OPEN
Compute Project

NVMe Project

OCP L.O.C.K.

0.8.1

Modular Base Specification
Effective April 22, 2025

Author: (See Acknowledgements section)

Current Template Version:

3 Layer (Base, Design and Product) Specification Template V1.6.0

Table of Contents

List of Figures

List of Tables

1 License

1.1 Open Web Foundation (OWF) CLA

Contributions to this Specification are made under the terms and conditions set forth in **Modified Open Web Foundation Agreement 0.9 (OWFa 0.9)**. (As of October 16, 2024) ("Contribution License") by:

Google
Microsoft
Samsung
Kioxia
Solidigm

Usage of this Specification is governed by the terms and conditions set forth in **Modified OWFa 0.9 Final Specification Agreement (FSA)** (As of October 16, 2024) **("Specification License")**.

You can review the applicable Specification License(s) referenced above by the contributors to this Specification on the OCP website at https://www.opencompute.org/contributions/templates-agreements.

For actual executed copies of either agreement, please contact OCP directly.

Notes:

> 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.

2 Acknowledgements

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

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

3 Compliance with OCP Tenets

3.1 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.

3.2 Efficiency

OCP L.O.C.K. is used to 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.

3.3 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.

3.4 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.

3.5 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.

4 Base specification

4.1 Introduction

OCP L.O.C.K. (Layered Open-source Cryptographic Key management) is a feature set conditionally compiled into Caliptra Subsystem 2.1+, which provides secure 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 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.

OCP L.O.C.K. may be integrated within a variety of self-encrypting storage devices, and is not restricted exclusively to NVMe.

4.2 Background

In the life of a storage device in a datacenter, the device leaves the supplier, a customer writes user data to the device, and then the device is decommissioned. Customer data is not allowed to leave the data center. The cloud service provider (CSP) needs high confidence that the storage device leaving the datacenter is secure. The current default CSP policy to ensure this level of security is to destroy the drive. Other policies may exist that leverage drive capabilities (e.g., Purge), but are not generally deemed inherently trustworthy by these CSPs [1]. This produces significant e-waste and inhibits any re-use/recycling.

Self-encrypting drives (SEDs) store data encrypted at rest to media encryption keys (MEKs). SEDs include the following building blocks:

- The storage media that holds data at rest.
- An encryption engine which performs line-rate encryption and decryption of data as it enters and exits the drive.
- A controller which exposes host-side APIs for managing the lifecycle of MEKs.

MEKs may be bound to user credentials, which the host must provide to the drive in order for the associated data to be readable. A given MEK may be bound to one or more credentials. This model is captured in the TCG Opal [2] specification.

MEKs may be securely purged, to effectively purge all data which was encrypted to the MEK. To purge an MEK, it is sufficient for the controller to purge all copies of it, or a key with which it was protected.

4.3 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.

  - This includes attacks where vendor firmware signing keys have been compromised.

- 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.

- MEKs managed by storage controller firmware are compromised due to implementation bugs or side channels.
- MEKs purged 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.

4.4 OCP L.O.C.K. goals

OCP L.O.C.K. is being defined to improve drive security. In an SED that takes Caliptra with OCP L.O.C.K. features enabled, Caliptra will act as a Key Management Block (KMB). The KMB will be the only entity that can read MEKs and program them into the SED's cryptographic engine. The KMB will expose services to controller firmware which will allow the controller to transparently manage each MEK's lifecycle, without being able to access the raw MEK itself.

The OCP L.O.C.K. KMB satisfies the following properties:

- Prevents leakage of media keys via firmware vulnerabilities or side channels, by isolating MEKs to a trusted hardware block.
- Binds MEKs to a given set of externally-supplied access keys, provisioned with replay-resistant transport security such that they can be injected without trusting the host.
- Uses epoch keys for attestable epoch-based cryptographic purge.
- Is able to be used in conjunction with the Opal or Key Per I/O [3] storage device specifications.

4.4.1 Non-goals

The following areas are out of scope for this project.

- Compliance with IEEE 1619 2025 requirements around key scope, i.e. restricting a given MEK to only encrypt a maximum of 244 128-bit blocks. Controller firmware will be responsible for enforcing this.
- Compliance with Common Criteria requirement FCS_CKM.1.1(c) [4] when supporting derived MEKs. Key Per I/O calls for DEKs to be injected into the device. To support OCP L.O.C.K.'s goals around enabling cryptographic purge, before the injected DEK is used to encrypt user data, it is first conditioned with an on-device secret that can be securely zeroized. FCS_CKM.1.1(c) currently does not allow injected keys to be thus conditioned

and therefore does not allow for cryptographic purge under the Key Per I/O model. A storage device that integates OCP L.O.C.K. and aims to be compliant with this Common Criteria requirement may not support Key Per I/O.

- Authorization for EPK/DPK/MPK rotation, or binding a given MEK to a particular locking range. The controller firmware is responsible for these.

4.4.2 Integration verification

A product which integrates OCP L.O.C.K. will be expected to undergo an OCP S.A.F.E. review, to ensure that the controller firmware correctly invokes OCP L.O.C.K. services.

4.5 Architecture

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

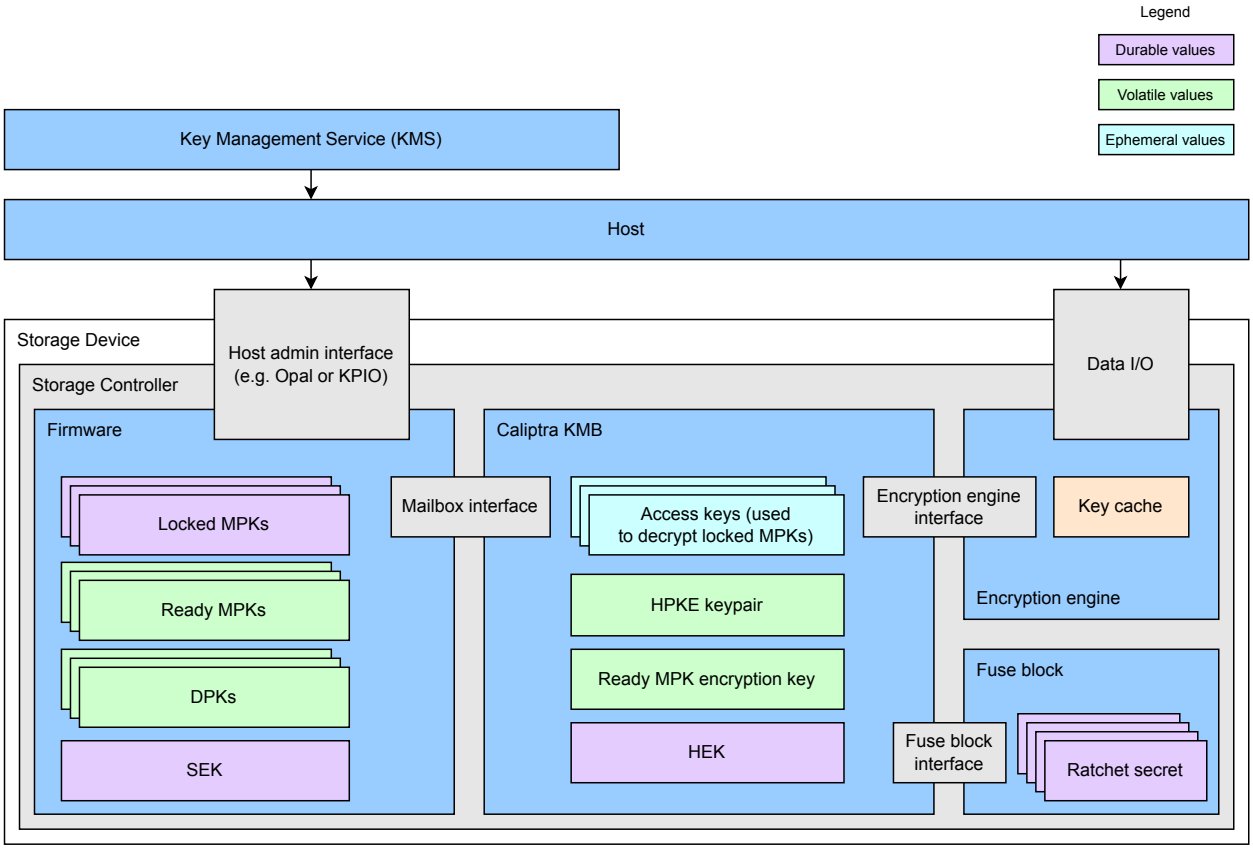

Figure 1: OCP L.O.C.K. high level blocks

To safeguard user data stored on the drive, KMB defines a set of “protection keys”, each of which must be available in order for an MEK to be accessible.

- The **data protection key (DPK)**, which is managed by the nominal owner of the data. A given MEK is bound to a single DPK.
  - In Opal the DPK may be protected by the user’s C_PIN, while in Key Per I/O the DPK may be the DEK associated with a given key tag.
- A configurable number of **Multi-party Protection Keys (MPKs)**, which are each managed by an additional entity that must assent before user data is available. A given MEK may be bound to zero or more MPKs.
  - Each multi-party entity grants access to the data by providing an access key to the drive. Each MPK is protected by a distinct access key, which is never stored persistently within the drive. MPK access keys are protected in transit using HPKE [5]. This enables use-cases where the access key is served to the drive from a remote key management service, without revealing the access key to the drive’s host.

  - Binding an MEK to zero MPKs allows for legacy Opal or Key Per I/O support.

- A composite **epoch protection key (EPK)**, which is a concatenation of two “component epoch keys” held within the device: the **Soft Epoch Key (SEK)** and the **Hard Epoch Key (HEK)**. The EPK is managed by the storage device itself, and all MEKs in use by the device are bound to it.
  - All MEKs in use by the drive can be purged by zeroizing either the SEK or HEK. New MEKs shall not be loadable until the zeroized epoch keys are regenerated.
  - KMB reports the zeroization state of the SEK and HEK, and therefore whether the drive is in a purged state.
  - The SEK is managed by controller firmware and shall be held in rewritable storage, e.g. in flash memory.
  - The HEK is managed by KMB and shall be held in fuses. This provides assurance that an advanced adversary is unable to recover key material that had been in use by the drive prior to the HEK zeroization.

The EPK, DPK, and set of configured MPKs are used together to derive an MEK secret, which protects a given MEK. The MEK protection is implemented as one of two methods:

- **MEK encryption** - the controller obtains a random MEK from KMB, encrypted by the MEK secret, and is allowed to load that encrypted MEK into the encryption engine via KMB. This supports Opal use-cases.
- **MEK derivation** - the controller instructs KMB to derive an MEK from the MEK secret and load the MEK into the encryption engine. This may support either Opal or Key Per I/O use-cases.

MEKs are never visible to controller firmware. Controller firmware instructs KMB to load MEKs into the key cache of the encryption engine, using standard interfaces described in Section 4.6. Each MEK has associated vendor-defined metadata, e.g. to identify the namespace and LBA range to be encrypted by the MEK.

KMB shall not cache MEKs in memory. The encryption engine shall remove all MEKs from the encryption engine on a power cycle or during zeroization of the storage device.

All keys randomly generated by KMB are generated using a DRBG seeded by Caliptra’s TRNG. The DRBG may be updated using entropy provided by the host.

4.5.1 Host APIs

The DPK can be modeled using existing TCG Opal or Key Per I/O host APIs.

Rotation of the HEK and SEK, management of MPKs and MPK access keys, and injection of host entropy, require additional host APIs beyond those available in TCG Opal or Key Per I/O. Such APIs are beyond the scope of the present document.

4.5.2 Key hierarchy

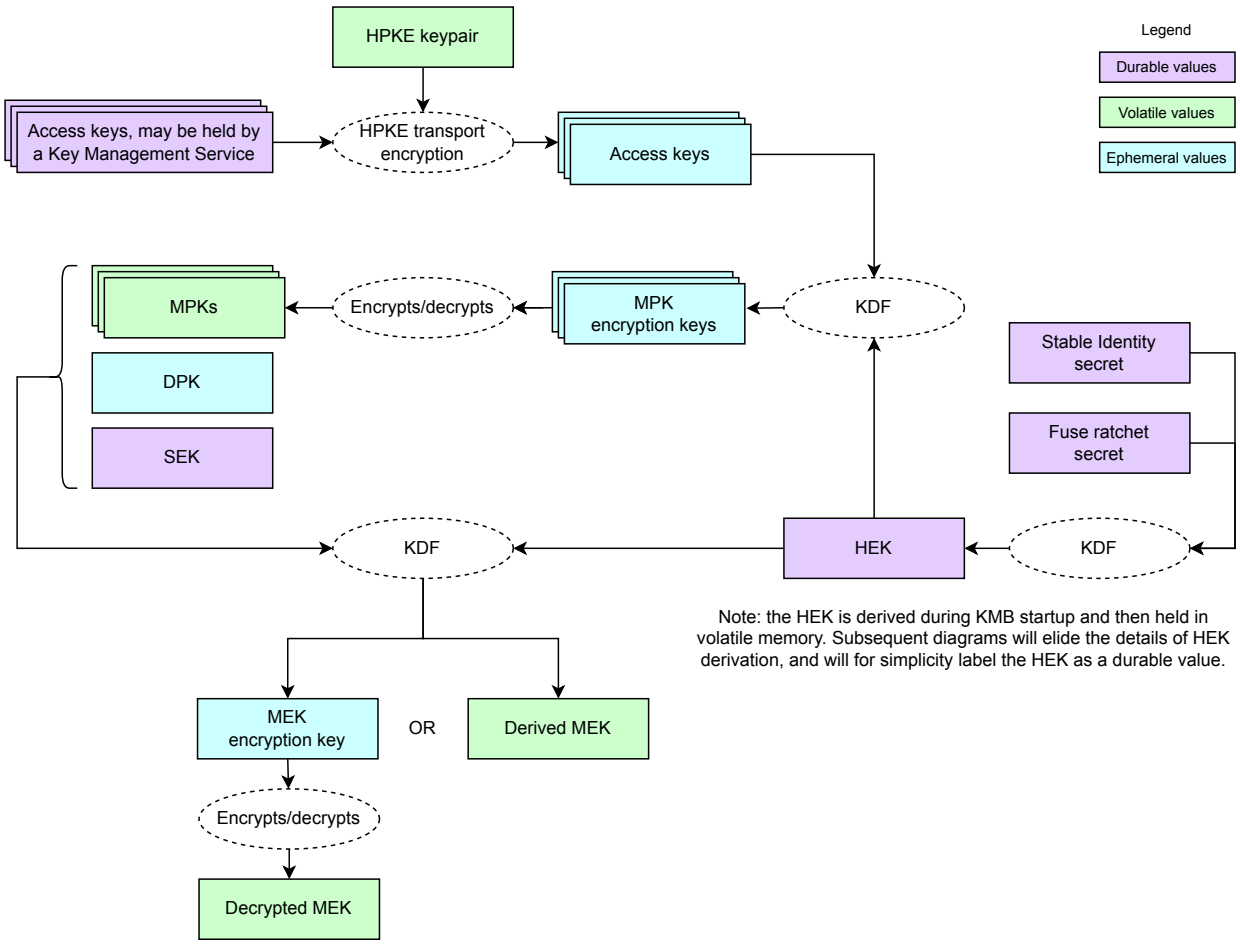

Figure 2: OCP L.O.C.K. key hierarchy

4.5.3 MPKs

MPKs are the mechanism by which KMB enforces multi-party authorization as a precondition to loading an MEK. MPKs exist in one of two states: locked or ready. In both these states the MPK is encrypted to a key known only to KMB.

- A locked MPK's encryption key is derived from the HEK as well as an externally-supplied access key. The locked MPK is held at rest by controller firmware.
- A ready MPK's encryption key (the Ready MPK Encryption Key) is a volatile key held within KMB which is lazily generated. Ready MPKs are held in controller firmware memory. Lazy generation allows injected host entropy to contribute to the key's generation.

The externally-supplied access key is encrypted in transit using an HPKE public key held by KMB. The "ready" state allows the HPKE keypair to be rotated after the access key has been provisioned to the storage device, without removing the ability for KMB to decrypt the MPK when later loading an MEK bound to that MPK.

For each MPK to which a given MEK is bound, the host is expected to invoke a command to supply the MPK's encrypted access key. Upon receipt the controller firmware passes that encrypted access key to KMB, along with the locked MPK, to produce the ready MPK which is cached in controller memory. This is done prior to the controller firmware actually loading the MEK, and is performed once for each MPK to which a given MEK is bound.

4.5.3.1 Transport encryption for MPK access keys

KMB maintains a set of HPKE keypairs, one per HPKE algorithm that KMB supports. Each HPKE public key is endorsed with a certificate that is generated by KMB and signed by Caliptra's DICE identity. HPKE keypairs are randomly generated on KMB startup, and mapped to unique handles. Keypairs may be periodically rotated and are lost when the drive resets. Controller firmware is responsible for enumerating the available HPKE public keys and exposing them to the user via a host interface.

When a user wishes to generate or ready a MPK (which is required prior to loading any MEKs bound to that MPK), the user performs the following steps:

1. Select the HPKE public key and obtain its certificate from the storage device.
2. Validate the HPKE certificate and attached DICE certificate chain.
3. Encrypt their access key to the HPKE public key.
4. Transmit the encrypted access key to the storage device.

Upon receipt, KMB will unwrap the access key and proceed to generate or ready the MPK.

Upon drive reset, the HPKE keypairs are regenerated, and any access keys for MPKs that had been made ready prior to the reset will need to be re-provisioned in order to transition those MPKs to a ready state again.

4.5.3.1.1 Algorithm support

KMB supports the following HPKE algorithms:

- P-384 ECDH
- Hybridized ML-KEM-1024 with P-384 ECDH

The following two diagrams illustrate the detailed cryptographic operations involved in unwrapping the access key using each of these algorithms. Subsequent diagrams elide these details and represent the operation as "HPKE unwrap".

4.5.3.1.2 MPK access key unwrap (ECDH)

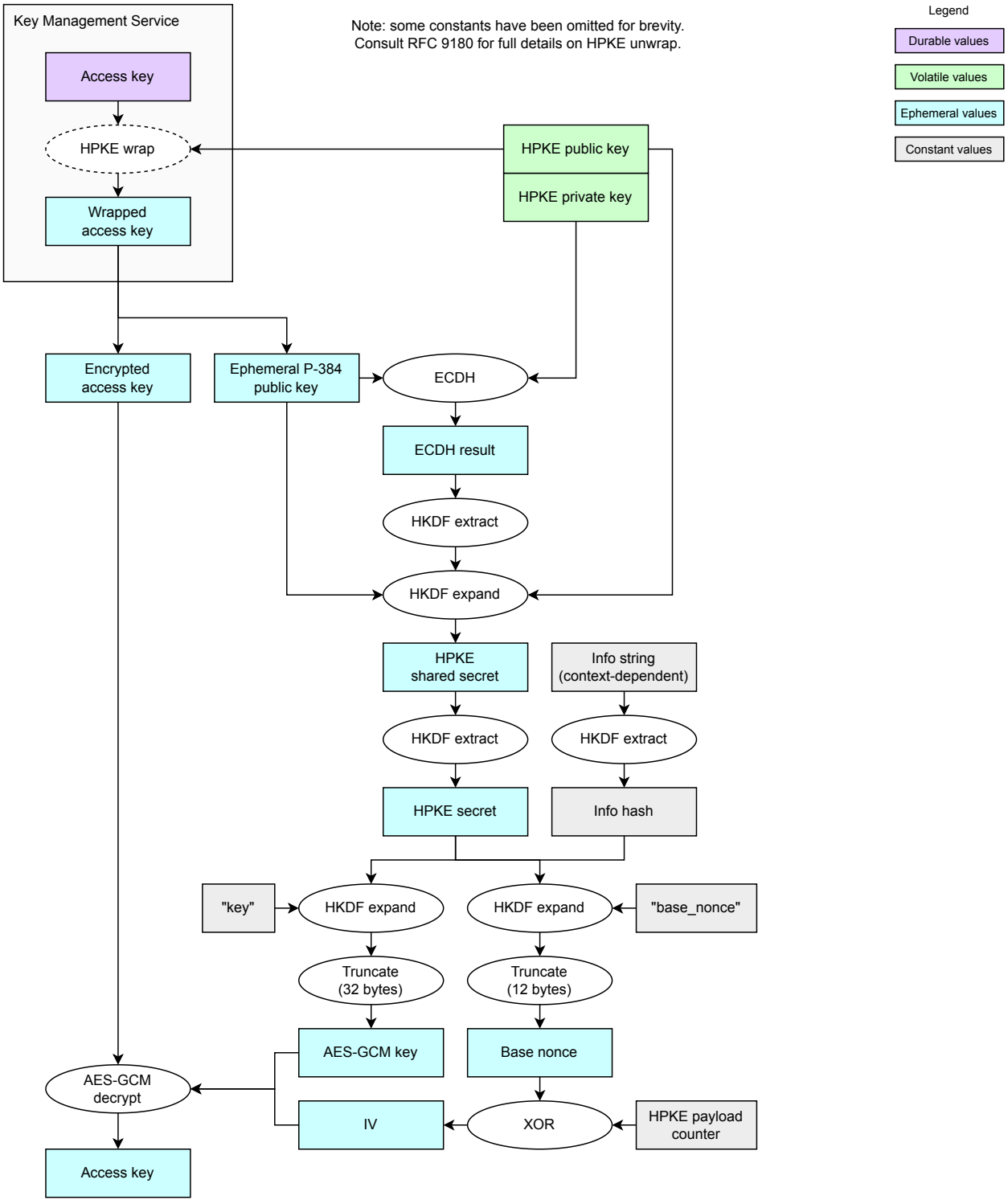

Figure 3: MPK access key unwrap with ECDH

4.5.3.1.3 MPK access key unwrap (Hybrid ML-KEM + ECDH)

The only difference here from the above flow is how the HPKE Shared Secret is derived. Operations which produce an AES key and IV from that shared secret are identical to the prior flow, and are omitted here for brevity.

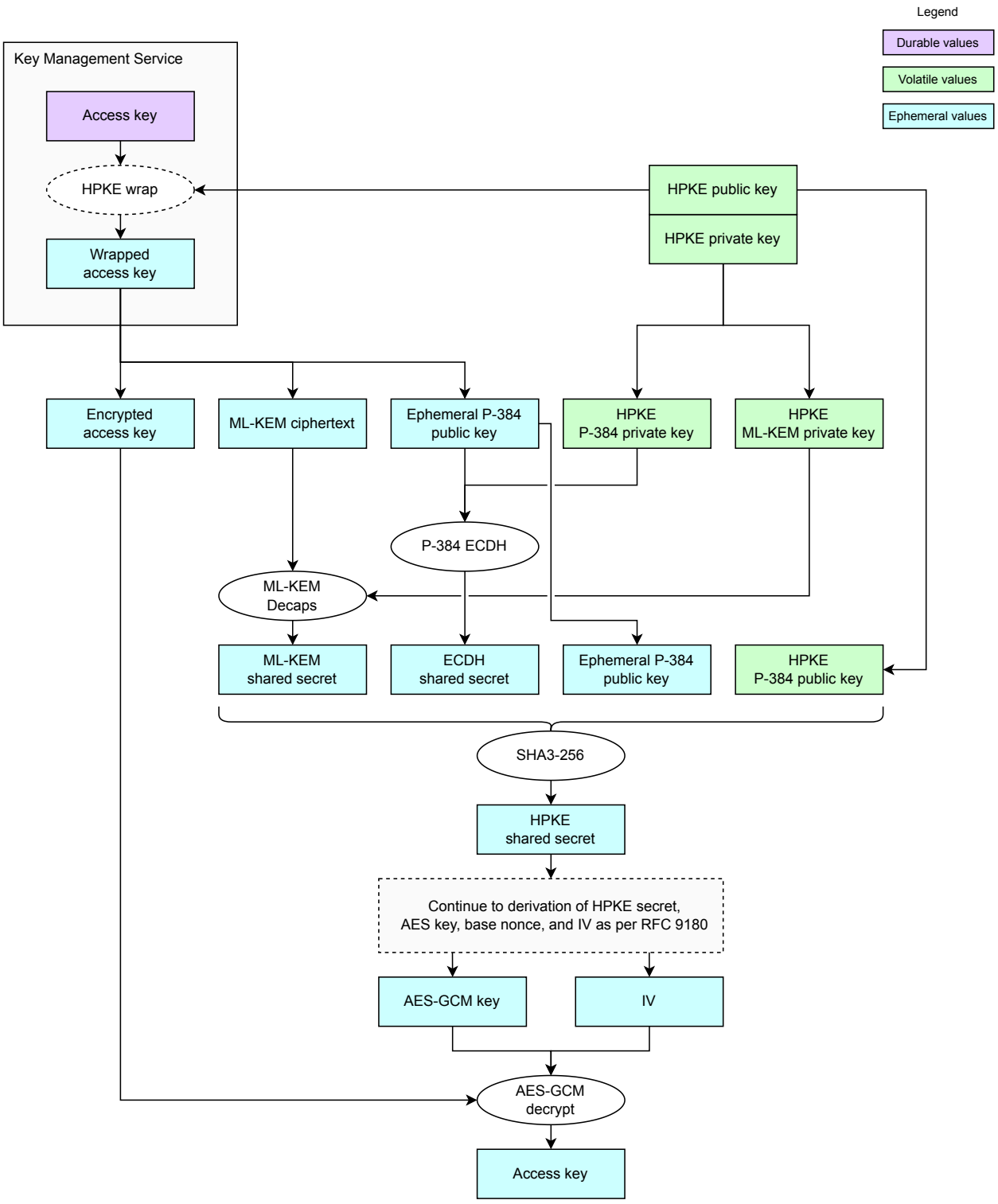

Figure 4: MPK access key unwrap with hybrid ML-KEM + ECDH

4.5.3.2 MPK lifecycle

MPKs can be generated, made ready, and have their access keys rotated.

4.5.3.2.1 MPK generation

Controller firmware may request that KMB generate a MPK, bound to a given access key. KMB performs the following steps:

1. Unwrap the given MPK access key using the HPKE keypair held within KMB.
2. Randomly generate a MPK.
3. Derive a MPK encryption key from the HEK and the decrypted access key.
4. Encrypt the MPK to the MPK encryption key.
5. Return the encrypted MPK to the controller firmware.

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

To mitigate against cryptographic attacks on the HPKE keypair that rely on repeated invocations of this command, this command is rate-limited.

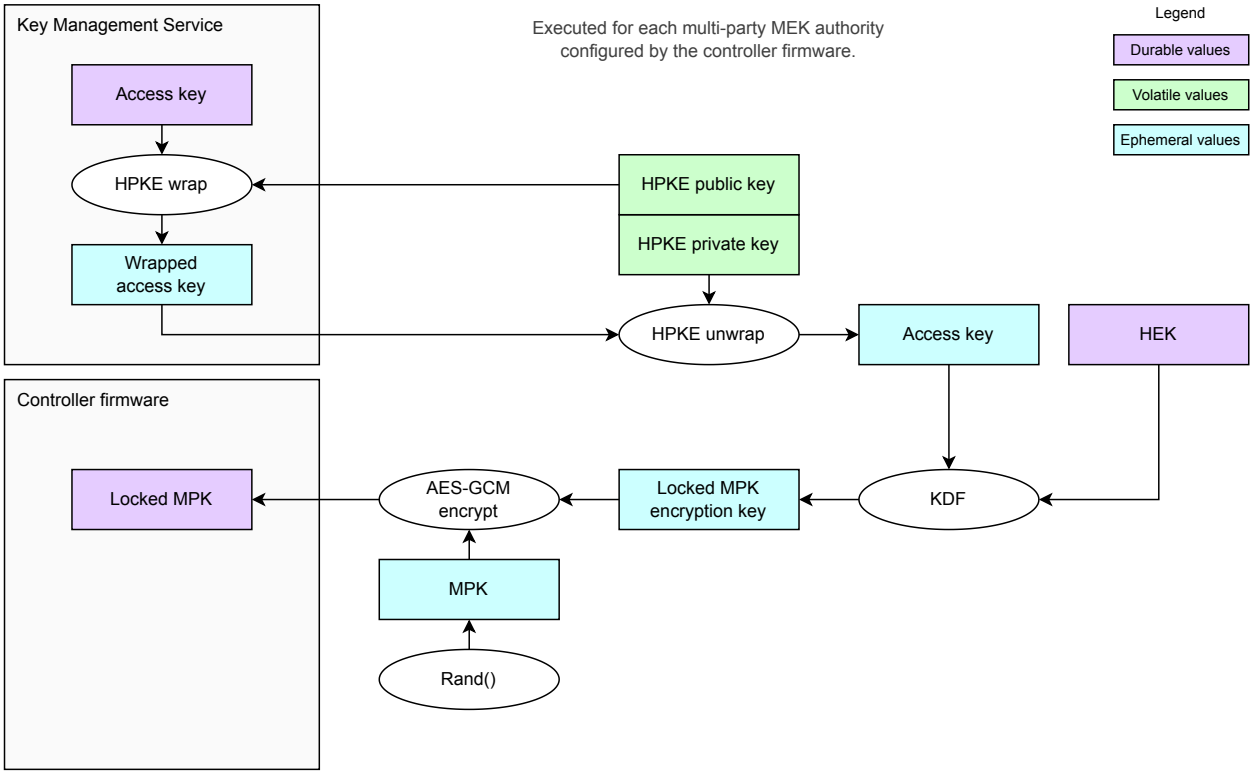

Figure 5: MPK generation

4.5.3.2.2 MPK readying

Encrypted MPKs stored at rest in persistent storage are considered “locked”, and must be made ready before they can be used to load an MEK. Ready MPKs are also encrypted when handled

by controller firmware, to the Ready MPK Encryption Key. Ready MPKs do not survive across device reset.

To ready a MPK, KMB performs the following steps:

1. Unwrap the given MPK access key using the HPKE keypair held within KMB.
2. Derive the MPK encryption key from the HEK and the decrypted access key.
3. Decrypt the MPK using the MPK encryption key.
4. Encrypt the MPK using the Ready MPK Encryption Key.
5. Return the re-encrypted “ready” MPK to the controller firmware.

Controller firmware may then stash the encrypted ready MPK in volatile storage, and later provide it to the KMB when loading an MEK, as described in Section 4.5.5.

To mitigate against cryptographic attacks on the HPKE keypair that rely on repeated invocations of this command, this command is rate-limited.

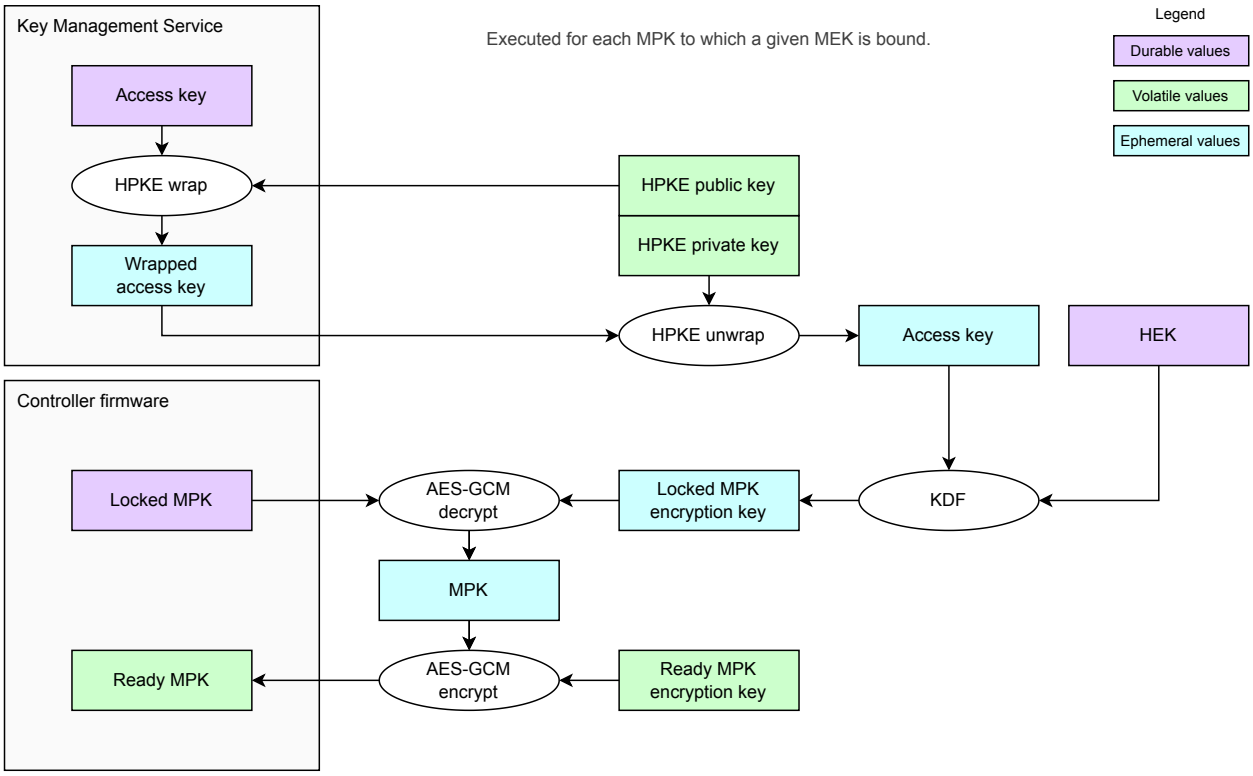

Figure 6: MPK readying

4.5.3.2.3 MPK access key rotation

The access key to which a MPK 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. This is accomplished using a slight variation on the usual access key import flow. When a new access key is provided to KMB during a rotation, the new access key is double-encrypted: first to the old access key, and then to the HPKE public key. KMB performs the following steps:

1. Unwrap the given old access key and encrypted new access key using the HPKE keypair held within KMB.
2. Decrypt the new access key using the old access key.
3. Derive the old MPK encryption key from the HEK and the decrypted old access key.
4. Derive the new MPK encryption key from the HEK and the decrypted new access key.
5. Decrypt the MPK using the old MPK encryption key.
6. Encrypt the MPK using the new MPK encryption key.
7. Return the re-encrypted MPK to the controller firmware.

Controller firmware then zeroizes the old encrypted MPK and stores the new encrypted MPK in persistent storage.

To mitigate against cryptographic attacks on the HPKE keypair that rely on repeated invocations of this command, this command is rate-limited.

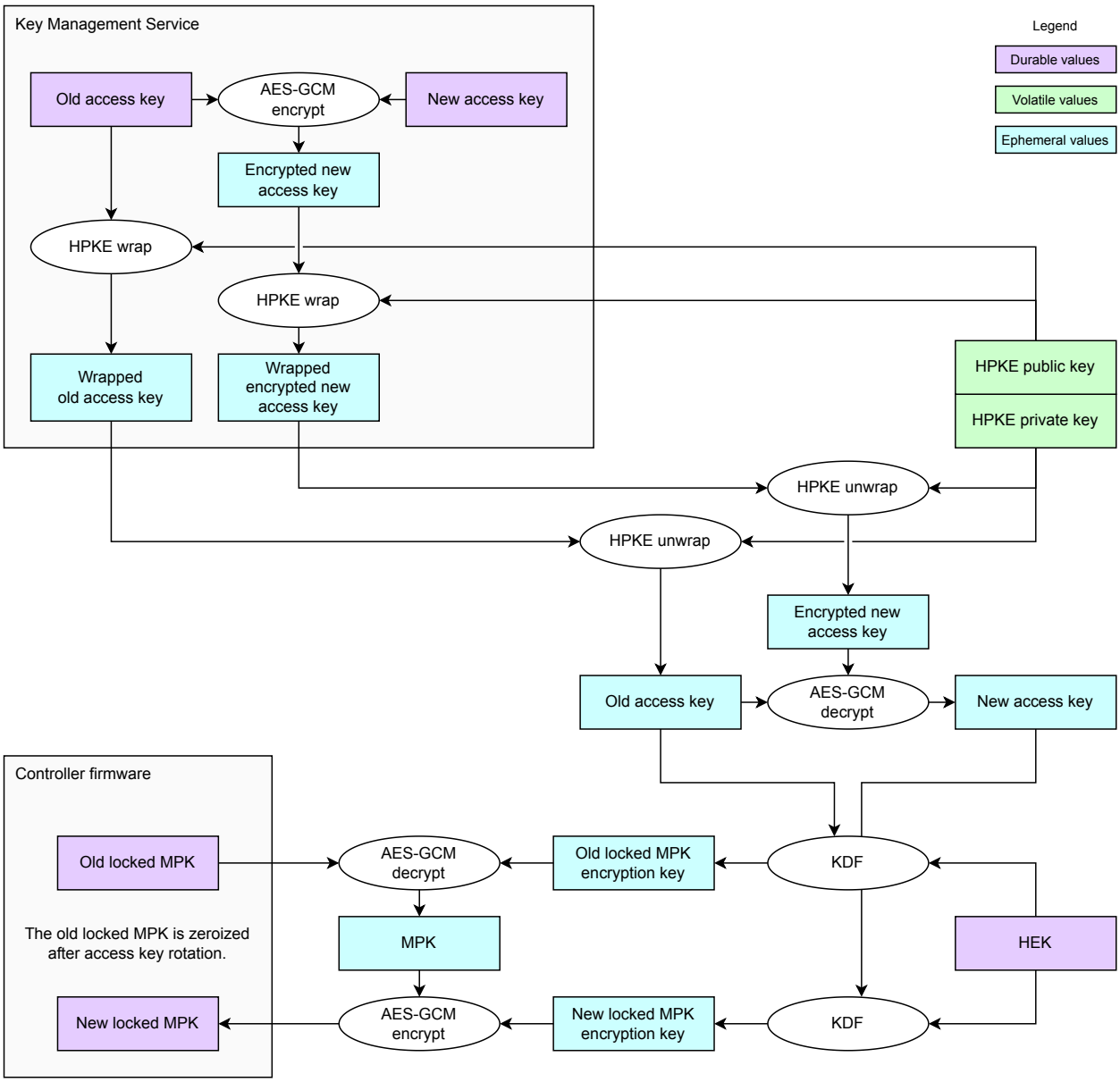

Figure 7: MPK access key rotation

4.5.4 DPKs

Controller firmware provides the DPK (Data Protection Key) to KMB when generating, loading, or deriving an MEK. The DPK is used in a KDF, along with the HEK, SEK, and MPKs, to derive the MEK secret, as illustrated in Section 4.5.5.3.

4.5.4.1 Use when generating or loading an MEK

In this flow, the DPK may be encrypted by a user's Opal C_PIN. Legacy controller firmware logic which decrypts MEKs can be repurposed to produce the DPK.

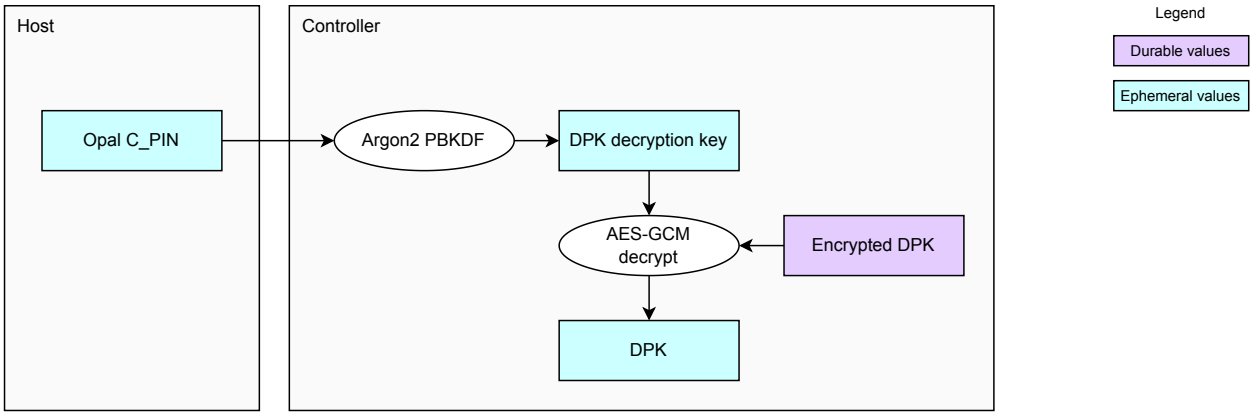

Figure 8: Example controller flow to decrypt a DPK based on a host-provided Opal C_PIN

4.5.4.2 Use when deriving an MEK

In this flow, the DPK may be the imported key associated with a Key Per I/O key tag.

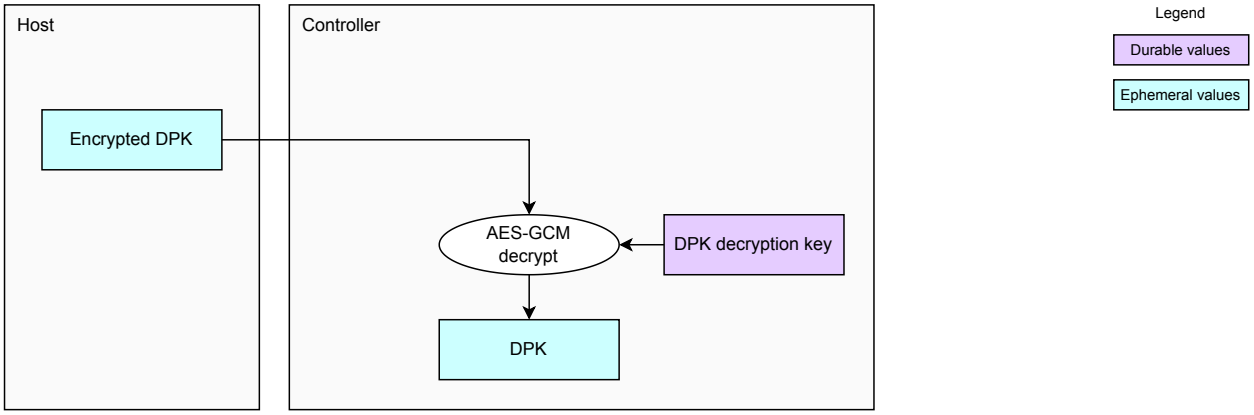

Figure 9: Example controller flow to accept an injected DPK

4.5.5 MEKs

KMB can encrypt randomly-generated MEKs, or compute derived MEKs.

4.5.5.1 Encrypting and decrypting a randomly-generated MEK

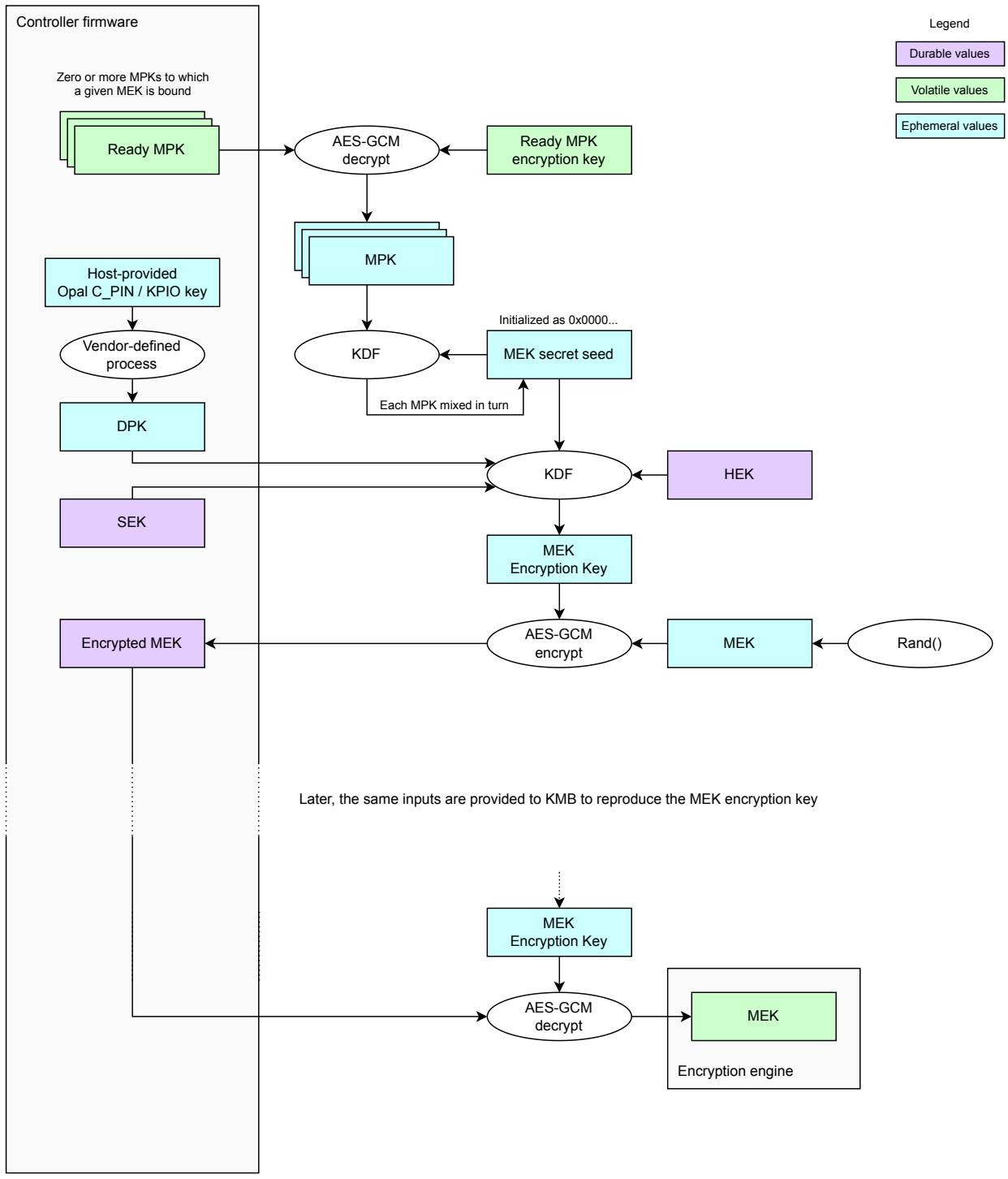

Figure 10: MEK encryption and decryption

4.5.5.2 Deriving an MEK

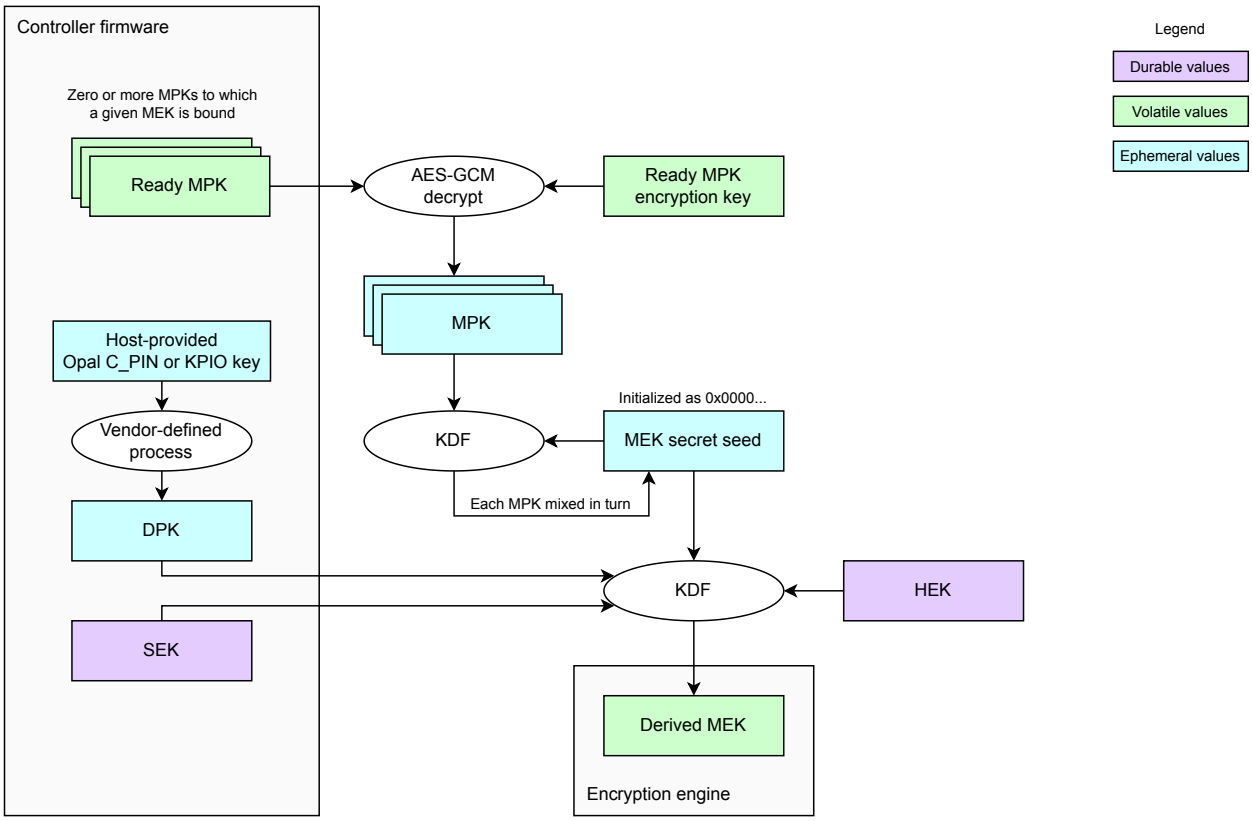

Figure 11: MEK derivation

4.5.5.3 MEK secret derivation

This diagram provides additional details on how the EPK, DPK, and MPKs are mixed together to produce the MEK secret, which then either encrypts/decrypts a random MEK or is used to compute a derived MEK.

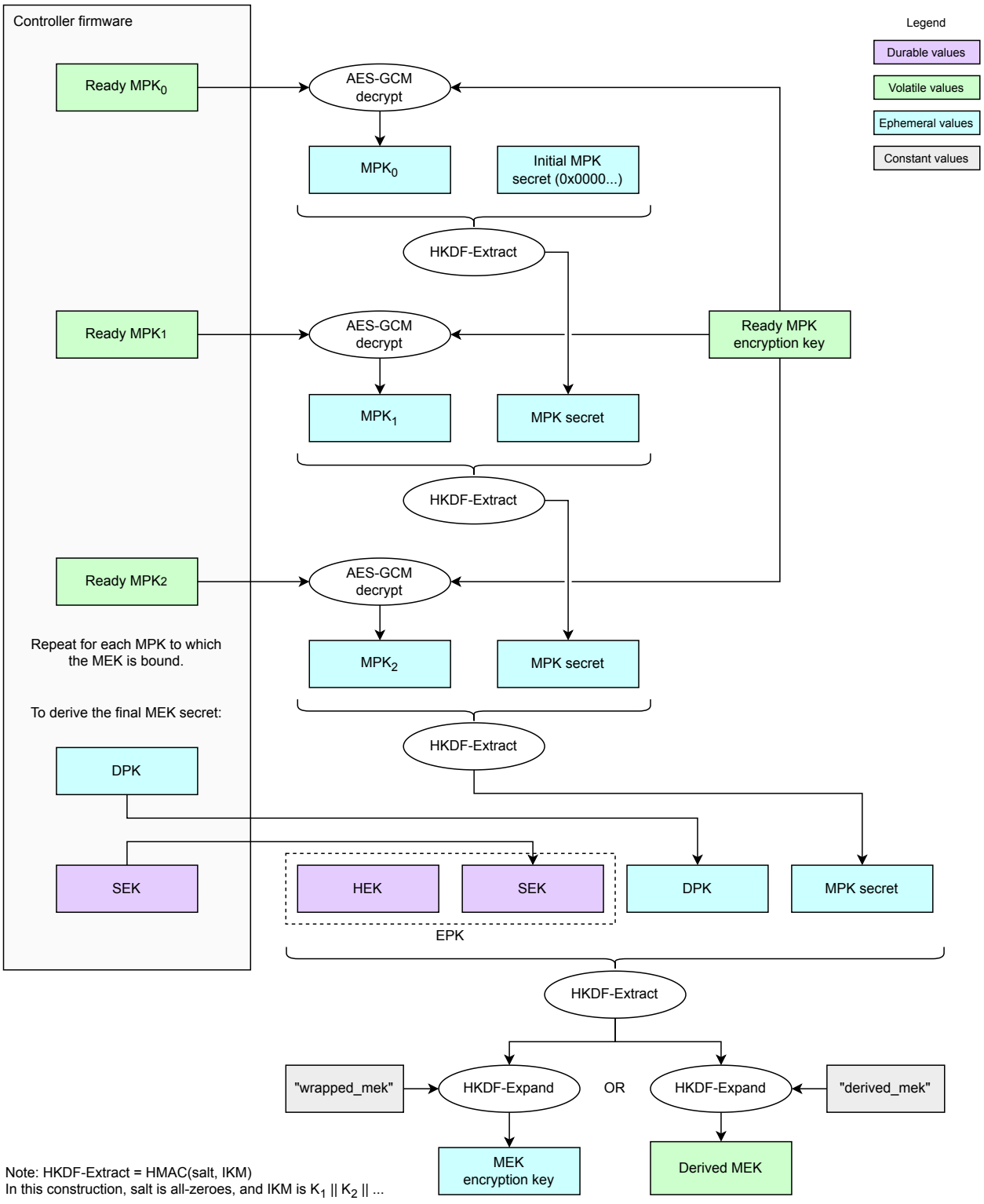

Figure 12: MEK secret derivation

4.5.6 HEKs and SEKs

KMB supports a pair of epoch keys: the HEK and SEK. Both must be available, i.e. non-zeroized, in order for MEKs to be loaded.

- The **HEK** is derived from secrets held in Caliptra's fuse block, and is never visible outside of Caliptra. If the HEK is zeroized, KMB does not allow MEKs to be loaded.
- The **SEK** is managed by controller firmware, and may be stored in flash. If the SEK is zeroized, controller firmware is responsible for enforcing that MEKs are not loaded.

Zeroizing either the HEK or SEK is equivalent to performing a cryptographic purge. HEK zeroization is effectively a "hard" purge, as it is highly difficult to recover secrets from a zeroized fuse bank.

The HEK is derived from a series of N 256-bit ratchet secrets present in a fuse bank, dubbed R0..RN-1, where N is between 4 and 16 inclusive. The vendor is responsible for determining the number of ratchet secrets available in fuses. These ratchet secrets have the following properties:

- Can individually transition from all-zeroes → randomized → all-ones. RX is only randomized once RX-1 has transitioned to all-ones.
- Programmable via the Caliptra fuse controller.
- Only readable by Caliptra Core, via fuse registers.
  - Internally, the fuse registers will be treated like the DICE UDS, in that their contents can only be deposited into Key Vault slots, without direct visibility by Caliptra firmware.

Caliptra features a Stable Identity Key (SIK), which is a non-ratchetable secret derived from the DICE UDS, the current value of the field entropy fuse bank, as well as the current lifecycle and debug states. The SIK can be used to protect long-lived data. Caliptra's Stable Identity feature is implemented in terms of a key ladder, with different keys for different firmware SVNs, with the property that firmware whose SVN is X can only obtain Stable Identity keys bound to major versions less than or equal to X. The SIK used by KMB is the key ladder secret bound to SVN 0, ensuring that firmware of any SVN can wield it.

The HEK is derived from the SIK, along with the currently-active ratchet secret. In normal cases, if the currently-active ratchet secret is unprogrammed or zeroized, the HEK is disabled and no MEKs are allowed to be loaded. KMB exposes a command that allows the owner to permanently opt into an operating mode where the HEK is derived solely from the SIK. In this mode, data written to the storage device can no longer be purged via HEK zeroization.

To satisfy FIPS 140 zeroization requirements once the HEK can no longer be zeroized, the FIPS boundary will need to include SEK zeroization functionality in the storage controller firmware.

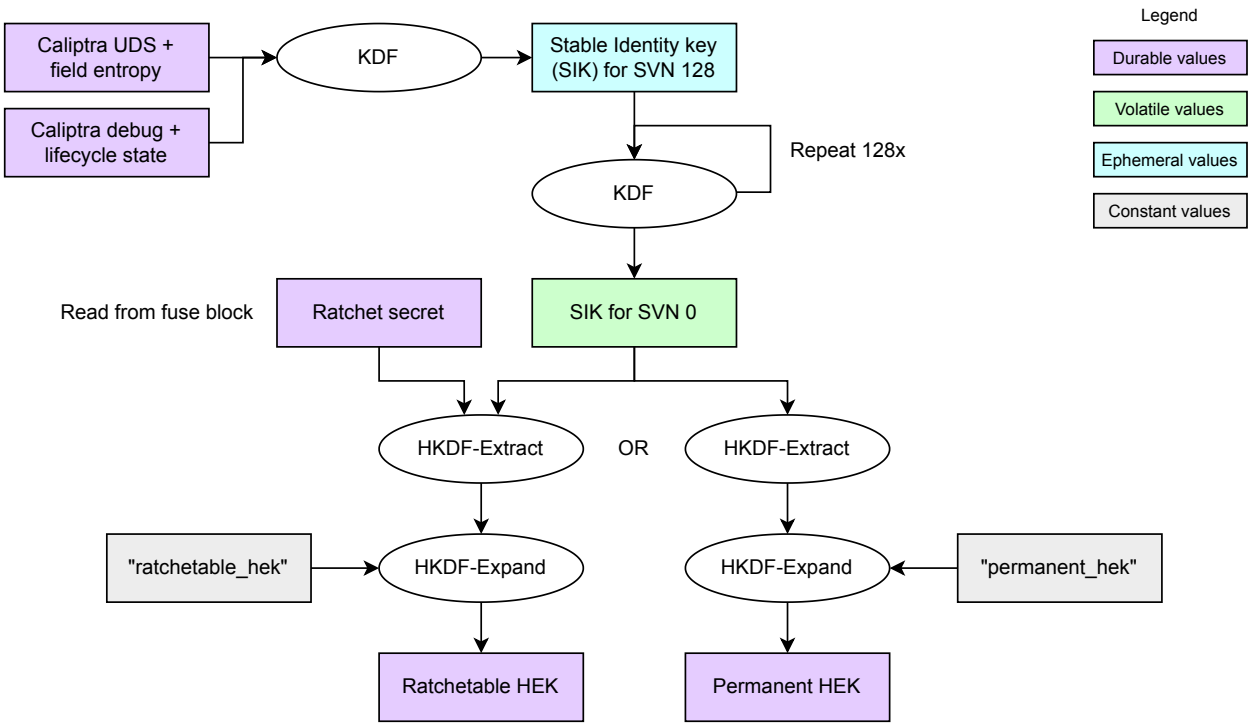

Figure 13: HEK derivation

4.5.6.1 HEK lifecycle

A device out of manufacturing is required to have programmed the first HEK. KMB exposes the following commands to manage the HEK lifecycle:

Command	Description
PROGAM_NEXT_HEK	Programs a random ratchet secret into the next slot. Upon success, if the SEK is programmed, MEKs may be loaded.
ZEROIZE_CURRENT_HEK	Sets the current ratchet secret fuses to all-ones. May be re-attempted if an error caused the active fuse bank to be left in an invalid state. Upon success, MEKs may not be loaded.
ENABLE_PERMANENT_HEK	Enables a mode where the HEK is derived solely from SIK. May be re-attempted if an error caused the active fuse bank to be left in an invalid state. Upon success, if the SEK is programmed, MEKs may be loaded.

If an error causes the active fuse bank to be left in an invalid state, MEKs may not be loaded. If this state persists, the device is unusable.

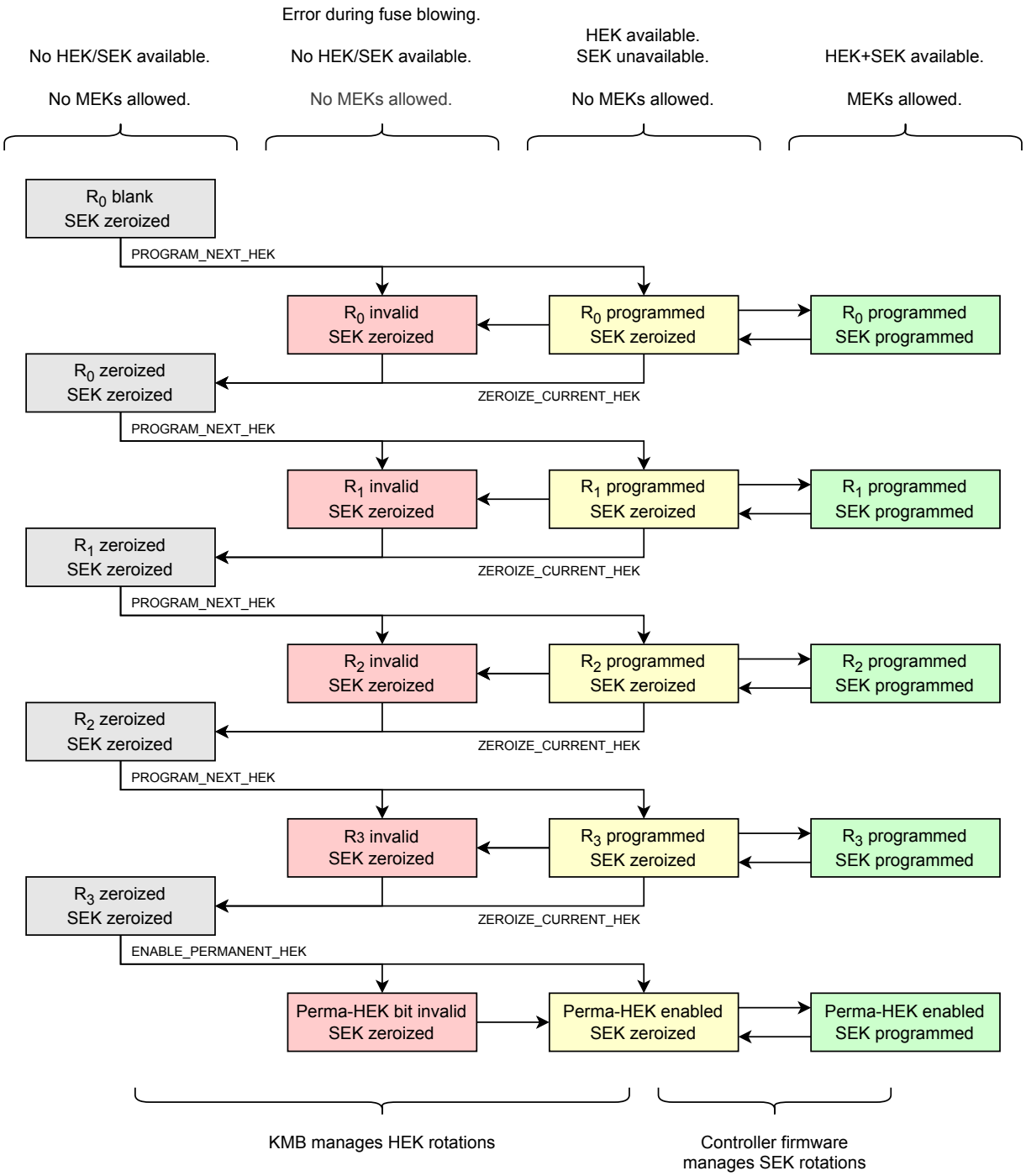

Figure 14: HEK & SEK state machine with four ratchet slots

4.5.6.2 SEK lifecycle

Controller firmware is responsible for enforcing that SEK programming and zeroization follows the state machine above. Specifically:

- The SEK may only be programmed once the HEK is available.
- The HEK may only be zeroized once the SEK is zeroized.

4.5.7 Random key generation via DRBG

KMB generates multiple kinds of random keys. It does so using randomness obtained from a DRBG, which is seeded from Caliptra's TRNG and which may be updated with entropy from the host.

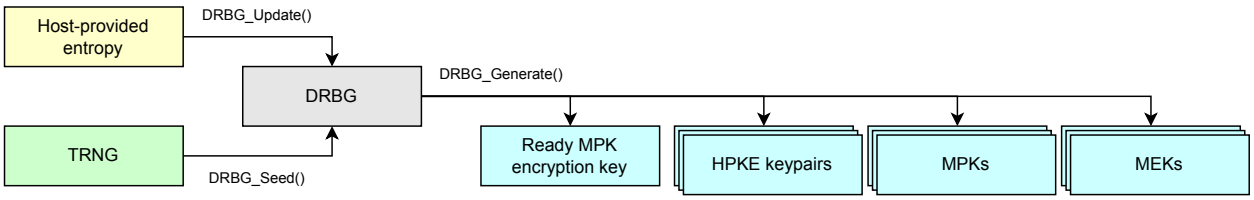

Figure 15: DRBG usage

4.5.8 Boot-time initialization

At initial boot, KMB firmware performs the following steps:

- Determines whether an HEK is available, based on the fuse configuration. If an HEK is unavailable, KMB enters a mode where MEKs are not allowed to be loaded.
- Initializes random HPKE keypairs for each supported algorithm, and maps them to unique handles.

4.6 Interfaces

OCP L.O.C.K. defines two interfaces:

- The **encryption engine interface** is exposed from the vendor-implemented encryption engine to KMB, and defines a standard mechanism for programming MEKs and control messages.
- The **mailbox interface** is exposed from KMB to storage controller firmware, and enables the controller to manage MEKs and associated keys.

4.6.1 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. MEKs are generated or derived within KMB and used by the encryption engine to encrypt and decrypt user data. This interface is used to load MEKs from KMB to the 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.

4.6.1.1 Overview

The encryption engine uses a stored MEK to encrypt and decrypt user data. For the purposes of this specification, the entity within the encryption engine used to store the MEKs is called the key cache. Each encryption and decryption of user data is coupled to a specific MEK which is stored in the 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.

To ensure that MEKs are only ever visible to KMB and the encryption engine, KMB is the only entity which can load and unload (metadata, aux, MEK) tuples. Controller firmware arbitrates all operations in the KMB to encryption engine interface, and is therefore responsible for managing which MEK is loaded in the key cache. Controller firmware has full control on metadata and optional aux. Figure 16 is an illustration of the KMB → encryption engine interface which shows:

- The tuple for loading an MEK.
- The metadata for unloading an MEK.
- An example of a key cache configuration within the encryption engine.

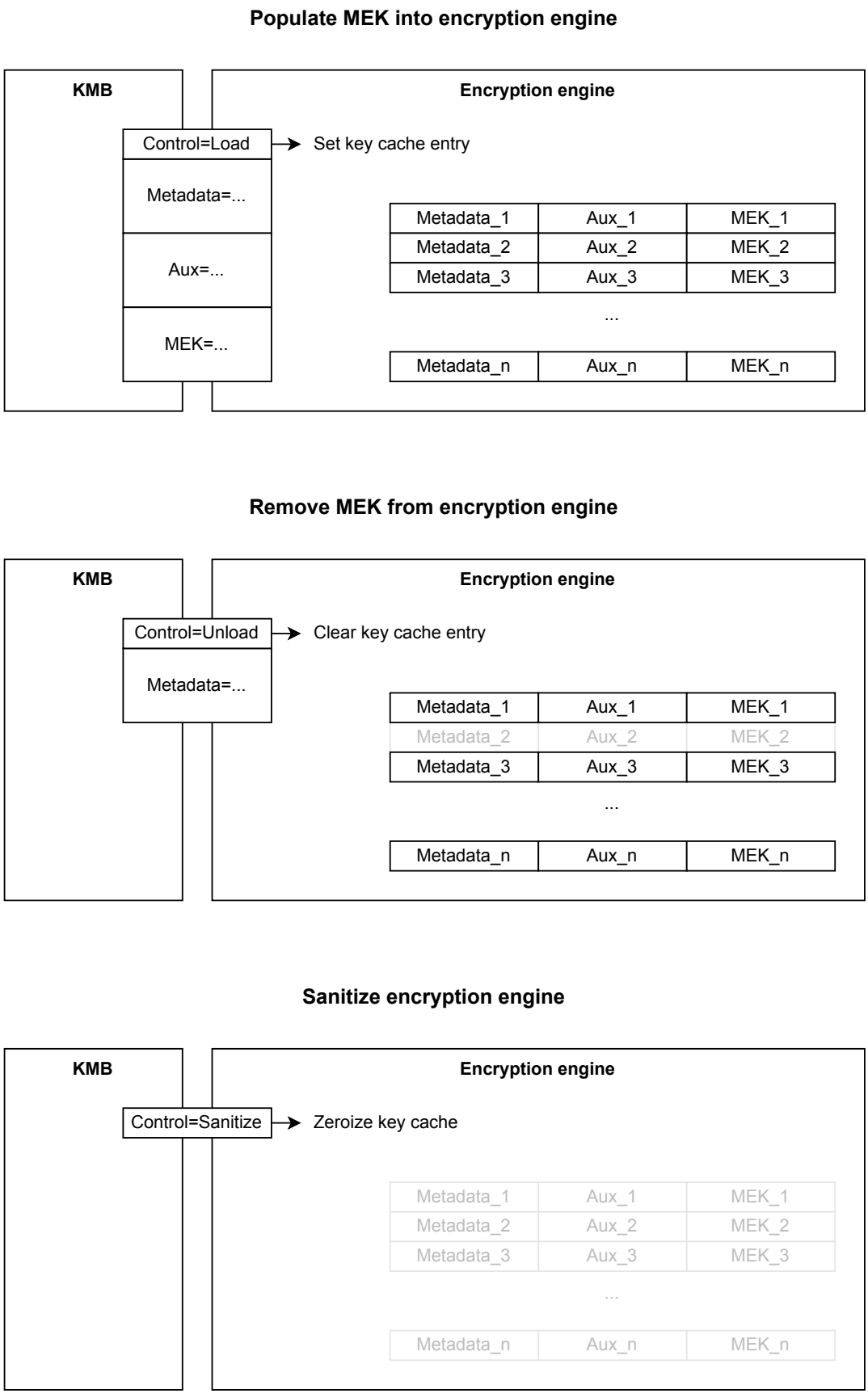

Figure 16: KMB to encryption engine SFR interface

4.6.1.2 Special Function Registers

KMB uses Special Function Registers (SFRs) to communicate with the encryption engine as shown in Table 1 and each of the following subsections which 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 (METD)	SFR_BASE + 10h	14h	Register to provide metadata
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 within KMB. The integrator should make sure that KMB can access these SFRs through these addresses.

4.6.1.2.1 Control register

Table 2 defines the Control register used to sequence the execution of a command and obtain the status of that command.

Table 2: 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 the command specified by the CMD field. See Table 3. Encryption engine error codes are surfaced back to controller firmware. 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.
15:6	RO	0h	Reserved
5:2	RW	0h	**Command (CMD):** This field specifies the command to execute or the command associated with the reported status. See Table 4.

(continued on next page)

(continued from previous page)

Bits	Type	Reset	Description
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 0000b.
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.

Table 3: CTRL error codes

Value	Description
0h	Command successful
1h to 3h	Reserved
4h to Fh	Vendor Specific

Table 4: CTRL command codes

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	**Zeroize**: Unload all of the MEKs from the encryption engine (i.e., zeroize the encryption engine MEKs).
4h to Fh	Reserved

From the KMB, the Control register is the register to write a command and receive its execution result. From its counterpart, the encryption engine, the Control register is used to receive a command and write its execution result.

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

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

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

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

Since the Control register is in fact a part of the encryption engine whose implementation can be unique to 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.

4.6.1.2.2 Metadata register

Table 5 defines the Metadata register used to pass additional data related to the MEK.

Table 5: Offset SFR_Base + 10h: METD – Metadata

Bytes	Type	Reset	Description
19:00	RW	0h	**Metadata (METD):** This field specifies metadata that is vendor specific and specifies the entry in the encryption engine for the MEK.

The KMB and the encryption engine must be the only components which have access to MEKs. Each MEK is associated with a unique identifier, which may be visible to other components, in

order for the MEK to be used for any key-related operations including data I/O. The **METD** field is used as such an identifier.

Instead of generating a random and unique identifier within the KMB while loading an MEK, the KMB takes an **METD** value as input from the controller firmware and write to the **METD** 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 the **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 17.

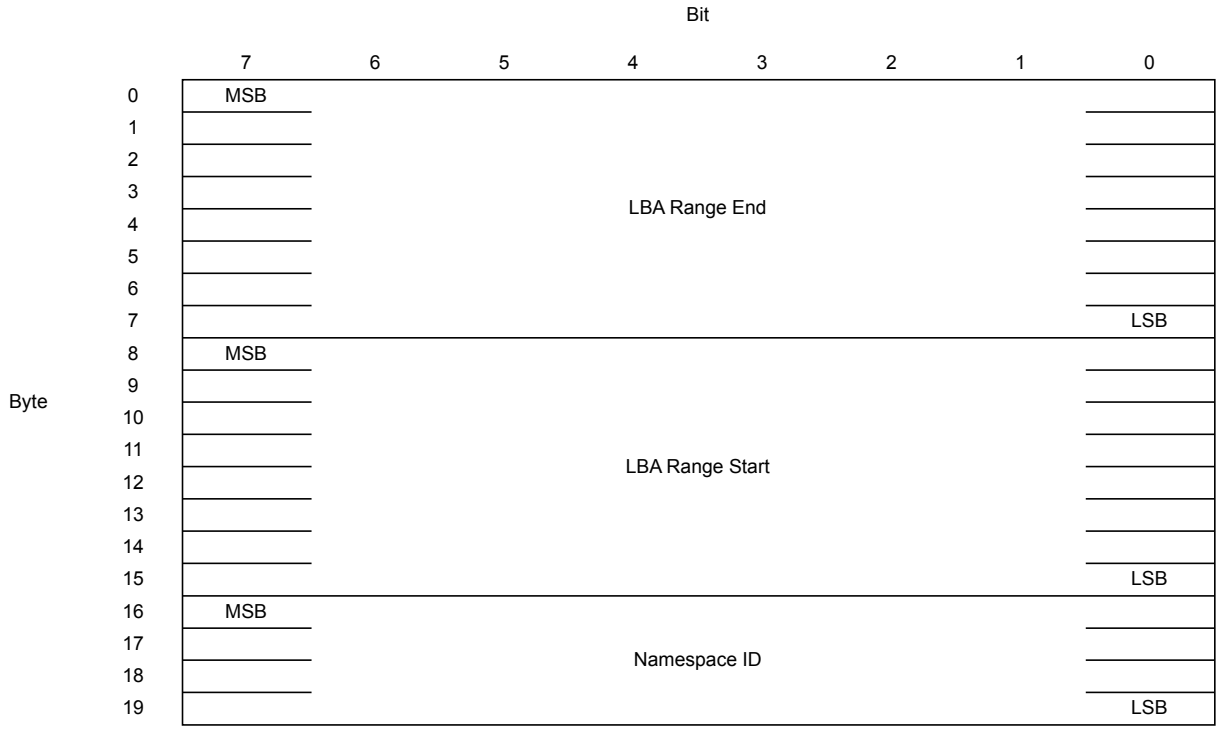

Figure 17: LBA range based metadata format

Address-based encryption is not the only encryption mechanism in SSDs. For example, in TCG Key Per I/O, an MEK is selected by a key tag, which does not map to an address. Figure 18 shows an example of **METD** in such cases.

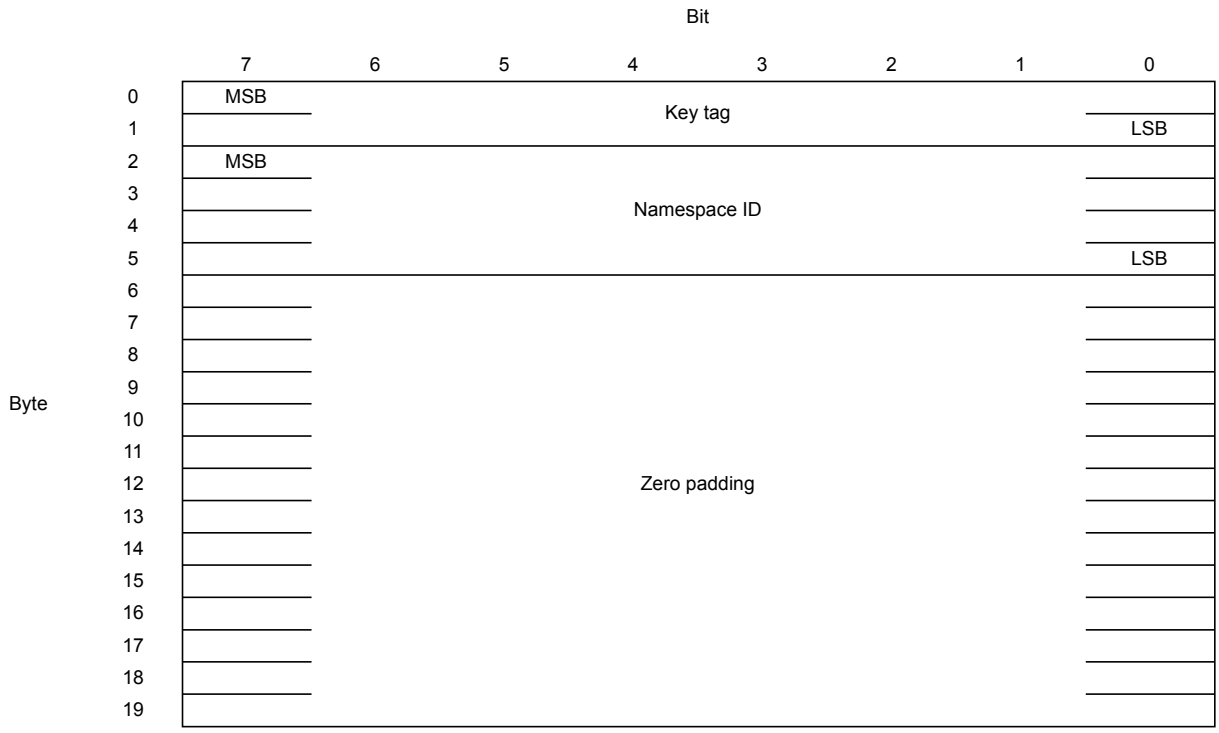

Figure 18: Key tag based metadata format

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.

4.6.1.2.3 Auxiliary Data register

Table 6 defines the Auxiliary Data register used to pass additional vendor-specific data related to the MEK.

Table 6: Offset SFR_Base + 20h: AUX – Auxiliary Data

Bytes	Type	Reset	Description
31:00	RW	0h	**Auxiliary Data (AUX):** This field specifies auxiliary data associated to the MEK.

The **AUX** field supports vendor-specific features on MEKs. The KMB itself only supports fundamental functionalities in order to minimize attack surfaces on MEKs. Moreover, vendors are free to design and implement their own MEK-related functionality within the encryption engine, as long as that functionality cannot 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 facilitates this association.

When the controller firmware instructs the KMB to load an 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 simple use case of the **AUX** field is to store an offset of initialization vector or nonce. It can also be used in a more complicated use case. Here is an example. Suppose that there exists a vendor who wants to design a system which supports several modes of operation through the encryption engine while using the KMB. Then, a structure of **AUX** value as on Figure 19 can be used.

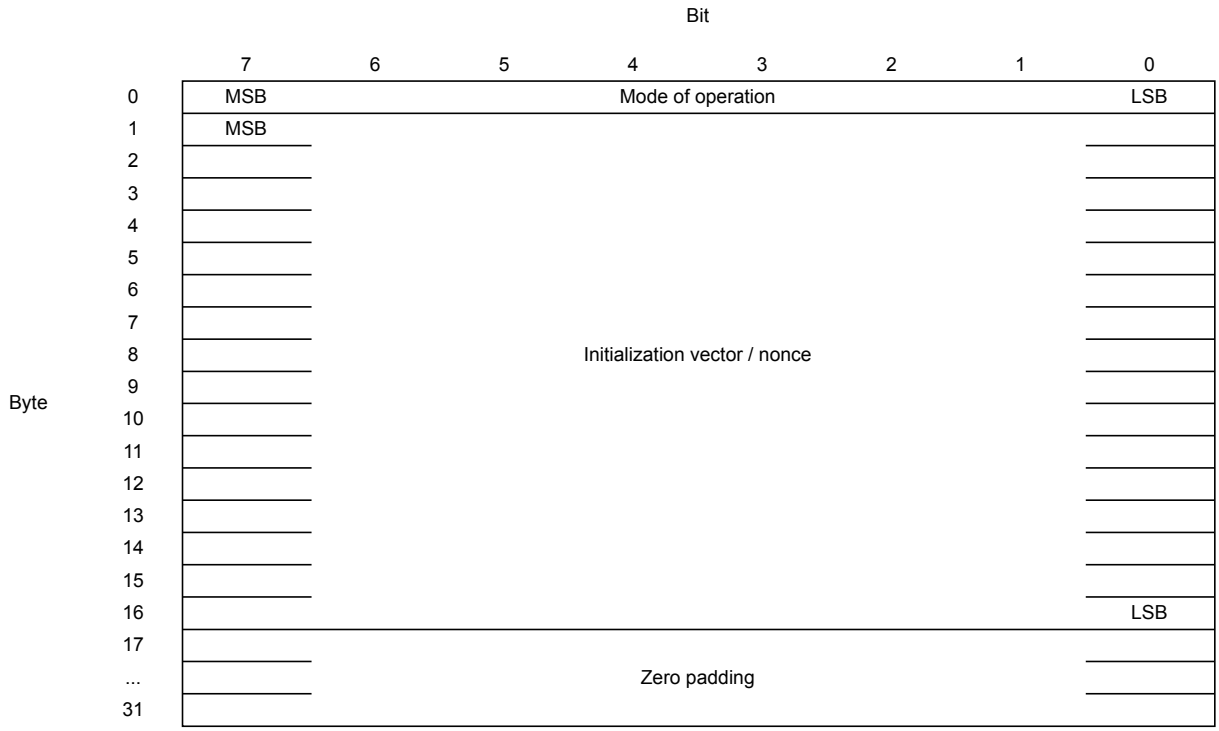

Figure 19: Auxiliary data format example

When the controller firmware instructs KMB to load an MEK, 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.

4.6.1.2.4 Media Encryption Key register

Table 7: Offset SFR_Base + 40h: MEK – Media Encryption Key

Bytes	Type	Reset	Description
63:00	WO	0h	**Media encryption key:** This field specifies a 512-bit encryption key.

The encryption engine is free to interpret the provided key in a vendor-defined manner. One sample interpretation for AES-XTS-512 is presented in Figure 20.

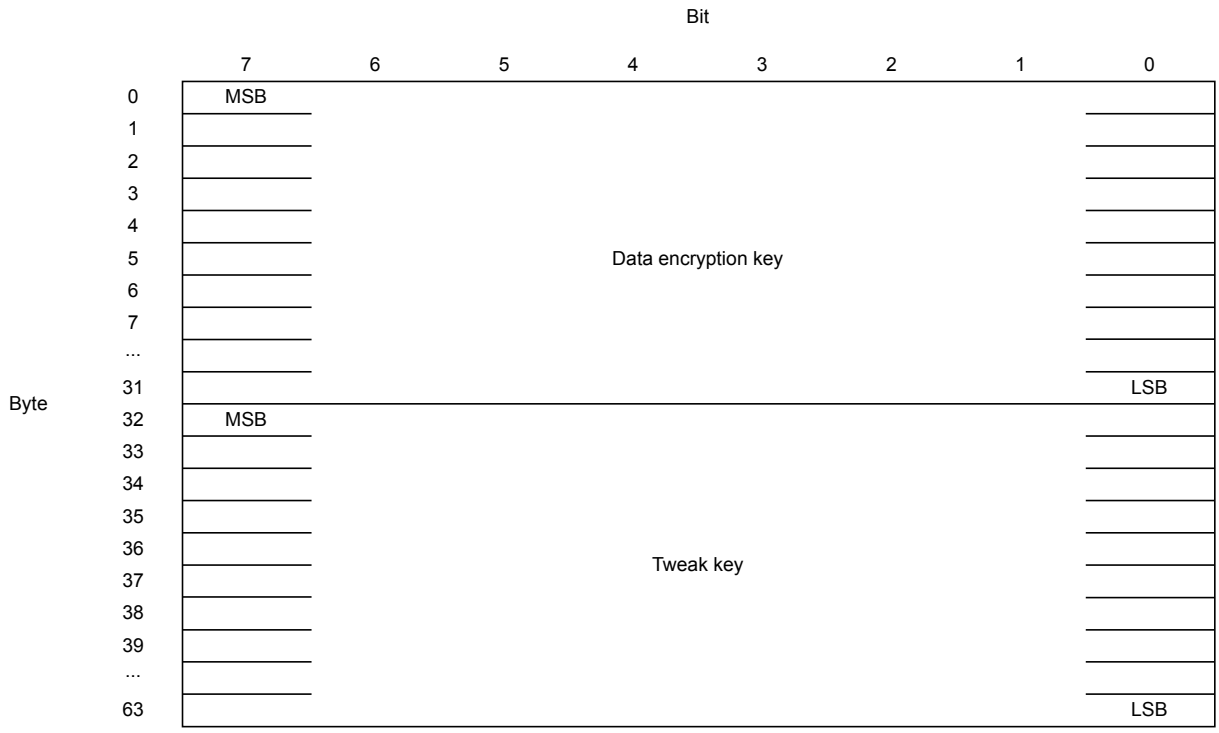

Figure 20: MEK format example for AES-XTS-512

If an algorithm used by the encryption engine does not require 512 bits of key material, the encryption engine is free to disregard unused bits.

Within KMB, loaded MEKs are only ever present in the Key Vault, so that they can be protected against any firmware-level attacks. KMB will write MEKs into the encryption engine's key cache using the DMA engine. Given an index and a destination identifier, the DMA engine will copy the key value stored in the given key vault slot to the destination address to which the DMA engine translates the destination identifier.

4.6.1.3 AES-XTS key validation requirements

FIPS 140-3 IG section C.I [6] states that in AES-XTS, the key is "parsed as the concatenation of two AES keys, denoted by *Key_1* and *Key_2*, that are 128 [or 256] bits long… The module **shall** check explicitly that *Key_1* ≠ *Key_2*, regardless of how *Key_1* and *Key_2* are obtained."

The encryption engine will be responsible for performing this check when in AES-XTS mode.

4.6.1.4 KMB command sequence

Figure 21 shows a sample command execution. This is an expected sequence when the controller firmware instructs the KMB to load an MEK. The internal behavior of the encryption engine is one of several possible mechanisms, and can be different per vendor.

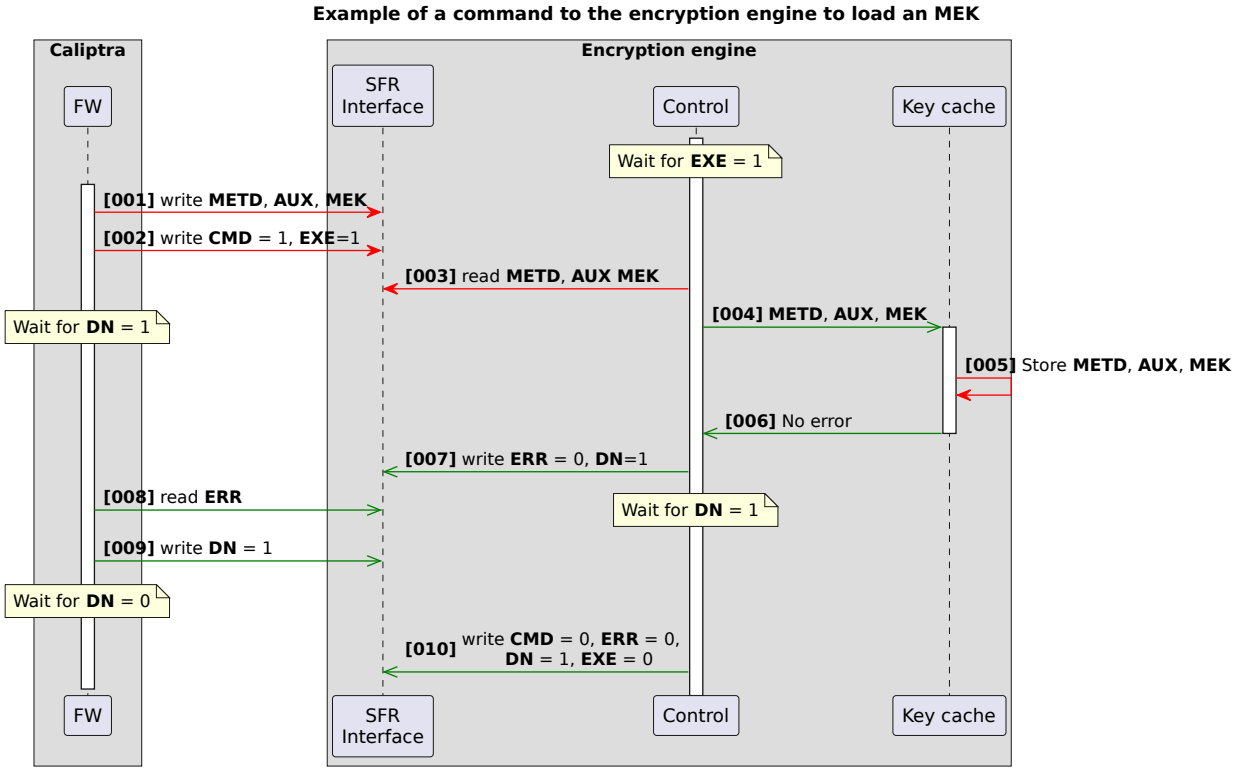

Figure 21: Command execution example for loading an MEK

4.6.2 Controller → KMB interface

This section provides the mailbox commands exposed by Caliptra as part of OCP L.O.C.K.

Each of these commands returns a `fips_status` field. This provides an indicator of whether KMB is operating in FIPS mode. The values for this field are as follows:

Value	Description
0h	FIPS mode enabled.
1h to FFFFh	Reserved.

4.6.2.1 GET_STATUS

Exposes a command that allows controller firmware 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 8: GET_STATUS input arguments

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

Table 9: 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.
reserved	u32[4]	Reserved.
engine_ready	u32	Ready status of the storage device crypto engine. OCP L.O.C.K. defines the low range (0h-FFFFh), vendor defines the high range (10000h-FFFFFFFFh). • Byte 0 Bit 0: 1 = Ready, 0 = Not ready

4.6.2.2 GET_ALGORITHMS

Exposes a command that allows controller firmware to determine the types of algorithms supported by KMB for endorsement, KEM, MPK, and access key generation.

Command Code: 0x4741_4C47 (“GALG”)

Table 10: GET_ALGORITHMS input arguments

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

Table 11: 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.
reserved	u32[4]	Reserved.

(continued on next page)

(continued from previous page)

Name	Type	Description
endorsement_algorithms	u32	Identifies the supported endorsement algorithms: • Byte 0 bit 0: ecdsa_secp384r1_sha384 [7] • Byte 0 bit 1: ml-dsa-87 [8]
hpke_algorithms	u32	Identifies the supported HPKE algorithms: {kem/aead/kdf}_id • Byte 0 bit 0: 0x0011, 0x0002, 0x0002 [5] • Byte 0 bit 1: 0x0a25, 0x0002, 0x0002 [9]
mpk_algorithms	u32	Indicates the size of MPKs: • Byte 0 bit 0: 256 bits
access_key_algorithm	u32	Indicates the size of access keys: • Byte 0 bit 0: 256 bits, with a 128-bit truncated SHA384 ID

4.6.2.3 CLEAR_KEY_CACHE

This command unloads all MEKs in the encryption engine and deletes all keys in KMB.

Command Code: 0x434C_4B43 (“CLKC”)

Table 12: CLEAR_KEY_CACHE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command.
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete.

Table 13: CLEAR_KEY_CACHE 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.

(continued on next page)

(continued from previous page)

Name	Type	Description
reserved	u32	Reserved.

4.6.2.4 ENDORSE_ENCAPSULATION_PUB_KEY

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

Command Code: 0x4E45_505B (“EEPK”)

Table 14: ENDORSE_ENCAPSULATION_PUB_KEY input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
kem_handle	u32	Handle for KEM keypair held in KMB memory.
endorsement_algorithm	u32	Endorsement algorithm identifier. If 0h, then just return public key.

Table 15: 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.
reserved	u32	Reserved.
pub_key_len	u32	Length of HPKE public key (Npk in RFC 9180).
endorsement_len	u32	Length of endorsement data. Zero if `endorsement_algorithm` is 0h.
pub_key	u8[pub_key_len]	HPKE public key.
endorsement	u8[endorsement_len]	DER-encoded X.509 certificate.

4.6.2.5 ROTATE_ENCAPSULATION_KEY

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

Command Code: 0x5245_4E4B (“RENK”)

Table 16: ROTATE_ENCAPSULATION_KEY input arguments

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

Table 17: 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.
reserved	u32	Reserved.
kem_handle	u32	Handle for new KEM keypair held in KMB memory.

4.6.2.6 GENERATE_MPK

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

Command Code: 0x4750_4D4B (“GPMK”)

Table 18: GENERATE_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
mpk_algorithm	u32	Indicates the size of MPKs. Only one bit shall be reported: • Byte 0 bit 0: 256 bits
info_len	u16	Length of the info argument.
info	u8[info_len]	Info argument to use with HPKE unwrap.

(continued on next page)

(continued from previous page)

Name	Type	Description
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key: • access_key_algorithm • kem_handle • kem_algorithm • kem_ciphertext • encrypted_access_key

Table 19: GENERATE_MPK 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.
reserved	u32	Reserved.
encrypted_mpk	EncryptedMpk	MPK encrypted to access_key.

4.6.2.7 REWRAP_MPK

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 MPK is decrypted using KDF(HEK, “MPK”, access_key_1). A new MPK is encrypted with the output of KDF(HEK, “MPK”, access_key_2). The new encrypted MPK is returned.

The Storage Controller stores the returned new encrypted MPK. The Storage Controller may attempt to do a trial decryption the new MPK without an error before deleting the old MPK. Controller firmware zeroizes the old encrypted MPK.

Command Code: 0x5245_5750 (“REWP”)

Table 20: REWRAP_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
info_len	u16	Length of the info argument.
info	u8[info_len]	Info argument to use with HPKE unwrap.

(continued on next page)

(continued from previous page)

Name	Type	Description
wrapped_access_key_1	WrappedAccessKey	KEM-wrapped access key: • access_key_algorithm • kem_handle • kem_algorithm • kem_ciphertext • encrypted_access_key
wrapped_enc_access_key_2	DoubleWrappedAccessKey	KEM-wrapped (access_key_2 encrypted to access_key_1).

Table 21: REWRAP_MPK 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.
reserved	u32	Reserved.
new_encrypted_mpk	EncryptedMpk	MPK encrypted to access_key_2.

4.6.2.8 READY_MPK

This command unwraps wrapped_access_key. Then the unwrapped access_key is used to decrypt locked_mpk using KDF(HEK, “MPK”, access_key). A “ready” MPK is encrypted with the Ready MPK Encryption Key. The encrypted ready MPK is returned.

Command Code: 0x5250_4D4B (“RPMK”)

Table 22: READY_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
info_len	u16	Length of the info argument.
info	u8[info_len]	Info argument to use with HPKE unwrap.

(continued on next page)

(continued from previous page)

Name	Type	Description
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key: • access_key_algorithm • kem_handle • kem_algorithm • kem_ciphertext • encrypted_access_key
locked_mpk	EncryptedMpk	MPK encrypted to the HEK and access key.

Table 23: READY_MPK 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.
reserved	u32	Reserved.
ready_mpk	EncryptedMpk	MPK encrypted to Ready MPK Encryption Key.

4.6.2.9 MIX_MPK

This command initializes the MEK secret seed if not already initialized or if `initialize` is set to 1, decrypts the specified MPK with the with the Ready MPK Encryption Key, and then updates the MEK secret seed in KMB by performing a KDF with the MEK secret seed and the decrypted MPK.

When generating an MEK, one or more MIX_MPK commands are processed to modify the MEK secret seed.

Command Code: 0x4D50_4D4B (“MPMK”)

Table 24: MIX_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
initialize	u32	If set to 1, the MEK secret seed is initialized before the given MPK is mixed. All other values reserved. Little-endian.

(continued on next page)

(continued from previous page)

Name	Type	Description
ready_mpk	EncryptedMpk	MPK encrypted to the Ready MPK Encryption Key.

Table 25: MIX_MPK 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.
reserved	u32	Reserved.

4.6.2.10 GENERATE_MEK

This command generates a random 512-bit MEK and encrypts it using the MEK encryption key, which is derived from the HEK, the MEK secret seed, and the given SEK and DPK.

The DPK may be a value decrypted by a user-provided C_PIN in Opal.

When generating an MEK, the MEK secret seed is initialized if no MPK has previously been mixed into the MEK secret seed.

Command Code: 0x474D_454B (“GMEK”)

Table 26: GENERATE_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
sek	u8[32]	“Soft epoch key”. May be rotated by the controller as part of a cryptographic purge.
dpk	u8[32]	“Data protection key”. May be a value decrypted by a user-provided C_PIN in Opal.

Table 27: GENERATE_MEK output arguments

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

(continued on next page)

(continued from previous page)

Name	Type	Description
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32	Reserved.
encrypted_mek	EncryptedMek	MEK encrypted to the derived MEK encryption key.

4.6.2.11 LOAD_MEK

This command decrypts the given encrypted 512-bit MEK using the MEK encryption key, which is derived from the HEK, the MEK secret seed, and the given SEK and DPK.

The DPK may be a value decrypted by a user-provided C_PIN in Opal.

When decrypting an MEK, the MEK secret seed is initialized if no MPK has previously been mixed into the MEK secret seed.

The decrypted MEK, 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.

Command Code: 0x4C4D_454B (“LMEK”)

Table 28: LOAD_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
sek	u8[32]	“Soft epoch key”. May be rotated by the controller as part of a cryptographic purge.
dpk	u8[32]	“Data protection key”. May be a value decrypted by a user-provided C_PIN in Opal.
metadata	u8[20]	Metadata for MEK to load into the drive crypto engine (i.e. NSID + LBA range).
aux_metadata	u8[32]	Auxiliary metadata for the MEK (optional; i.e. operation mode).
encrypted_mek	EncryptedMek	MEK encrypted to the derived MEK encryption key.
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command.
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete.

Table 29: LOAD_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.
reserved	u32	Reserved.

4.6.2.12 DERIVE_MEK

This command derives an MEK using the HEK, the MEK secret seed, and the given SEK and DPK.

When deriving an MEK, the MEK secret seed is initialized if no MPK has previously been mixed into the MEK secret seed.

The derived MEK, 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.

Command Code: 0x444D_454B (“DMEK”)

Table 30: DERIVE_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
sek	u8[32]	“Soft epoch key”. May be rotated by the controller as part of a cryptographic purge.
dpk	u8[32]	“Data protection key”.
metadata	u8[20]	Metadata for MEK to load into the drive crypto engine (i.e. NSID + LBA range).
aux_metadata	u8[32]	Auxiliary metadata for the MEK (optional; i.e. operation mode).
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command.
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete.

Table 31: DERIVE_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.
reserved	u32	Reserved.

4.6.2.13 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 32: UNLOAD_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
metadata	u8[20]	Metadata for MEK to unload from the drive crypto engine (i.e. NSID + LBA range).
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command.
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete.

Table 33: UNLOAD_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.
reserved	u32	Reserved.

4.6.2.14 ENUMERATE_KEM_HANDLES

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

Command Code: 0x4548_444C (“EHDL”)

Table 34: ENUMERATE_KEM_HANDLES input arguments

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

Table 35: ENUMERATE_KEM_HANDLES 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.
reserved	u32	Reserved.
kem_handle_count	u32	Number of KEM handles (N).
kem_handles	KemHandle[N]	List of (KEM handle value, KEM algorithm) tuples.

4.6.2.15 ZEROIZE_CURRENT_HEK

This command programs all un-programmed bits in the current HEK slot, so all bits are programmed. May re-attempt a previously-failed zeroize operation.

Command Code: 0x5A43_464B (“ZCFK”)

Table 36: ZEROIZE_CURRENT_HEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
hek_slot	u32	Current HEK slot to zeroize.

Table 37: ZEROIZE_CURRENT_HEK output arguments

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

(continued on next page)

(continued from previous page)

Name	Type	Description
fips_status	u32	Indicates if the command is FIPS approved or an error.

4.6.2.16 PROGRAM_NEXT_HEK

This command generates a random key and programs it into the next-available HEK slot.

Command Code: 504E_464B (“PNFK”)

Table 38: PROGRAM_NEXT_HEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
hek_slot	u32	Next HEK slot to program.

Table 39: PROGRAM_NEXT_HEK 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.
reserved	u32	Reserved.

4.6.2.17 ENABLE_PERMANENT_HEK

This command enables a state where the HEK is derived from non-ratchetable secrets. The command is only allowed once all HEK fuse slots are programmed and zeroized.

Command Code: 4550_464B (“EPFK”)

Table 40: ENABLE_PERMANENT_HEK input arguments

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

Table 41: ENABLE_PERMANENT_HEK 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.
reserved	u32	Reserved.

4.6.2.18 REPORT_EPOCH_KEY_STATE

This command reports the state of the epoch keys. The controller indicates the state of the SEK, while KMB internally senses the state of the HEK.

Command Code: 5245_4B53 (“REKS”)

Table 42: REPORT_EPOCH_KEY_STATE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
reserved	u32	Reserved.
sek_state	u16	SEK state. See Table 44.
nonce	u8[16]	Freshness nonce.

Table 43: REPORT_EPOCH_KEY_STATE 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.
reserved	u32	Reserved.
total_hek_slots	u16	Total number of HEK slots.
active_hek_slot	u16	Currently-active HEK slot.
hek_state	u16	State of the currently-active HEK. See Table 45.
next_action	u16	A bit field representation of the next actions that can be taken on the epoch keys. See Table 46.
eat_len	u16	Total length of the IETF EAT.
eat	u8[eat_len]	CBOR-encoded and signed IETF EAT. See Section A for the format.

Table 44: SEK state values

Value	Description
0h	ZEROIZED
1h	PROGRAMMED
2h to FFFFh	Reserved

Table 45: HEK state values

Value	Description	KMB willing to load MEKs in this state
0h	EMPTY	No
1h	ZEROIZED	No
2h	INVALID	No
3h	PROGRAMMED	Yes
4h	PERMANENT	Yes
5h to FFFFh	Reserved	

Table 46: Next action values

Value	Command
0h	PROGRAM_NEXT_SEK
1h	ZEROIZE_CURRENT_SEK
2h	PROGRAM_NEXT_HEK
3h	ZEROIZE_CURRENT_HEK
4h	ENABLE_PERMANENT_HEK
5h to FFFFh	Reserved

4.6.2.19 Fault handling

A KMB mailbox command can fail to complete in the following ways:

- An ill-formed command.
- Encryption engine timeout.
- Encryption engine reported error.

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

Depending on the type of fault, controller firmware may resubmit the mailbox command.

Each mailbox command that causes a command to execute on the encryption engine includes a timeout value is specified by the command. KMB 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.

Table 47 defines the additional mailbox result codes that may be returned by KMB.

Table 47: KMB mailbox command result codes

Name	Value	Description
LOCK_ENGINE_TIMEOUT	0x4C45_544F (“LETO”)	Timeout occurred when communicatingwith the drive crypto engine to execute a command
LOCK_ENGINE_CODE + u16	0x4443_xxxx (“ECxx”)	Vendor-specific error code in thelow 16bits
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
LOCK_ACCESS_KEY_UNWRAP	0x4C41_4B55 (“LAKU”)	Error during access key decryption
LOCK_MPK_DECRYPT	0x4C50_4445 (“LPDE”)	Error during MPK decryption
LOCK_MEK_DECRYPT	0x4C4D_4445 (“LMDE”)	Error during MEK decryption
LOCK_HEK_INVALID_SLOT	0x4C46_4953 (“LFIS”)	Incorrect HEK slot when programming or zeroizing

4.6.2.19.1 Fatal errors

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

4.6.2.19.2 Non-fatal errors

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

4.7 Terminology

The following acronyms and abbreviations are used throughout this document.

Abbreviation	Description

(continued on next page)

(continued from previous page)

Abbreviation	Description
AES	Advanced Encryption Standard
CSP	Cloud Service Provider
DPK	Data Protection Key
DICE	Device Identifier Composition Engine
DRBG	Deterministic Random Bit Generator
ECDH	Elliptic-curve Diffie–Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
HEK	Hard Epoch Key
HKDF	HMAC-based key derivation function
HMAC	Hash-Based Message Authentication Code
HPKE	Hybrid Public Key Encryption
IETF EAT	IETF Entity Attestation Token
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
MPK	Multi-party Protection Key
NIST	National Institute of Standards and Technology
OCP	Open Compute Project
RTL	Register Transfer Level
SED	Self-encrypting drive
SIK	Stable Identity Key
SEK	Soft Epoch Key
SSD	Solid-state drive
UART	Universal asynchronous receiver-transmitter
XTS	XEX-based tweaked-codebook mode with ciphertext stealing

4.8 Compliance

Item	Requirement	Mandatory

(continued on next page)

(continued from previous page)

Item	Requirement	Mandatory
1	The device shall integrate Caliptra.	Yes
2	OCP L.O.C.K. shall be enabled.	Yes
3	Media encryption keys shall only be programmable to the encryption engine via Caliptra.	Yes

4.9 Repository location

See https://github.com/chipsalliance/Caliptra/tree/main/doc/ocp_lock.

Appendix A: EAT format for attesting to the epoch key state

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

Appendix B: Sequence diagrams

B.1 Sequence of events at boot

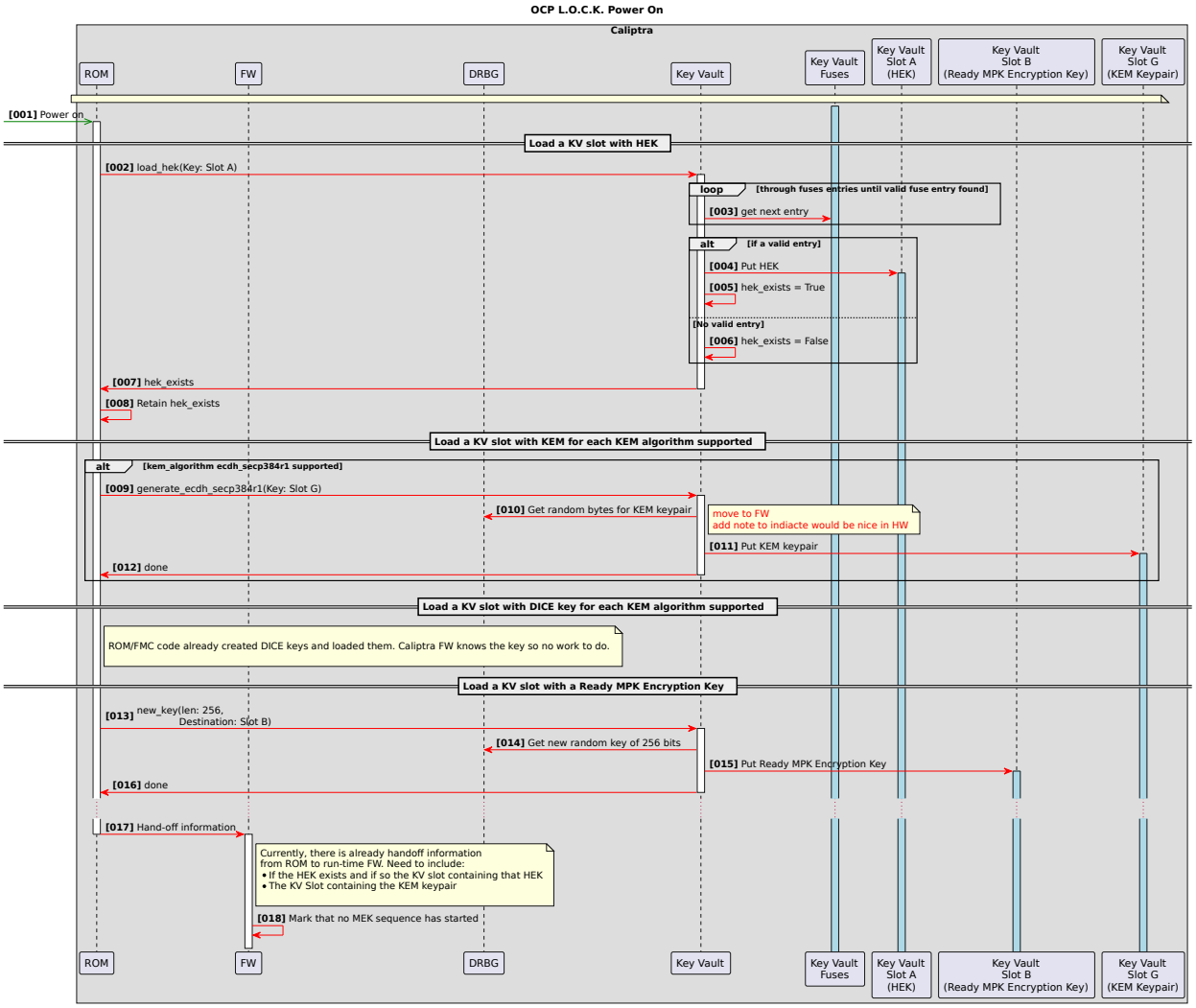

Figure 22: UML: Power on

B.2 Sequence to obtain the current status of KMB

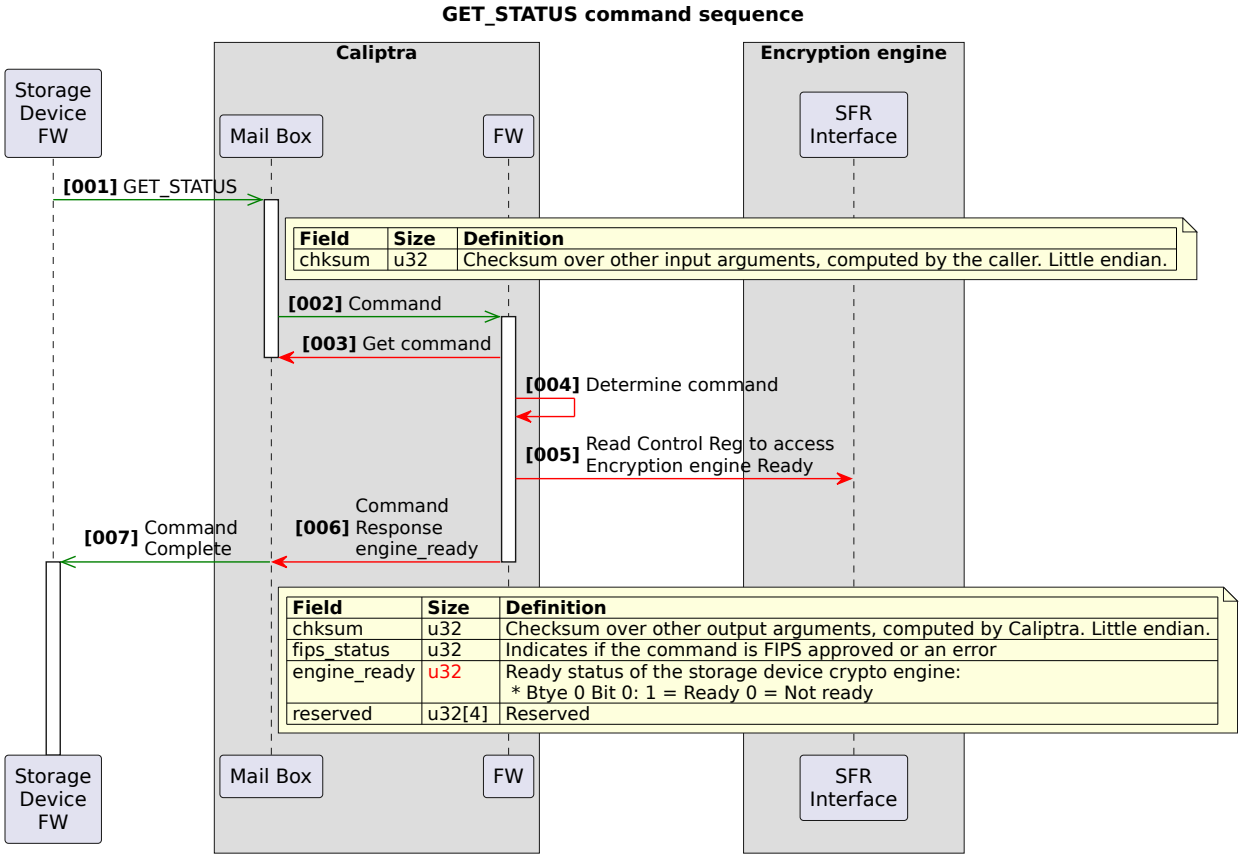

Figure 23: UML: Get Status

B.3 Sequence to obtain the supported algorithms

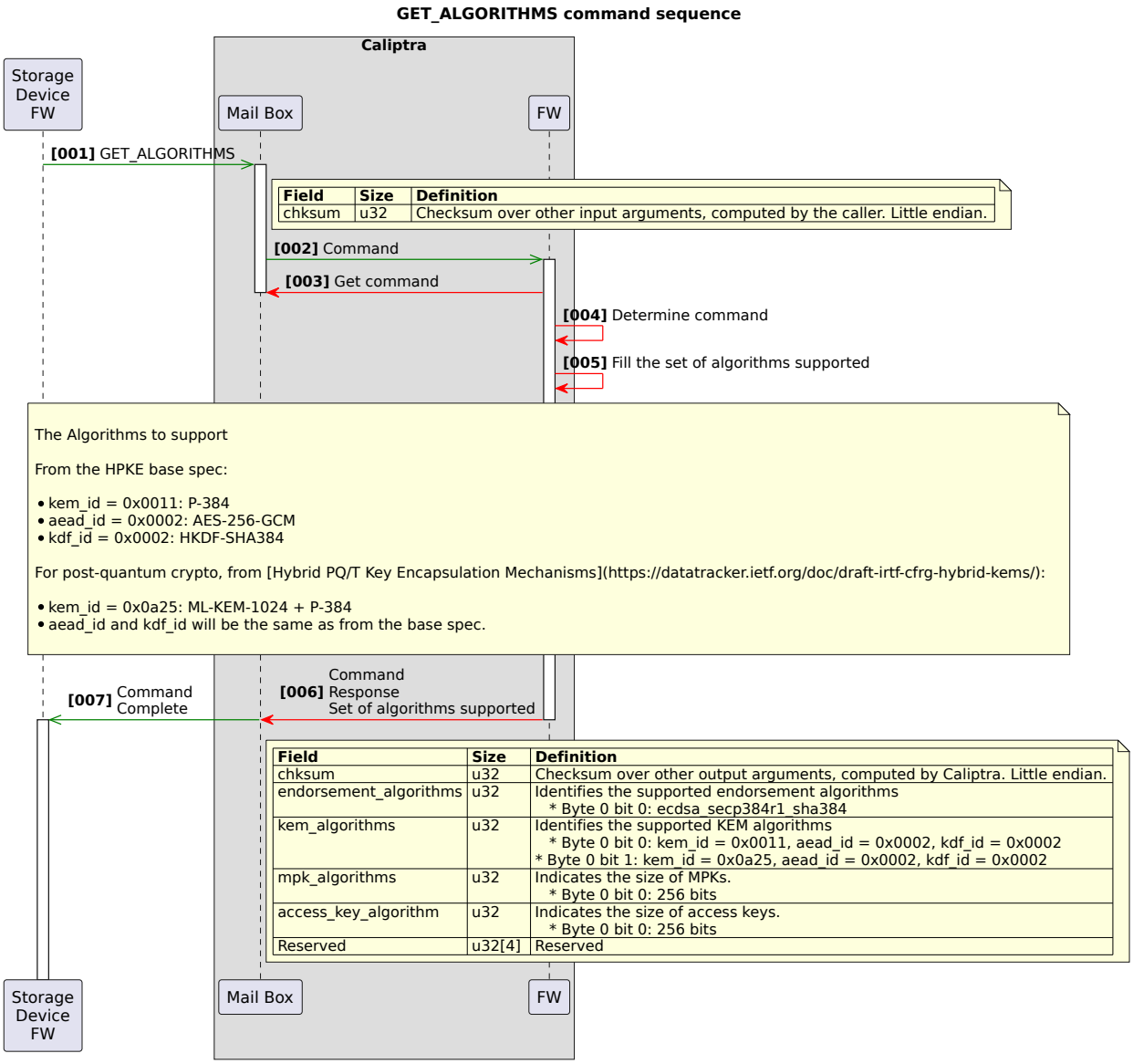

Figure 24: UML: Get Supported Algorithms

B.4 Sequence to endorse an HPKE public key

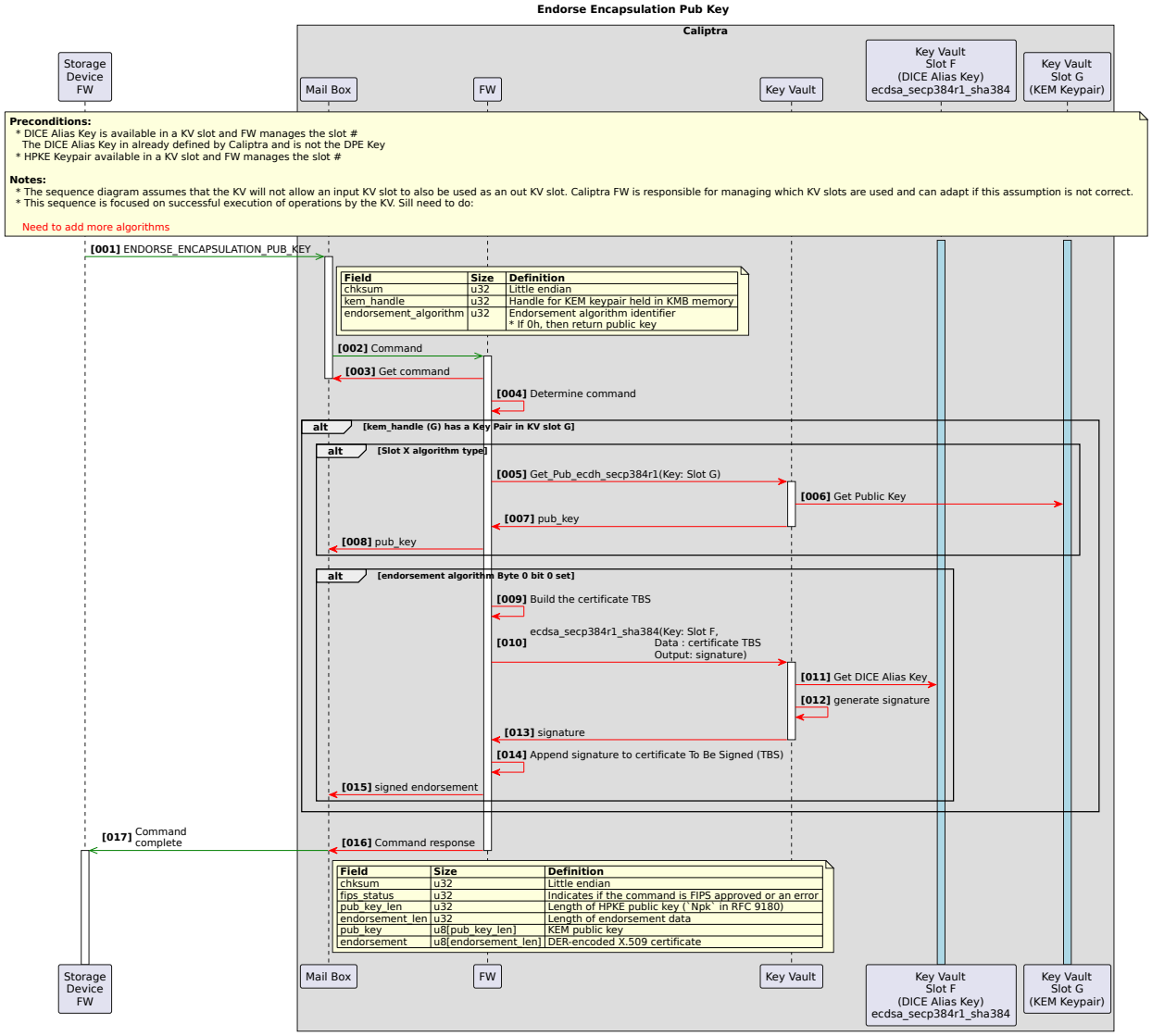

Figure 25: UML: Endorsing an HPKE public key

B.5 Sequence to rotate an HPKE keypair

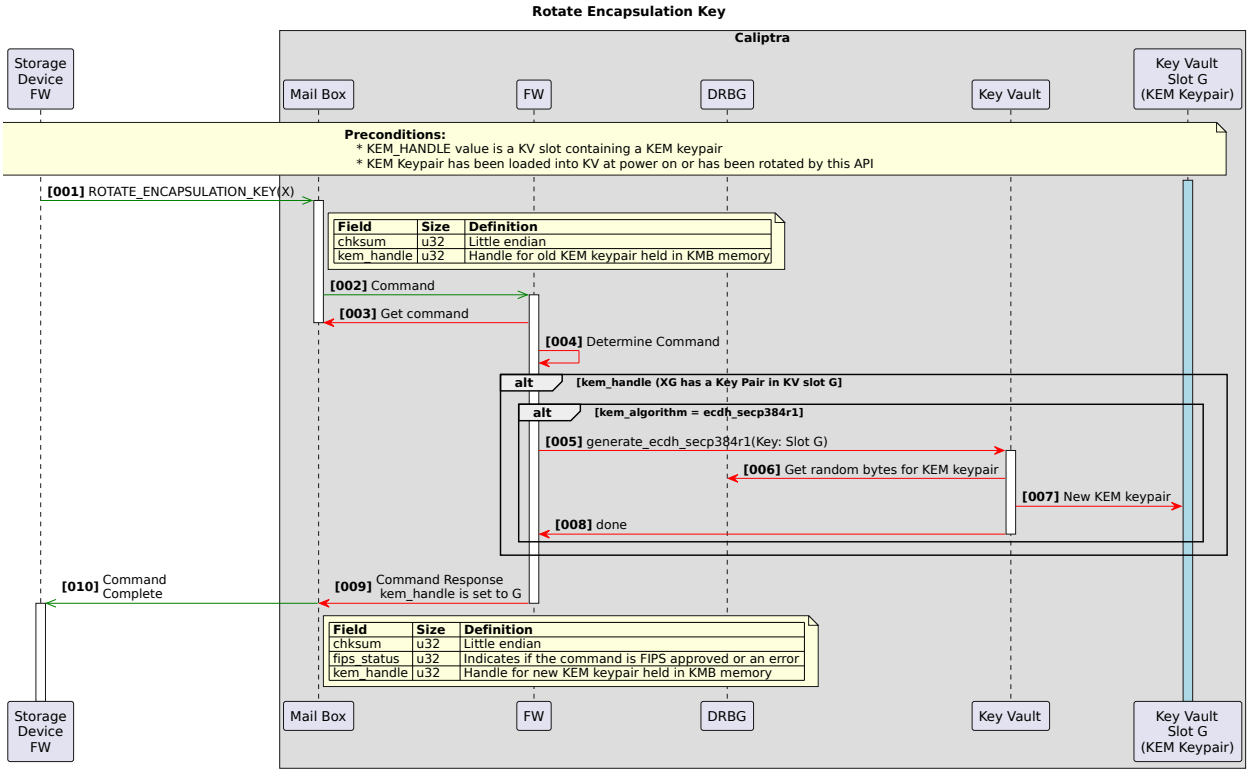

Figure 26: UML: Rotating a KEM Encapsulation Key

B.6 Sequence to generate a MPK

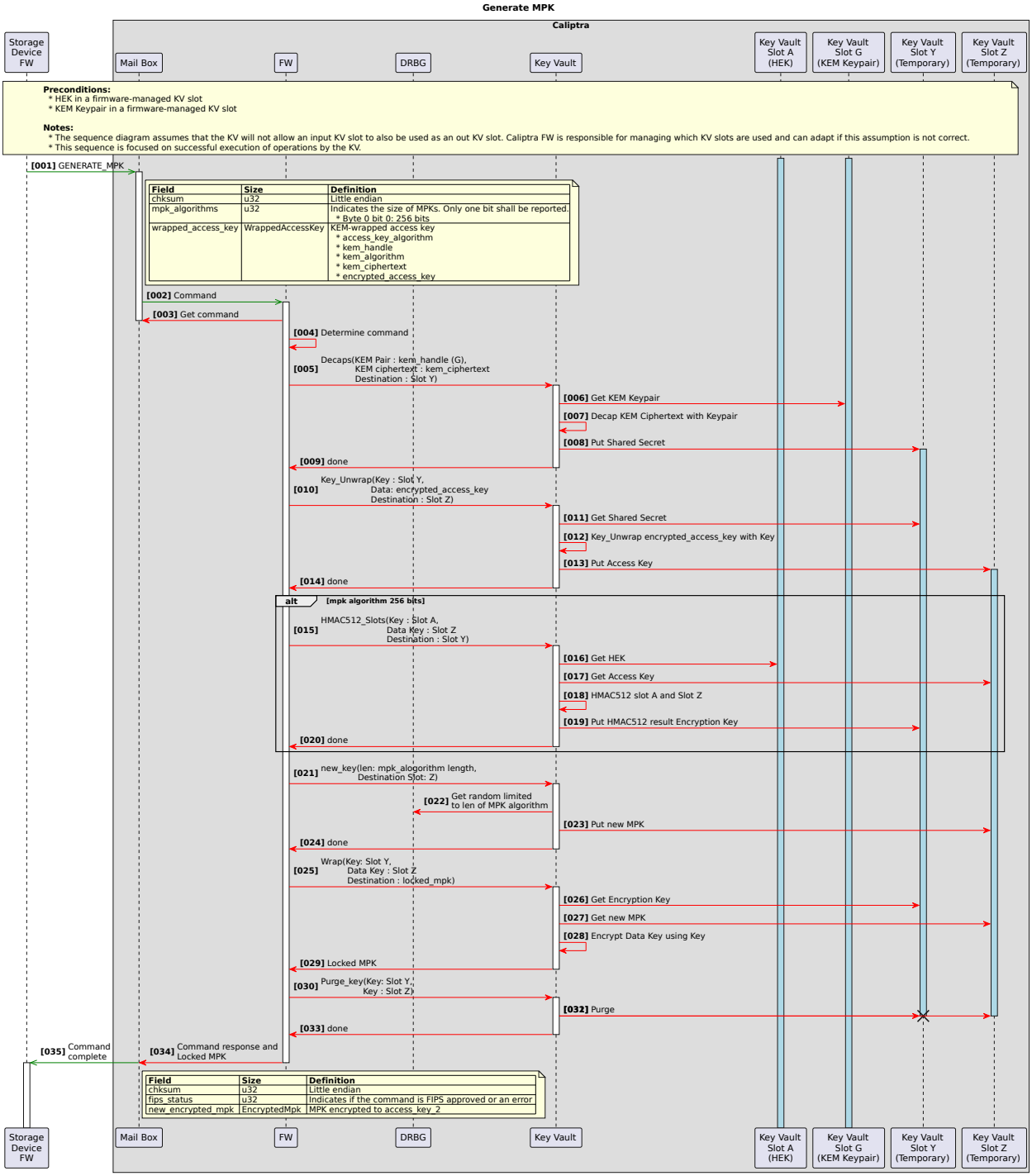

Figure 27: UML: Generating a MPK

B.7 Sequence to ready a MPK

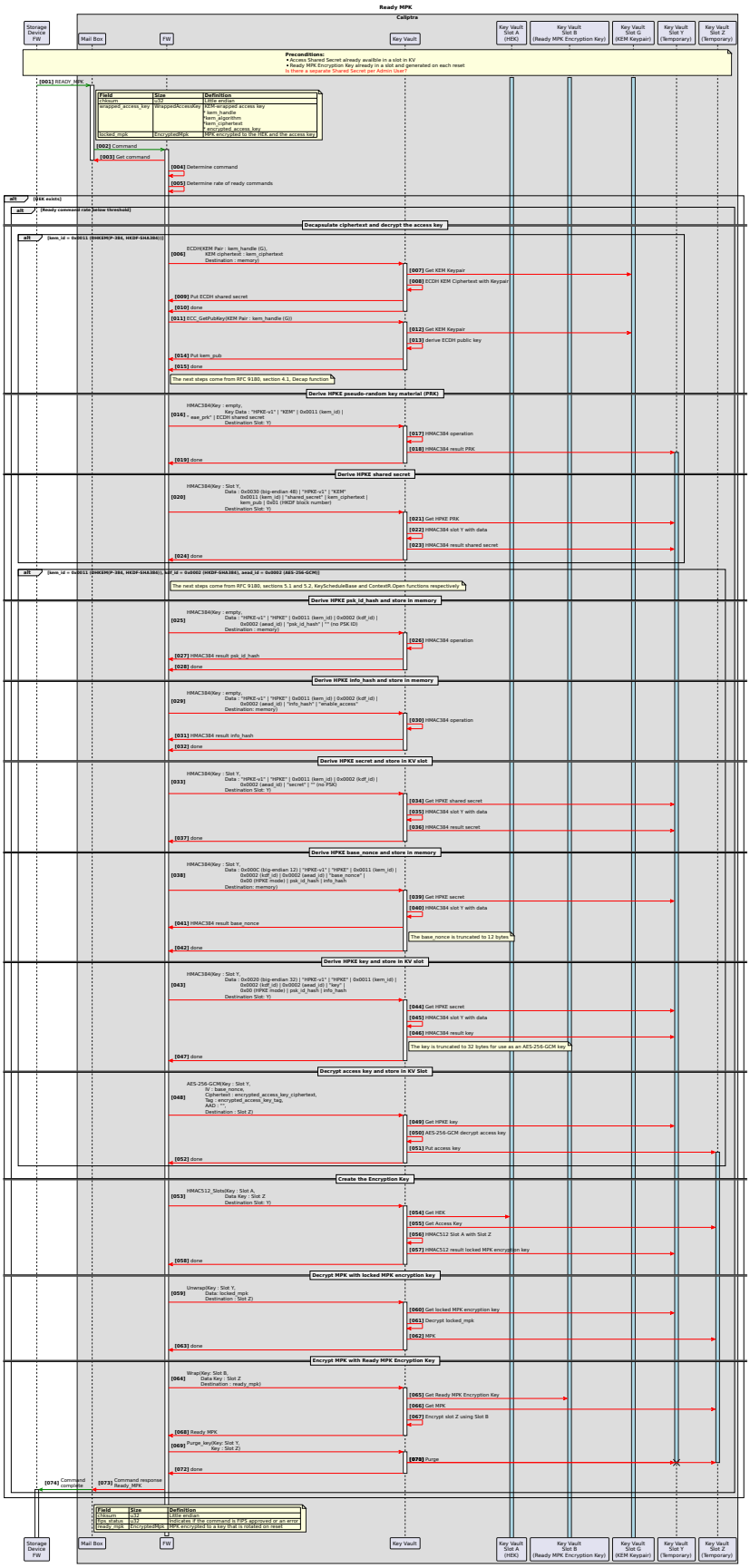

Figure 28: UML: Readying a MPK

B.8 Sequence to rotate the access key of a MPK

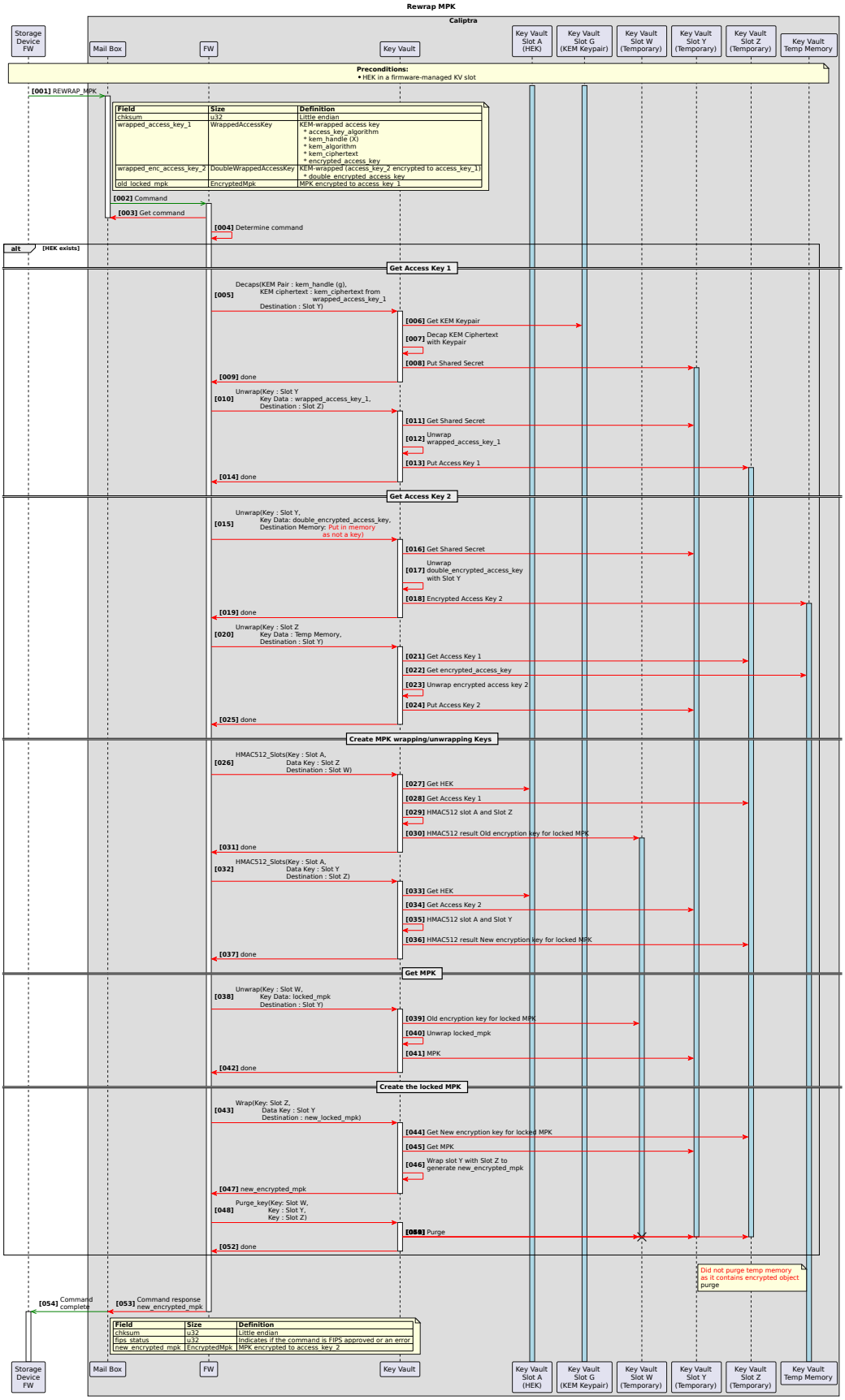

Figure 29: UML: Rewrapping a MPK

B.9 Sequence to mix a MPK into the MEK secret seed

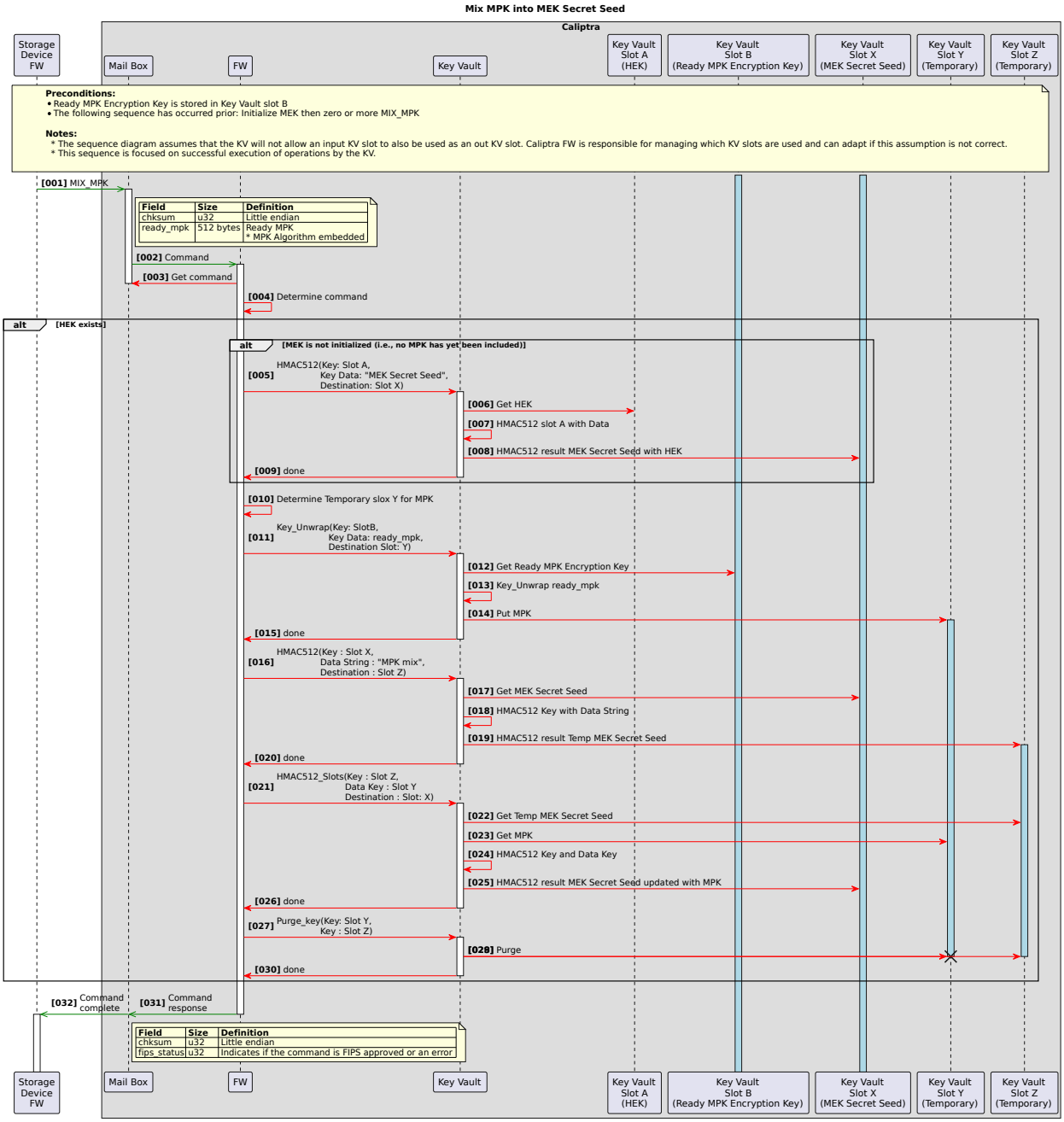

Figure 30: UML: Mixing a MPK

B.10 Sequence to load an MEK

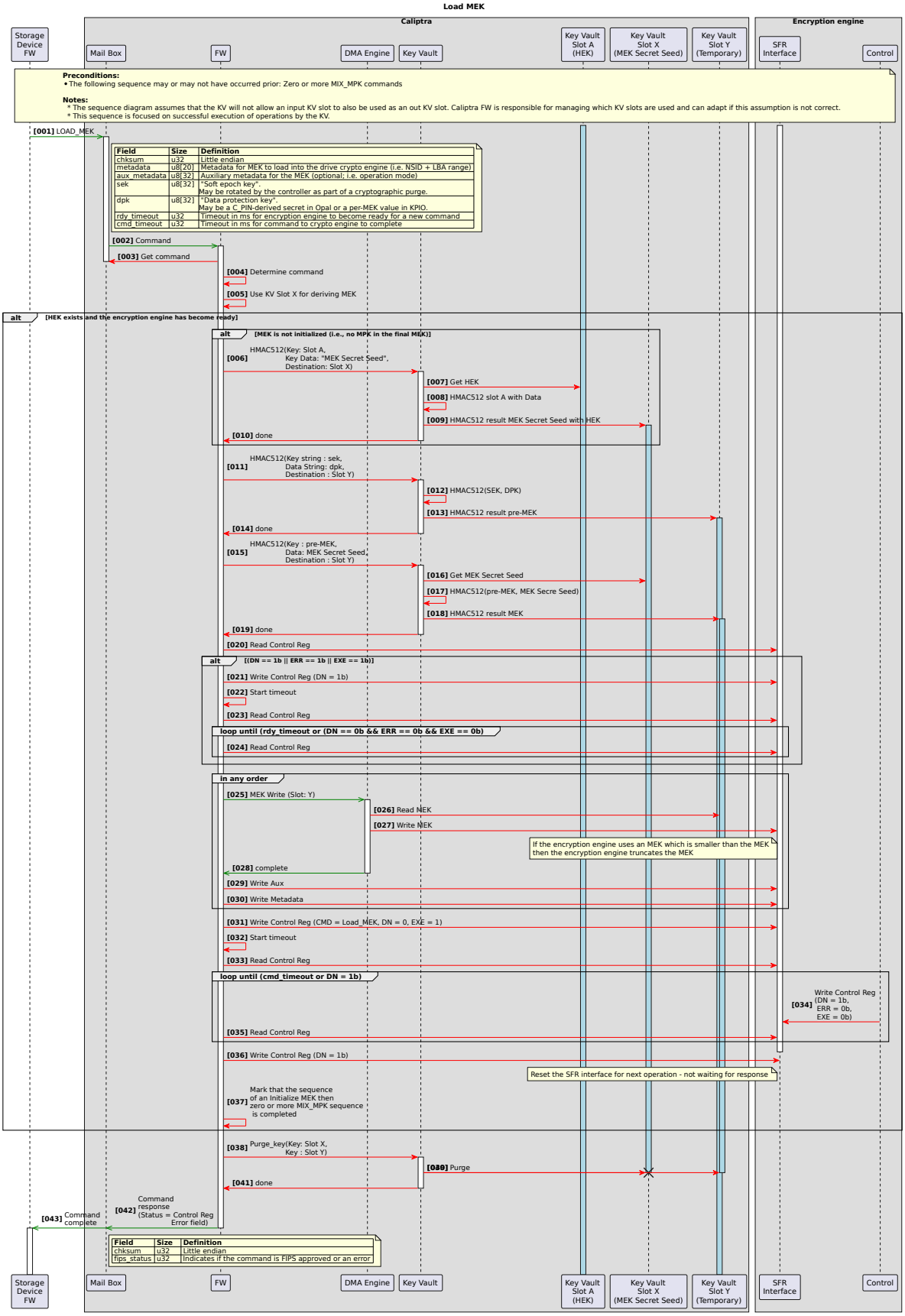

Figure 31: UML: Loading an MEK

B.11 Sequence to load MEK into the encryption engine key cache

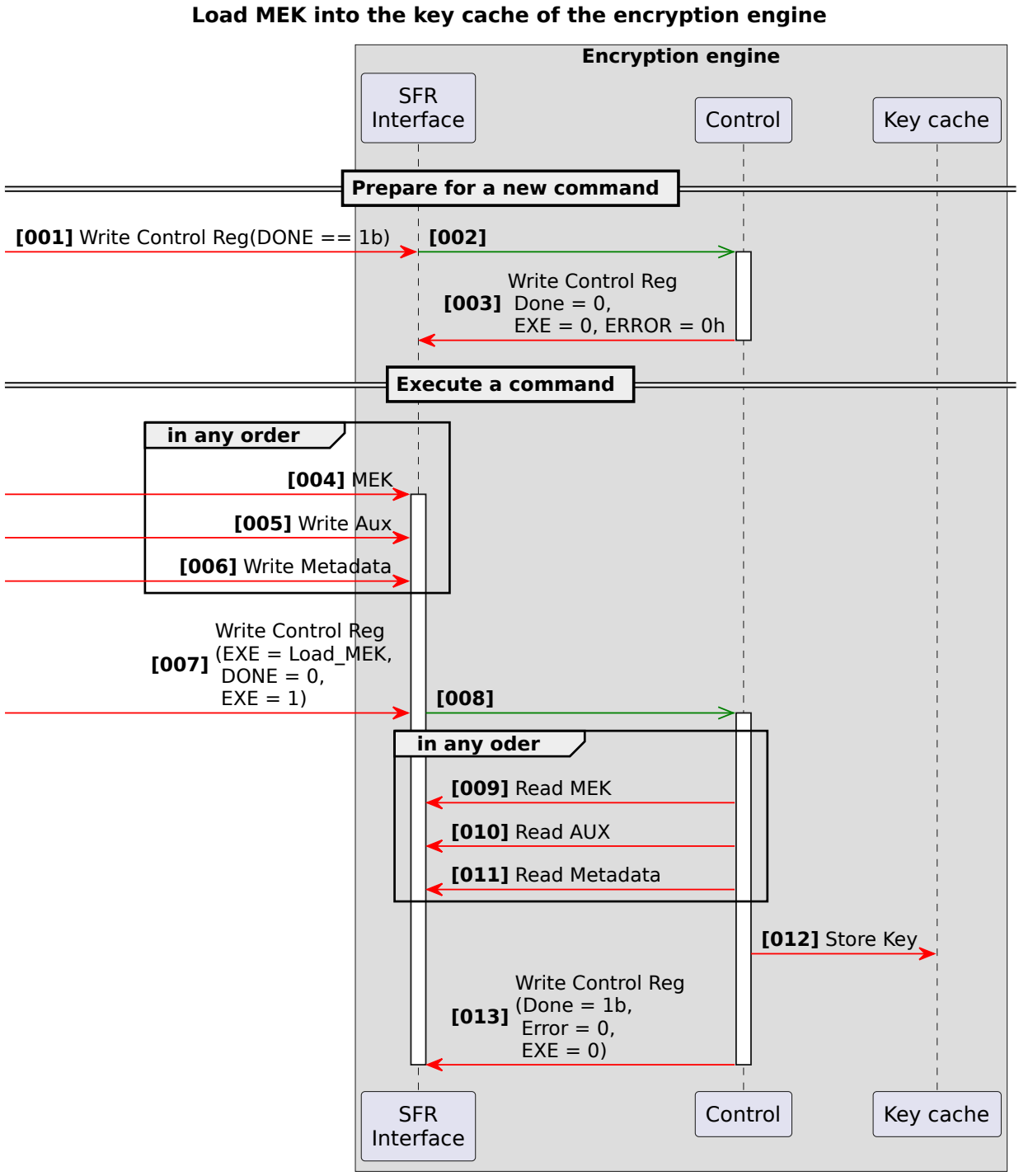

Figure 32: UML: Loading an MEK into the encryption engine key cache

B.12 Sequence to unload an MEK from the encryption engine key cache

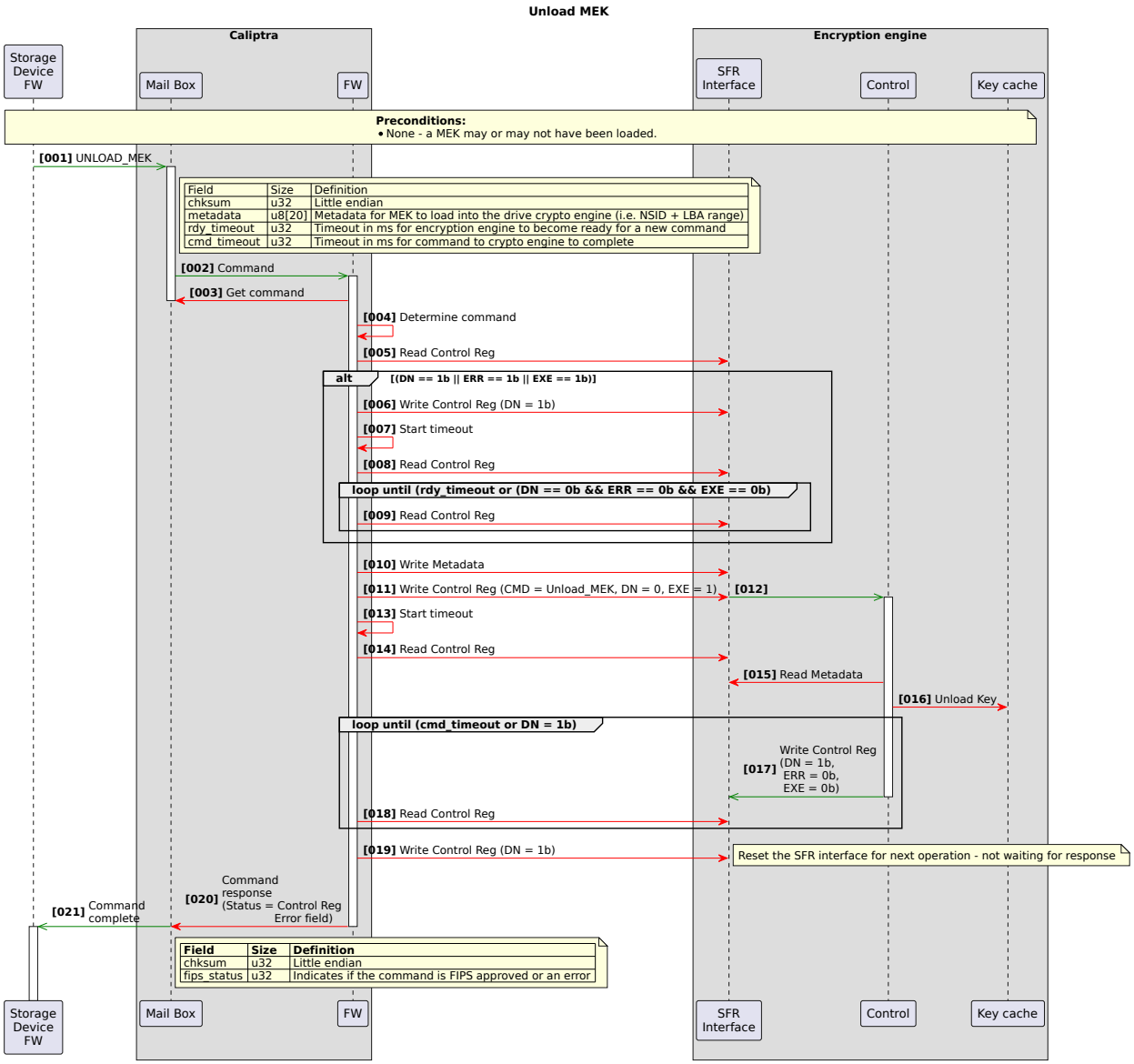

Figure 33: UML: Unloading an MEK

B.13 Sequence to unload all MEKs (i.e., purge) from the encryption engine key cache

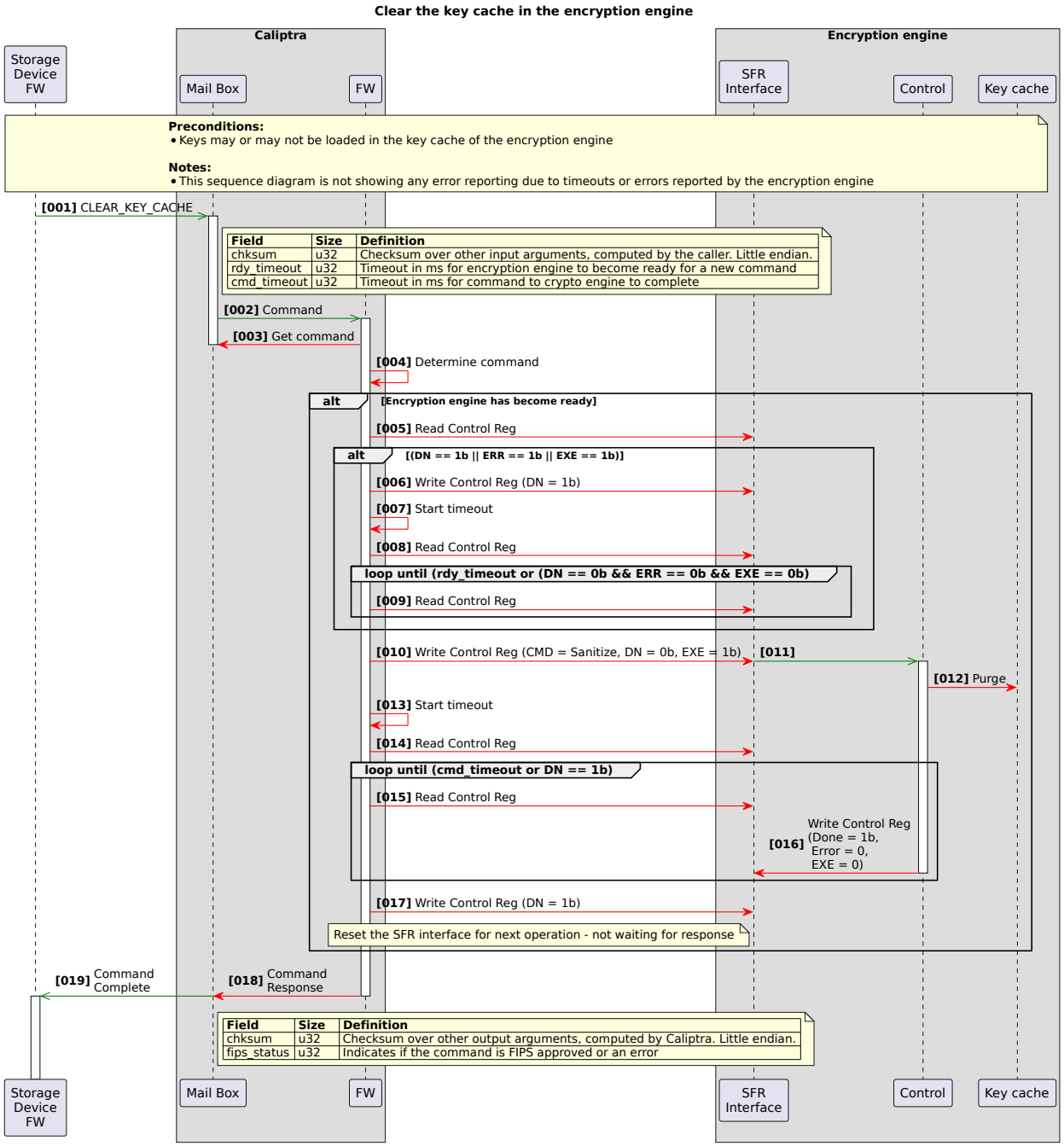

Figure 34: UML: Unloading all MEKs

References

[1] C. Meijer and B. van Gastel, “Self-encrypting deception: weaknesses in the encryption of solid state drives.” Available: https://www.cs.ru.nl/~cmeijer/publications/Self_Encrypting_Deception_Weaknesses_in_the_Encryption_of_Solid_State_Drives.pdf

[2] “TCG Storage Security Subsystem Class: Opal Specification.” Trusted Computing Group. Available: https://trustedcomputinggroup.org/resource/storage-work-group-storage-security-subsystem-class-opal/

[3] “TCG Storage Security Subsystem Class (SSC): Key Per I/O.” Trusted Computing Group. Available: https://trustedcomputinggroup.org/resource/tcg-storage-security-subsystem-class-ssc-key-per-i-o/

[4] “collaborative Protection Profile for Full Drive Encryption - Encryption Engine.” Common Criteria, Sep. 2016. Available: https://www.commoncriteriaportal.org/files/ppfiles/CPP_FDE_EE_V2.0.pdf

[5] “Hybrid Public Key Encryption.” IETF, Feb. 2022. Available: https://datatracker.ietf.org/doc/html/rfc9180

[6] “Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program.” NIST, Apr. 2025. Available: https://csrc.nist.gov/csrc/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf

[7] “The Transport Layer Security (TLS) Protocol Version 1.3.” IETF, Aug. 2018. Available: https://datatracker.ietf.org/doc/html/rfc8446

[8] J. Massimo, P. Kampanakis, S. Turner, and B. E. Westerbaan, “Internet X.509 Public Key Infrastructure: Algorithm Identifiers for ML-DSA.” Feb. 2025. Available: https://datatracker.ietf.org/doc/html/draft-ietf-lamps-dilithium-certificates

[9] D. Connolly, “Hybrid PQ/T Key Encapsulation Mechanisms.” Feb. 2025. Available: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hybrid-kems