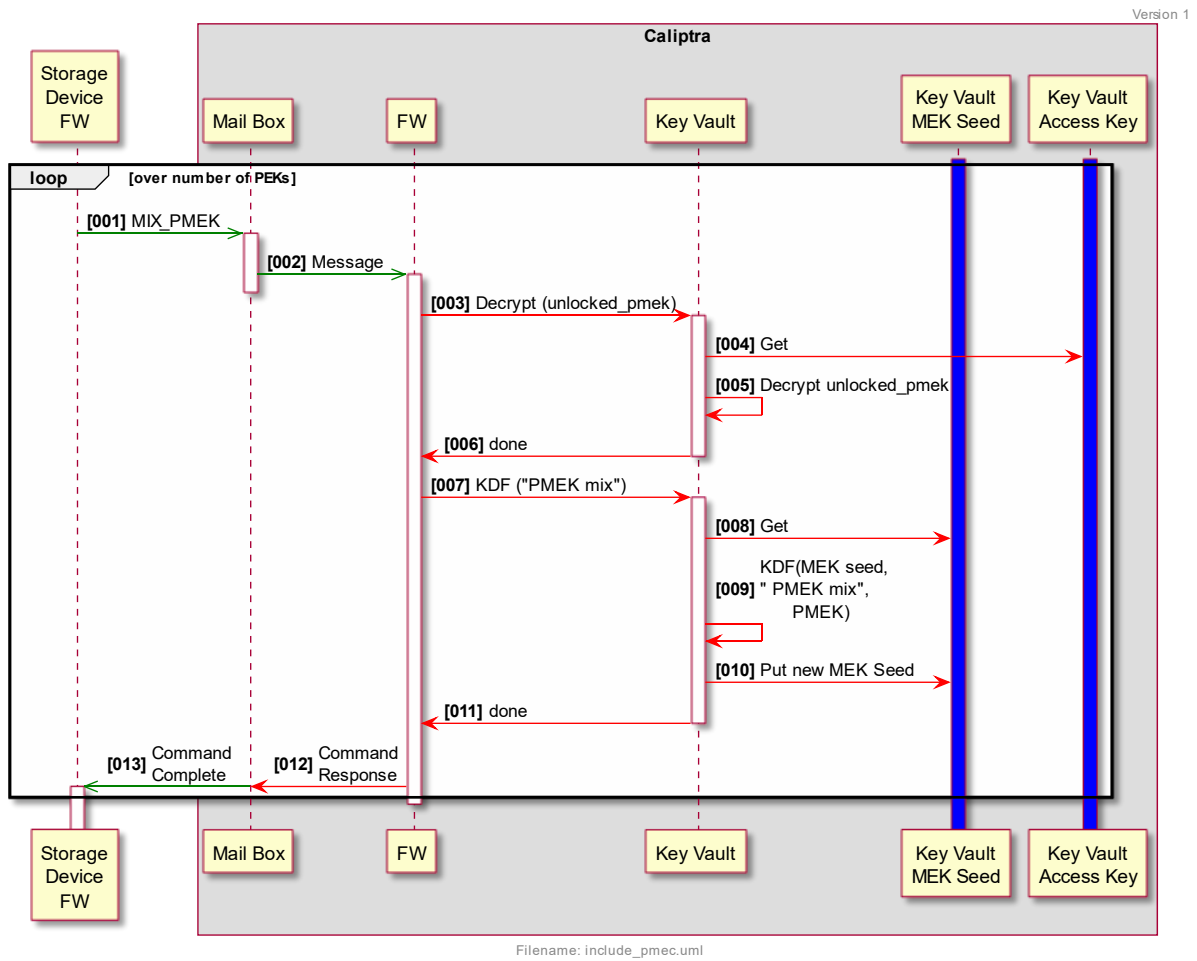


Figure 6 Sequence to load an MEK

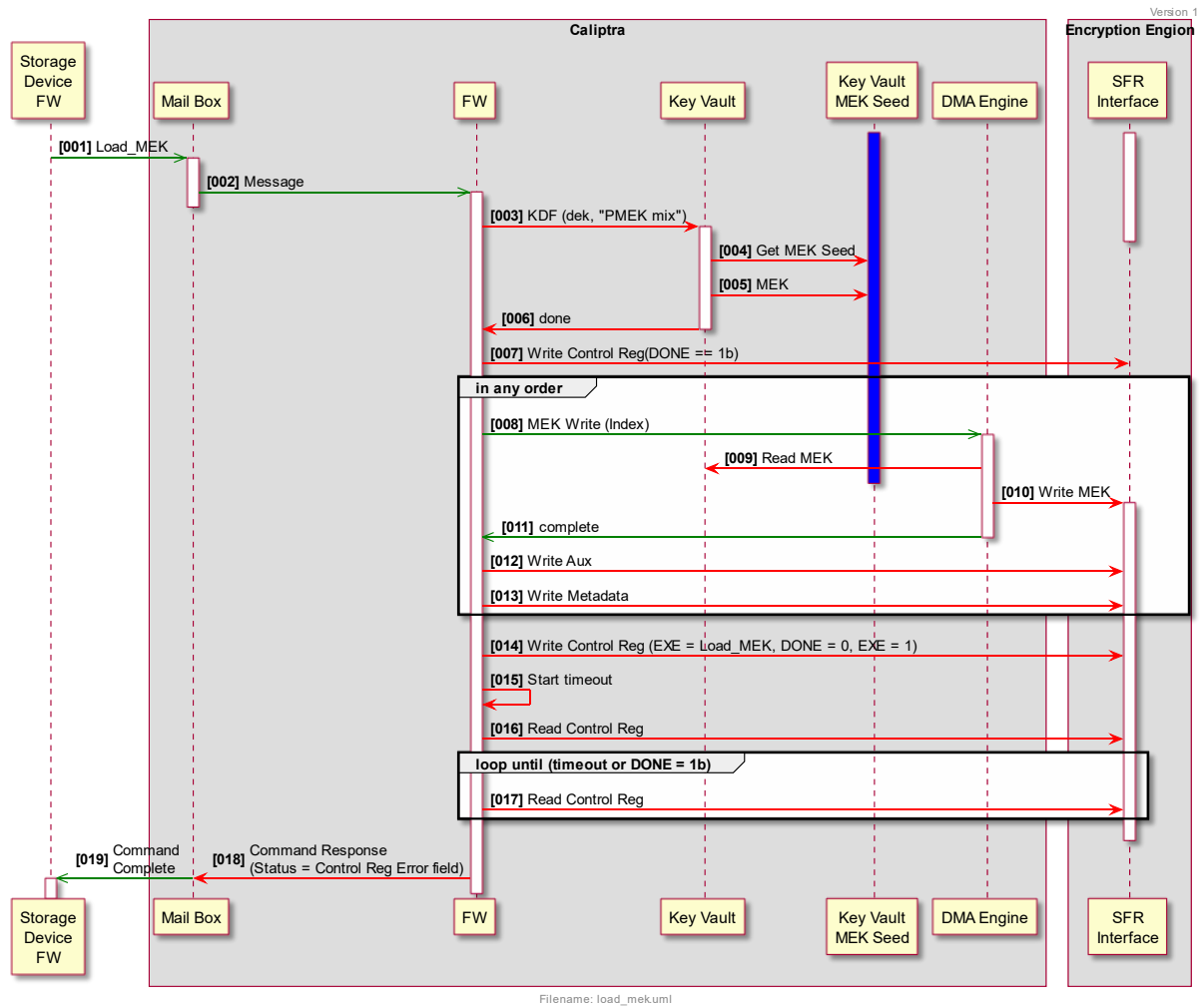
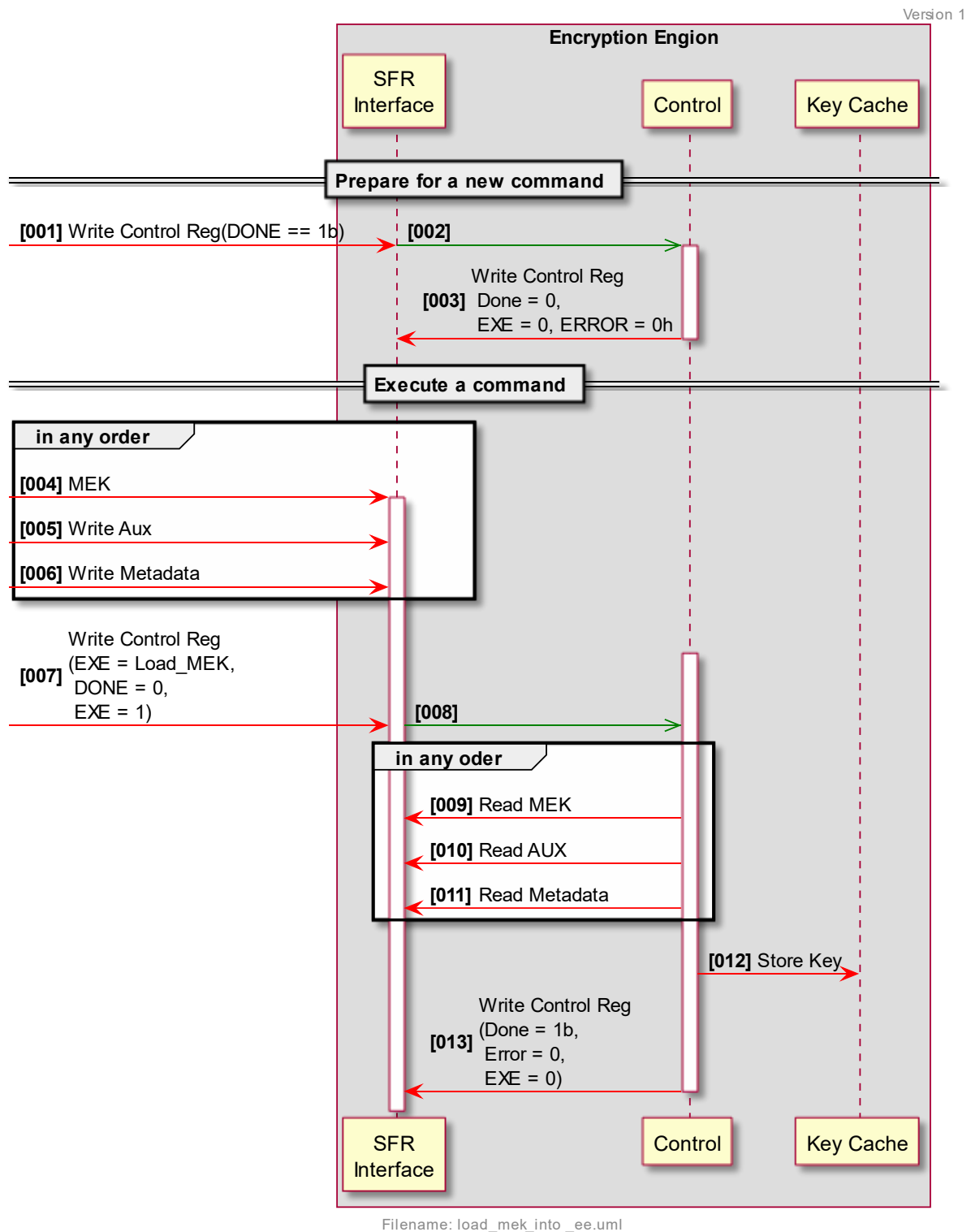


Figure 7 Sequence to load Encryption Engine with Key Cache from SFR interface.



PMEK lifecycle

To be filled out when more information is available.

PMEK Generation

Controller firmware may request that Caliptra generate a random PMEK, bound to a given access key. Caliptra performs the following steps:

1. Unwrap the given PMEK access key. See [below] for details on access key transport security.
2. Randomly generate a PMEK.
3. Derive a PMEK encryption key from the storage root key and the decrypted access key.
4. Encrypt the PMEK to the PMEK encryption key.
5. Return the encrypted PMEK to the controller firmware.

Controller firmware may then store the encrypted PMEK in persistent storage.

PMEK unlock

Encrypted PMEKS stored at rest in persistent storage are considered "locked", and must be unlocked before they can be used to derive MEKS. Unlocked PMEKS are also encrypted when handled by controller firmware. Unlocked PMEKS do not survive across device reset.

To unlock a PMEK, Caliptra performs the following steps:

1. Unwrap the given PMEK access key.
2. Derive the PMEK decryption key from the storage root key and the decrypted access key.
3. Decrypt the PMEK using the PMEK decryption key.
4. Encrypt the PMEK using an ephemeral export key that is randomly initialized on startup and lost on reset.
5. Return the re-encrypted "unlocked" PMEK to the controller firmware.

Controller firmware may then stash the encrypted unlocked PMEK in volatile storage, and later provide it to Caliptra when deriving an MEK, as described [above].

PMEK access key rotation

The access key to which a PMEK is bound may be rotated. The user must prove that they have knowledge of both the old and new access key before a rotation is allowed. Caliptra performs the following steps:

1. Unwrap the given old and new access keys.
2. Derive the old PMEK decryption key from the storage root key and the decrypted old access key.
3. Derive the new PMEK decryption key from the storage root key and the decrypted new access key.
4. Decrypt the PMEK using the old PMEK decryption key.
5. Encrypt the PMEK using the new PMEK decryption key.
6. Return the re-encrypted PMEK to the controller firmware.

Controller firmware then erases the old encrypted PMEK and stores the new encrypted PMEK in persistent storage.

Transport encryption for PMEK access keys

In L.O.C.K., Caliptra maintains a set of key-encapsulation-mechanism (KEM) keypairs, one per algorithm that L.O.C.K. supports. Each KEM public key is endorsed with a certificate that is generated by Caliptra and signed by Caliptra's DICE identity. KEM keypairs are randomly generated on Caliptra startup, may be periodically rotated, and are lost when the drive resets.

When a user wishes to unlock a PMEK (which is required prior to deriving any MEKS bound to that PMEK), the user performs the following steps:

1. Obtain the KEM public key and certificate from the storage device.

2. Validate the KEM certificate and attached DICE certificate chain.
3. Run `Encaps` and encrypt their access key to the resulting shared key.
4. Transmit the KEM ciphertext and encrypted access key to the storage device.

Upon receipt, Caliptra will perform the following steps:

1. Run `Decaps` and decrypt the user's access key with the resulting shared secret.
2. Derive the PMEK encryption key using the storage root key and the decrypted access key.
3. Perform PMEK generation, unlock, or rotation actions detailed [above].

Upon drive reset, the KEMs are regenerated, and any access keys for PMEKs that had been unlocked prior to the reset will need to be re-provisioned.

Access key rotation flows

As noted [above], during access key rotation the user must prove knowledge of both the old and new access keys. This is accomplished using a slight variation on the encaps-decaps flow. When a new access key is provided to Caliptra during a rotation, the new access key is double-encrypted: first to the old access key, and then to the shared secret obtained from the `Encaps` operation.

Caliptra then performs a double decryption when unwrapping the new access key, proving that the provisioner of the new access key also knows the old access key.

Algorithm support

OCP L.O.C.K. shall support the following KEM algorithms:

- P384 ECDH
- Hybridized ML-KEM with P384 ECDH

Storage root key rotation

As the storage root key is held in fuses, storage can only be rotated a limited number of times.

This section will be fleshed out with additional details as they become available.

Reporting of erasure state

A key use-case around cryptographic erasure of user data is confirmation that erasure occurred. This allows third parties to verify cleanliness state, and mitigates risk of bugs in host software that drives cryptographic erasure.

L.O.C.K. will enable reporting of whether a drive is clean following a storage root key rotation.

This section will be fleshed out with additional details as they become available.

Externally-injected entropy

Caliptra with L.O.C.K. relies on randomness to generate PMEKs and KEM keypairs. Storage devices feature their own source of randomness. For users that prefer not to solely trust device-level randomness sources, Caliptra allows the host to sample entropy from its own randomness source and inject it into Caliptra. Injected randomness is used to permute DRBG state that is held in volatile memory within Caliptra.

Cryptography helper commands

It is anticipated that controller firmware will perform its own vendor-specific cryptographic operations

outside of the Caliptra boundary. To allow those operations to leverage FIPS-validated cryptographic implementations within Caliptra, Caliptra will support helper mailbox commands that expose the following raw primitive cryptographic operations:

- Get random bytes, using a DRBG as defined in NIST [\[SP 800-90A\]](#)
- Perform AES-256-GCM encryption or decryption, as defined in NIST [\[SP 800-38D\]](#)
- Perform HMAC-384-KDF key derivation, as defined in NIST [\[SP 800-108\]](#)

Post-Quantum Cryptography (PQC) requirements

This section will be fleshed out with additional details as they become available.

Hardware

The following figure describes the OCP L.O.C.K..

Figure 8: KMB block diagram

This figure will be fleshed out with additional details as they become available.

- Memory requirements:

This section will be fleshed out with additional details as they become available.

- Cryptography requirements:

This section will be fleshed out with additional details as they become available.

KMB-Encryption Engine Interface

This section defines the interface between the Key Management Block (KMB) and an Encryption Engine. An Encryption Engine is used to encrypt/decrypt user data and its design and implementation are vendor specific. Media Encryption Keys (MEKs) are the keys which are generated within KMB and used by Encryption Engine to encrypt and decrypt user data. This interface is used to load MEKs from KMB to Encryption Engine or to cause the Encryption Engine to unload (i.e., remove) loaded MEKs. The MEKs transferred between the KMB and the Encryption Engine shall not be accessible by the Controller Firmware.

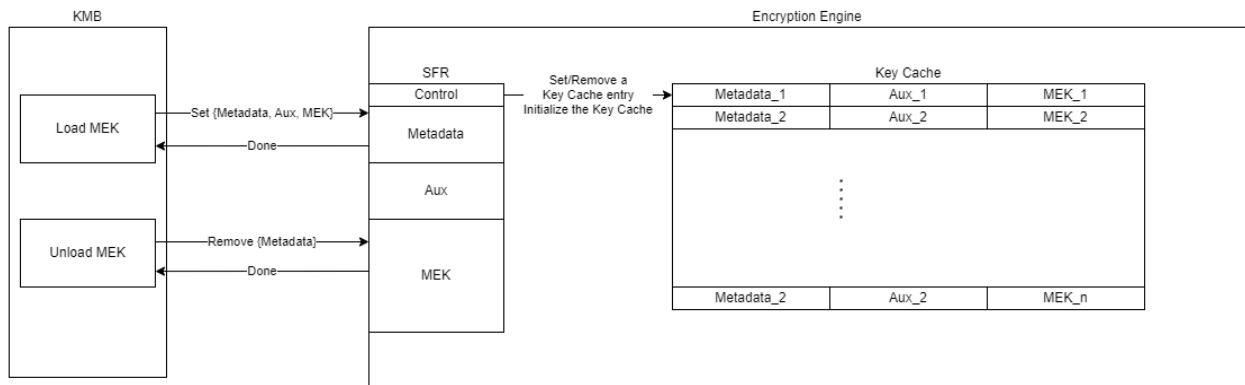
Overview

The Encryption Engine uses a stored MEKs to encryption and decryption use data. For the purposes of this specification, the entity within the Encryption Engine used to store the MEKs is called Key Cache. Each encryption and description of user data is coupled to a specific MEK which is stored in Key Cache bound to a unique identifier, called *metadata*. Each (*metadata*, MEK) pair is also associated with additional information, called *aux*, which is used neither as MEK nor an identifier, but has some additional information about the pair. Therefore, the Key Cache as an entity which stores (*metadata*, *aux*, MEK) tuples.

In order to achieve the security goals for KMB, KMB is limited to be the unique component which loads/unloads a (*metadata*, *aux*, MEK) tuple into the Key Cache and unloads a tuple within the device, so that MEKs can only be exposed to KMB and Encryption Engine. Controller Firmware arbitrates all operations in the KMB to Encryption Engine interface, therefore Controller FW is responsible for managing which MEK is loaded in Key Cache. Controller Firmware has full control on *metadata* and optional *aux*. Figure 8 is an illustration of the KMB to Encryption Engine interface that shows:

- the tuple for loading a MEK;
- the Metadata for unloading a MEK; and
- an example of a Key Cache within the Encryption Engine.

Figure 8: KMB to Encryption SFR Interface



KMB uses Special Function Registers (SFRs) to communicate with Encryption Engine which are described in the following sections.

Special Function Registers (SFRs)

KMB uses Special Function Registers (SFRs) to communicate with Encryption Engine as shown in Table 1 and each of the following subsections describe the registers.

Table 1. KMB to Encryption Engine SFRs

Register	Address	Byte Size	Description
Control	SFR_BASE + 0h	4h	Register to handle commands
Metadata	SFR_BASE + 10h	14h	Register to provide metadata

Table 1. KBM to Encryption Engine SFRs

Register	Address	Byte Size	Description
Auxiliary Data (AUX)	SFR_BASE + 30h	20h	Register to provide auxiliary values
Media Encryption Key (MEK)	SFR_BASE + 50h	40h	Register to provide MEK

SFR_BASE is an address that is configured on KMB. The integrator should make sure that KMB can access these SFRs through these addresses.

Control Register (Offset SFR_Base + 0h)

Figure 9 defines the Control register used to sequence the execution of a command and obtain the status of that command.

Figure 9: Offset SFR_Base + 0h: CTRL – Control

Bits	Type	Reset	Description												
31	RO	0h	Ready (RDY): After an NVM Subsystem Reset, this bit is set to 1b, then the Encryption Engine is ready to execute commands. If this bit is set to 0b, then the Encryption Engine is not ready to execute commands.												
30:20	RO	0h	Reserved												
19:16	RO	0h	Error (ERR): If the DONE bit is set to 1b by the Encryption Engine, then this field is set to a non-zero value to indicate the Encryption Engine detected an error during the execution of the command specified by the CMD field. The definition of a non-zero value is vendor specific. <table><tr><th>Value</th><th>Description</th></tr><tr><td>0h</td><td>Command Successful</td></tr><tr><td>1h to 3h</td><td>Reserved</td></tr><tr><td>4h to Fh</td><td>Vendor Specific</td></tr></table> If the DONE bit is set to 1b by the Encryption Engine and this field is set to 0h, then the Encryption Engine is indicating a successful execution of a command specified by the CMD field. If the DONE bit is set to 1b by KMB, then the field is set to 0000b.	Value	Description	0h	Command Successful	1h to 3h	Reserved	4h to Fh	Vendor Specific				
Value	Description														
0h	Command Successful														
1h to 3h	Reserved														
4h to Fh	Vendor Specific														
15:6	RO	0h	Reserved												
2:5	RW	0h	Command (CMD): This field specifies the command to execute or the command associated with the reported status. <table><tr><th>Value</th><th>Description</th></tr><tr><td>0h:</td><td>Reserved</td></tr><tr><td>1h</td><td>Load MEK: Load the key specified by the AUX field and MEK register into the Encryption Engine as specified by the METD field.</td></tr><tr><td>2h</td><td>Unload MEK: Unload the MEK from the Encryption Engine as specified by the METD field.</td></tr><tr><td>3h</td><td>Sanitize: Unload all of the MEKs from the Encryption Engine (i.e., Sanitize the Encryption Engine MEKs.).</td></tr><tr><td>4h to Fh</td><td>Reserved</td></tr></table>	Value	Description	0h:	Reserved	1h	Load MEK: Load the key specified by the AUX field and MEK register into the Encryption Engine as specified by the METD field.	2h	Unload MEK: Unload the MEK from the Encryption Engine as specified by the METD field.	3h	Sanitize: Unload all of the MEKs from the Encryption Engine (i.e., Sanitize the Encryption Engine MEKs.).	4h to Fh	Reserved
Value	Description														
0h:	Reserved														
1h	Load MEK: Load the key specified by the AUX field and MEK register into the Encryption Engine as specified by the METD field.														
2h	Unload MEK: Unload the MEK from the Encryption Engine as specified by the METD field.														
3h	Sanitize: Unload all of the MEKs from the Encryption Engine (i.e., Sanitize the Encryption Engine MEKs.).														
4h to Fh	Reserved														
1	RW	0b	Done (DN): This bit indicates the completion of a command by the Encryption Engine. If this bit is set to 1b by the Encryption Engine, then the Encryption Engine has completed the command specified by the CMD field. If the EXE bit is set to 1b and this bit is set to 1b, then the Encryption Engine has completed executing the command specified by the CMD field and the ERR field indicates the status of the execution of that command. A write of the value 1b to this bit shall cause the Encryption Engine to: • set this bit to 0b; • set the EXE bit to 0b; and • set the ERR field to 00b.												
0	RW	0b	Execute (EXE): A write of the value 1b to this bit specifies that the Encryption Engine is to execute the command specified by the CMD field. If the DONE bit is set to 1 by KMB, then the bit is set to 0b.												

From the KMB, the Controller register is the register to write a command and receive its execution

result. From its counterpart, the Encryption Engine, the Controller register is used to receive a command and write its execution result.

The expected change flow of the Controller register to handle a command is as follows:

1. If **RDY** is set to 1b, then KMB writes **CMD** and **EXE**
 - A. **CMD**: either 1h, 2h or 3h
 - B. **EXE**: 1b
2. The Encryption Engine writes **ERR** and **DN**
 - A. **ERR**: either 0b or a non-zero value depending on the execution result
 - B. **DN**: 1b
3. The KMB writes **DN**
 - A. **DN**: 1b
4. The Encryption Engine writes **CMD**, **ERR**, **DN** and **EXE**
 - A. **CMD**: 0h
 - B. **ERR**: 0h
 - C. **DN**: 0b
 - D. **EXE**: 0b

The KMB therefore interacts with the Control register as follows in the normal circumstance:

1. The KMB writes **CMD** and **EXE**
 - A. **CMD**: either 1h, 2h or 3h
 - B. **EXE**: 1b
2. The KMB waits **DN** to be 1
3. The KMB writes **DN**
 - A. **DN**: 1b
4. The KMB waits **DN** to be 0

Since the Controller register is in fact a part of the Encryption Engine whose implementation can be unique by each vendor, behaviors of the Control register with the unexpected flow are left for vendors. For example, a vendor who wants robustness might integrate a write-lock into the Control register in order to prevent two almost simultaneous writes on **EXE** bit.

Metadata Register (Offset SFR_Base + 0h)

Figure 10 defines the Metadata register.

Figure 10: Offset SFR_Base + 10h: METD – Metadata

Bytes	Type	Reset	Description
19:00	RW	0b	Metadata (METD) : This field specifies metadata that is vendor specific and specifies the entry in the Encryption Engine for the Encryption Key.

For the security goal of this project, the KMB and the Encryption Engine supposed to be the only components which have access to MEKs. Each MEK must then be bound to a unique identifier, which can be accessible by other components, in order for an appropriate key to be used for any key-related operations including data I/O. In a LOCK-enabled system, the **METD** field is expected to be used as such identifier.

Instead of generating a random and unique identifier within the KMB while generating an MEK, the KMB takes an **METD** value as input from the Controller Firmware and write to the Metadata register without any modification for the sake of the following reasons:

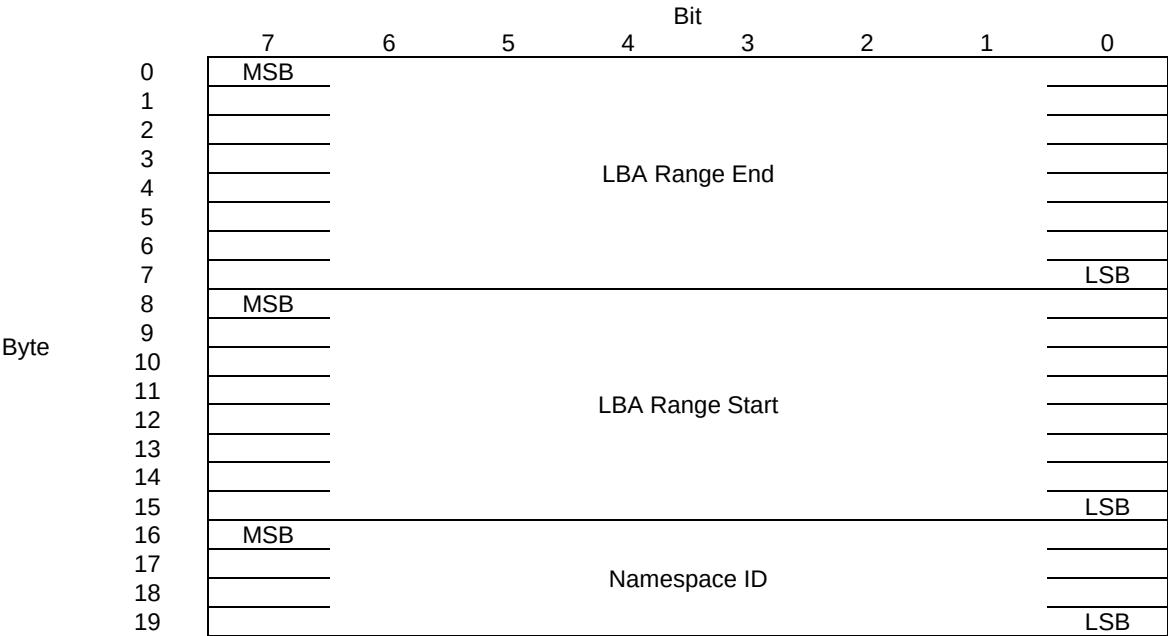
1. A vendor does not need to implement an additional algorithm to map between identifiers in its own system and in the KMB

2. A vendor-unique key-retrieval algorithm can easily be leveraged into a **METD**-generation algorithm

In order to reduce ambiguity, two examples of **METD** field will be given: Logical Block Addressing (LBA) range-based metadata; and key-tag based metadata.

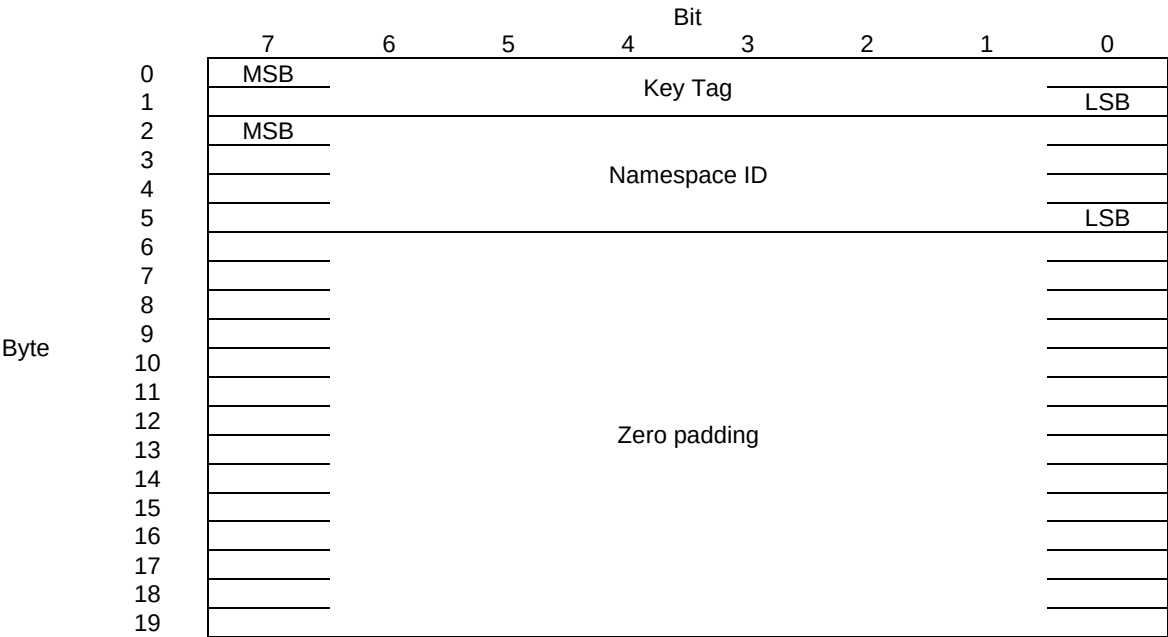
When an SSD stores data with address-based encryption, an MEK can be uniquely identified by a (LBA range, Namespace ID) pair. Then, the (LBA range, Namespace ID) pair can be leveraged into **METD** as on Figure 11.

Figure 11: LBA Range Based Metadata Format



Address-based encryption is not however the only encryption mechanism in SSDs. For example, in the TCG Key per I/O [...], an MEK is selected by a key tag, which is hard to be mapped with an address. Figure 12 shows an example of **METD** in such cases.

Figure 12: Key Tag Based Metadata Format



The above examples are not the only possible values of **METD**. Vendors are encouraged to design

and use their own **METD** if it fits better to their system.

Auxiliary Data Register (Offset SFR_Base + 0h)

Figure 13 defines the Auxiliary Data register.

Figure 13: Offset SFR_Base + 20h: AUX – Auxiliary Data

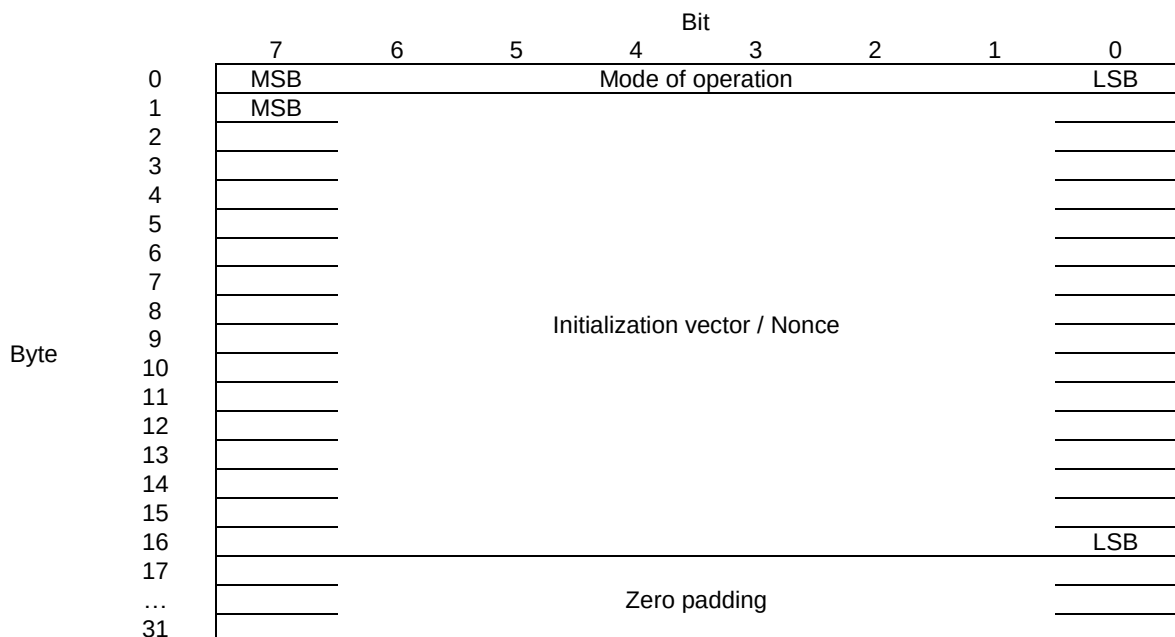
Bytes	Type	Reset	Description
19:00	RW	0h	Auxiliary Data (AUX): This field specifies auxiliary data associated to the MEK.

At first glance, the usage of **AUX** might not be straightforward. The intuition behind introducing the **AUX** field is to support vendor-specific features on MEKs. The KMB itself is only supporting fundamental functionalities in order to minimize attack surfaces on MEKs. Moreover, vendors are not restricted to design and implement their own MEK-related functionalities on the Encryption Engine unless they can be used to exfiltrate MEKs. In order to support these functionalities, some data may be associated and stored with an MEK and the **AUX** field is introduced to store such data with each MEK.

When the Controller Firmware requests the KMB to generate a new MEK, the Controller Firmware is expected to provide an **AUX** value. Similar to the **METD** field, the KMB will write the **AUX** value into the Auxiliary Data register without any modification.

One of simple use cases of **AUX** field is to store an offset of initialization vector or nonce. It can also be used in a more complicate use case. Here is an example. Suppose that there exists a vendor who wants to design a system which supports several mode of operations through the Encryption Engine while using the KMB. Then, a structure of **AUX** value as on Figure 14 can be used.

Figure 14: Auxiliary Data Format Example



When the Controller Firmware requests an MEK generation to the KMB, the Controller Firmware can use the **AUX** value to specify which mode of operation should be used and which value should be used as an initialization vector or a nonce with the generated MEK.

Media Encryption Key (MEK) Register (Offset SFR_Base + 50h)

Figure 15 defines the MEK register.

Figure 15: Offset SFR_Base + 40h: MEK – Media Encryption Key

Bytes	Type	Reset	Description
31:00	WO	0h	Secret Encryption Key (SEK): This field specifies a 256-bit Encryption Key

Figure 15: Offset SFR_Base + 40h: MEK – Media Encryption Key

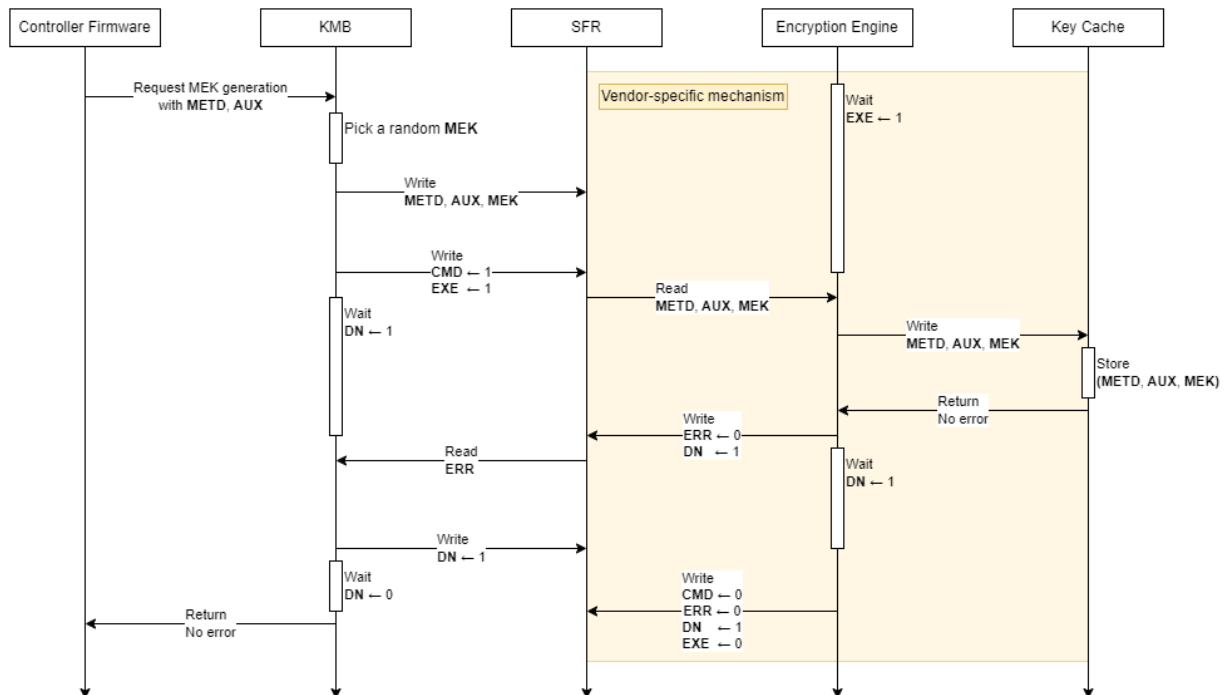
Bytes	Type	Reset	Description
63:32	WO	0h	Tweakable Key (TWK): This is the 256-bit AES-XTS tweakable key.

Since the AES-XTS is one of the most popular algorithms for data encryption, the MEK register is also designed with the key format of AES-XTS. The layout of the MEK register is however designed to help understanding the structure. The Encryption Engine is not restricted to only support the AES-XTS. The choice of encryption algorithm is solely dependent on vendors. When a vendor decides to use a different encryption algorithm, an MEK can be seen as a 64-byte random value rather than a (secret key, tweak key) pair and how to slice the 64-byte random value into an encryption key will be left to the vendor.

As a part of Caliptra, KMB protects MEK as secure as any secret key in Caliptra. MEK is stored in the Caliptra Key Vault, so that it can be protected against any firmware-level attacks. When KMB needs to write an MEK into the MEK register, it will be accomplished by using the DMA engine. Given an index and a destination identifier, the DMA engine copies the key value stored in the key vault of given index to the destination address to which the DMA engine translates the destination identifier.

KMB Command Sequence

Figure 16 shows a sample command execution. This is an expected sequence when the Controller Firmware requests the KMB to generate a new MEK. The behavior of Encryption Engine can be seen as one of possible mechanisms, and it can be different by vendors.

Figure 16: Command Execution Example

Encryption Engine Requirements

This section will be fleshed out with additional details as they become available.

Storage Root Key Fuse requirements

This section will be fleshed out with additional details as they become available.

Storage Root Key Fuse programming

This section will be fleshed out with additional details as they become available.

Error reporting and handling

This section describes OCP L.O.C.K error reporting and handling.

This section will be fleshed out with additional details as they become available.

Fatal errors

This section will be fleshed out with additional details as they become available.

Non-fatal errors

This section will be fleshed out with additional details as they become available.

Runtime Firmware environment

This section provides an extension to the Runtime Firmware environment defined for OCP Caliptra Runtime Firmware specification due to the support of OCP L.O.C.K.

Boot and initialization

The section defined additional boot and initialization needed to support OCP L.O.C.K.

The Runtime Firmware main function SHALL cause the generation of Storage Root Key and store this into volatile memory within KMB. The Storage Root key is generated from the Storage Root Key fuses.

Fault handling

A mailbox command can fail to complete in the following ways due to OCP L.O.C.K.:

- An ill-formed command
- Encryption Engine timeout
- Encryption Engine reported error

In all of these cases, the error is reported in the command returned status.

Depending on the type of fault, the SoC may to resubmit the mailbox command

For each Mailbox command that causes a command to execute on the Encryption Engine includes a timeout value is specified by the command. Caliptra aborts the command executing on the Encryption Engine if the Encryption Engine does not complete the command within the specified timeout and reports a LOCK_ENGINE_TIMEOUT result code.

Mailbox commands

Table 2 defines the additional Caliptra result codes due to supporting OCP L.O.C.K.

Table 2: OCP L.O.C.K. mailbox command result codes

Name	Value	Description
LOCK_ENGINE_TIMEOUT	0x4C45_544F ("LETO")	Timeout occurred when communicating with the drive crypto engine to execute a command
LOCK_ENGINE_CODE + u16	0x4443_xxxx ("ECxx")	Vendor-specific error code in the low 16 bits
LOCK_BAD_ALGORITHM	0x4C42_414C ("LBAL")	Unsupported algorithm, or algorithm does not match the given handle
LOCK_BAD_HANDLE	0x4C42_4841 ("LBHA")	Unknown handle
LOCK_NO_HANDLES	0x4C4E_4841 ("LNHA")	Too many extant handles exist
LOCK_KEM_DECAPSULATION	0x4C4B_4445 ("LKDE")	Error during KEM decapsulation

Name	Value	Description
LOCK_ACCESS_KEY_UNWRAP	0x4C41_4B55 ("LAKU")	Error during access key decryption
LOCK_PMEK_DECRYPT	0x4C50_4445 ("LPDE")	Error during PMEK decryption
LOCK_DECRYPT_FAILED	0x4C44_4546 ("LDEF")	Error during raw AES GCM decryption

The following sections define the additional Caliptra mailbox commands due to supporting OCP L.O.C.K.

GET_STATUS

Exposes a command that allows the SoC to determine if the Encryption Engine is ready to process commands as well vendor-defined drive crypto engine status data.

Command Code: 0x4753_5441 ("GSTA")

Table: **GET_STATUS** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
timeout	u32	Timeout in ms for communicating with the drive crypto engine

Table: **GET_STATUS** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
engine_ready	u16	Ready status of the storage device crypto engine. Supported values <we need to discuss>
engine_status	u8[16]	Vendor-defined drive crypto engine status data (NOT in the interface)
Reserved	u32[4]	Reserved fields

GET_CAPABILITIES

Exposes a command that allows the SoC to determine if OCP L.O.C.K. is supported on Caliptra and any specific OCP L.O.C.K. capabilities.

Command Code: 0x4743_4150 ("GCAP")

Table: **GET_STATUS** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: **GET_STATUS** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little

Name	Type	Description
		endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
capabilities	u8[16]	Ready status of the storage device crypto engine. - Byte 0 bit 0: OCP L.O.C.K. is supported is set to 1b - All other bytes/bits are reserved

GET_ALGORITHMS

Exposes a command that allows the SoC to determine the types of algorithms supported by KBM for endorsement, KEM, PMEK, and access key generation.

Command Code: 0x4743_4150 ("GCAP")

Table: **GET_ALGORITHMS** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: **GET_ALGORITHMS** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
endorsement_algorithms	u32	Identifies the supported endorsement algorithms Byte 0 bit 0: ecdsa_secp384r1_sha384²
kem_algorithms	u32	Identifies the supported KEM algorithms Byte 0 bit 0: ecdh_secp384r1_aes256_gcm³
pme_k_algorithms	u32	Indicates the size of PMEKs. Only one bit shall be reported. Byte 0 bit 0: 256 bits
access_key_algorithm	u32	Indicates the size of access keys. Only one bit shall be reported. Byte 0 bit 0: 256 bits
Reserved	u32[4]	Reserved

CLEAR_SECRETS

This command unloads all MEKs in the Encryption Engine and deletes all keys in KMB.

Command Code: 0x4353_4543 ("CSEC")

Table: **CLEAR_SECRETS** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
timeout	u32	Timeout in ms for communicating with the drive crypto engine

² As defined in [RFC 8446](#)

³

Table: **CLEAR_SECRETS** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

MIX_EXTERNAL_ENTROPY

This command mixes external entropy into a DRBG for subsequent KEM and PMEK generation.

Command Code: 0x4D45_454E (“MEEN”)

Table: **MIX_EXTERNAL_ENTROPY** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
timeout	u32	Timeout in ms for communicating with the drive crypto engine

Table: **MIX_EXTERNAL_ENTROPY** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

GET_ENCAPSULATION_PUB_KEY

This command mixes external entropy into a DRBG for subsequent KEM and PMEK generation.

Command Code: 0x4D45_454E (“MEEN”)

Table: **GET_ENCAPSULATION_PUB_KEY** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
kem_handle	u32	Handle for KEM keypair held in KMB memory

Table: **GET_ENCAPSULATION_PUB_KEY** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
pub_key	KemPubKey	KEM public key

ENDORSE_ENCAPSULATION_PUB_KEY

This command generates a signed certificate for the specified KEM using the specified endorsement algorithm.

Command Code: 0x4E45_505B (“EPEK”)

Table: **ENDORSE_ENCAPSULATION_PUB_KEY** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
kem_handle	u32	Handle for KEM keypair held in KMB memory
endorsement_algorithm	u32	Specifies the endorsement algorithm as defined by the endorsement_algorithms field in the GET_ALGORITHMS command. Only one bit shall be set.

Table: **ENDORSE_ENCAPSULATION_PUB_KEY** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
endorsement_len	u32	Length of endorsement field in bytes (N)
endorsement	u8[N]	DER-encoded X.509 certificate (includes nonce as extension)

ROTATE_ENCAPSULATION_KEY

This command rotates the KEM keypair indicated by the specified handle and stores the new KEM keypair in volatile memory iwithin KMB.

Command Code: 0x5245_4E4B ("RENK")

Table: **ROTATE_ENCAPSULATION_KEY** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
kem_handle	u32	Handle for old KEM keypair held in KMB memory

Table: **ROTATE_ENCAPSULATION_KEY** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
kem_handle	u32	Handle for new KEM keypair held in KMB memory

GENERATE_PMEK

This command unwraps the specified access key, generates a random PMEK, then uses the Storage Root Key and access key to encrypt the PMEK which is returned for the Storage Controller to persistently store.

Command Code: 0x5245_4E4B ("RENK")

Table: **GENERATE_PMEK** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key

Table: **GENERATE_PMEK** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
encrypted_pmek	EncryptedPmek	Encrypted PMEK

REWRAP_PMEK

This command Unwraps access_key_1 and enc_access_key_2. Then access_key_1 is used to decrypt enc_access_key_2. The specified PMEK is decrypted using KDF(Storage root key, "PMEK", access_key_1). A new PMEK is encrypted with the output of KDF(Storage root key, "PMEK", access_key_2). The new encrypted PMEK is returned.

The Storage Controller stores the returned new encrypted PMEK. The Storage Controller may attempt to do a decryption the new PMEK without an error before deleting old PMEK. Controller FW erases the old encrypted PMEK

Command Code: 0x5245_5750 ("REWP")

Table: **REWRAP_PMEK** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
wrapped_access_key_1	WrappedAccessKey	KEM-wrapped access key 1
wrapped_enc_access_key_2	DoubleWrappedAccessKey	KEM-wrapped access key KEM-wrapped (access_key_2 encrypted to access_key_1)
old_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_1

Table: **REWRAP_PMEK** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
New_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_2

UNLOCK_PMEK

This command Unwraps access_key_1 and enc_access_key_2. Then access_key_1 is used to decrypt enc_access_key_2. The specified PMEK is decrypted using KDF(Storage root key, "PMEK",

access_key_1). A new PMEK is encrypted with the output of KDF(Storage root key, "PMEK", access_key_2). The new encrypted PMEK is returned.

The Storage Controller stores the returned new encrypted PMEK. The Storage Controller may attempt to do a decryption the new PMEK without an error before deleting old PMEK. Controller FW erases the old encrypted PMEK

Command Code: 0x5245_5750 ("REWP")

Table: **UNLOCK_PMEK** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
wrapped_access_key_1	WrappedAccessKey	KEM-wrapped access key 1
wrapped_enc_access_key_2	DoubleWrappedAccessKey	KEM-wrapped access key KEM-wrapped (access_key_2 encrypted to access_key_1)
old_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_1

Table: **UNLOCK_PMEK** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
new_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_2

INITIALIZE_MEK

This command Initializes the MEK seed stored in KMB by resetting any previously-existing content and performing a KDF operation on the Strage Root Key and the string "MEK seed".

Command Code: 0x494D_454B ("IMEK")

Table: **INITIALIZE_MEK** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: **INITIALIZE_MEK** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

MIX_PMEK

This command decrypts the specified PMEK with the with the ephemeral export secret and then updates the MEK seed in KMB by performing a KDF with the Meek seed, the decrypted PMEK, and the string "PMIX mix".

When generating a MEK, it is required that the MEK seed be initialized and then one or more MIX_PMEK are subsequently processed to modify the MEK seed. If this sequence is not met when processing this command, then the command shall fail with a xxx status.

Command Code: 0x4D50_4D4B ("MPMK")

Table: MIX_PMEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
unlock_pmek	EncryptedPmek	Unlocked encrypted PMEK

Table: MIX_PMEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

LOAD_MEK

This command causes the specified controller data encryption key to be combined with the MEK seed. The final MEK seed, specified metadata, and aux_metadata are loaded into the Encryption Engine Key Cache. The metadata is specific to the storage controller and specifies the information to the Encryption Engine on where within the Key Cache, the MEK is loaded.

The storage controller specified data encryption key may be a C-PIN-derived secret for Opal or a per-MEK value in KPIO.

The final MEK is generated by performing a KDF on the existing MEK seed in the KMB, the dek, and the string "MEK".

When generating a MEK, it is required that the MEK seed be initialized and then one or more MIX_PMEK are subsequently processed to modify the MEK seed prior to processing this command. If this sequence is not met when processing this command, then the command shall fail with a xxx status.

Command Code: 0x4C4D_454B ("LMEK")

Table: LOAD_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
dek	u8[32]	Storage controller-supplied "data encryption key"
metadata	u32	Metadata for MEK to load into the Encryption Engine of the Storage Controller. For example, the metadata may identify the NSID and an LBA range.
aux_metadata	u32	Auxiliary metadata for the MEK (optional; e.g. operation mode)
timeout	u32	Timeout in ms for communicating with the drive crypto engine

Table: **LOAD_MEK** output arguments

Name	Type	Description
checksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

UNLOAD_MEK

This command causes the MEK associated to the specified metadata to be unloaded for the Key Cache of the Encryption Engine. The metadata is specific to the storage controller and specifies the information to the Encryption Engine on where within the Key Cache, the MEK is loaded.

Command Code: 0x554D_454B ("UMEK")

Table: **UNLOAD_MEK** input arguments

Name	Type	Description
checksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
metadata	u8[4]	Metadata to identify the location of the MEK to be unload in the Key Cache of the Encryption Engine.
timeout	u32	Timeout in ms for communicating with the Encryption Engine

Table: **UNLOAD_MEK** output arguments

Name	Type	Description
checksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

ENUMERATE_HANDLES

This command returns a list of all currently-active handles for resources held by KMB.

Command Code: 0x4548_444C ("EHDL")

Table: **ENUMERATE_HANDLES** input arguments

Name	Type	Description
checksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: **ENUMERATE_HANDLES** output arguments

Name	Type	Description
checksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
handle_count	u32	Number of handles (N)
handles	HandleWithType[N]	List of (handle value, handle type) tuples

INVALIDATE_HANDLE

This command invalidates the resource associated with the given handle, freeing its resources.

Command Code: 0x4948_444C ("IHDL")

Table: INVALIDATE_HANDLE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
Handle	u32	Handle value

Table: INVALIDATE_HANDLE output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

GET_RANDOM

This command returns a random number in the specified number of bytes.

Command Code: 0x4752_414E ("GRAN")

Table: GET_RANDOM input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
len	u32	Number of bytes to be returned with a random number

Table: GET_RANDOM output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
len	u32	Number of bytes returned with a random value (N)
rand_bytes	u32	Bytes returned with a random value

RAW_DECRYPT_AES256_GCM

This command decrypts the specified ciphertext and returns the plaintext.

Command Code: 0x5244_4147 ("RDAG")

Table: RAW_DECRYPT_AES256_GCM input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Name	Type	Description
len	u32	Number of bytes to be returned with a random number
key	u8[32]	AES 256 key
iv	u8[12]	IV
tag	u8[16]	Integrity tag
ciphertext_len	u32	Number of bytes of ciphertext to decrypt (N)
ciphertext	u8[N]	Encrypted data

Table: **RAW_DECRYPT_AES256_GCM** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
plaintext_len	u32	Number of bytes returned in the plaintext (M)
plaintext	u8[M]	Bytes of the decrypted ciphertext

RAW_KDF_HMAC384

This command returns a derived key from the specified key, label, and context.

Command Code: 0x524B_484D ("RKHM")

Table: **RAW_KDF_HMAC384** input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
key	u8[48]	Input key
label_len	u32	Number of bytes in the label (N)
context_len	u32	Number of bytes in the context (N)
label	u8[N]	Label
context	u8(M)	Context

Table: **RAW_KDF_HMAC384** output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
derived_key	u8[48]	Derived key

Terminology

The following acronyms and abbreviations are used throughout this document.

Abbreviation	Description
AES	Advanced Encryption Standard
CSP	Cloud Service Provider
DEK	Data Encryption Key
DICE	Device Identifier Composition Engine
DRBG	Deterministic Random Bit Generator
ECDH	Elliptic-curve Diffie–Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
HMAC	Hash-Based Message Authentication Codes
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
KMB	Key Management Block
L.O.C.K.	Layered Open-Source Cryptographic Key-management
MEK	Media Encryption Key
ML-KEM	Module-Lattice-Based Key-Encapsulation Mechanism
NIST	National Institute of Standards and Technology
OCP	Open Compute Project
PMEK	Partial Media Encryption Key
RTL	Register Transfer Level
SED	Self-encrypting drive
SSD	Solid-state drive
UART	Universal asynchronous receiver-transmitter
XTS	XEX-based tweaked-codebook mode with ciphertext stealing

References

1. Need to add

Acknowledgments

The Caliptra Workgroup acknowledges the following individuals for their contributions to this specification.

CONTRIBUTORS

- Andrés Lagar-Cavilla (Google)
- Amber Huffman (Google)
- Charles Kuzman (Google)
- Jeff Anderson (Google)
- Chris Sabol (Google)
- Lee Prewitt (Microsoft)
- Michael Norris (Microsoft)
- Eric Eilertson (Microsoft)
- Bryan Kelly (Microsoft)
- Anjana Parthasarathy (Microsoft)
- Ben Keen (Microsoft)
- Jisoo Kim (Samsung)
- Gwangbae Chio (Samsung)
- Eric Hibbard (Samsung)
- Mike Allison (Samsung)
- Scott Shadley (Solidigm)
- Gamil Cain (Solidigm)
- Festus Hategekimana (Solidigm)
- John Geldman (Kioxia)
- Fred Knight (Kioxia)
- Paul Suhler (Kioxia)
- James Borden (Kioxia)

Footnotes
