OPEN

Compute Project

NVMe™ Project

OCP L.O.C.K.

0.8.5

Modular Base Specification
Effective June 12, 2025

Author: (See Acknowledgements section)

Current Template Version:

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

Table of Contents

1 License **8**
1.1 Open Web Foundation (OWF) CLA 8
2 Acknowledgements **9**
3 Compliance with OCP Tenets **10**
3.1 Openness 10
3.2 Efficiency 10
3.3 Impact 10
3.4 Scale 10
3.5 Sustainability 10
4 Base specification **11**
4.1 Introduction 11
4.2 Background 11
4.3 Threat model 11
4.4 OCP L.O.C.K. goals 12
4.4.1 Non-goals 12
4.4.2 Integration verification 13
4.5 Architecture 14
4.5.1 Host APIs 15
4.5.2 Key hierarchy 16
4.5.3 MPKs 19
4.5.4 DPKs 27
4.5.5 HEKs and SEKs 28
4.5.6 MEKs 33
4.5.7 Random key generation via DRBG 36
4.5.8 Boot-time initialization 37
4.5.9 Authorization 37
4.6 Interfaces 38
4.6.1 Encryption engine interface 38
4.6.2 Mailbox interface 48
4.7 Terminology 67
4.8 Compliance 68
4.9 Repository location 69
Appendices **70**
A Preconditioned AES-Encrypt calculations **70**
B EAT format for attesting to the epoch key state **71**
C Sequence diagrams **72**
C.1 Sequence of events at boot 72
C.2 Sequence to obtain the current status of KMB 73
C.3 Sequence to obtain the supported algorithms 74
C.4 Sequence to endorse an HPKE public key 75
C.5 Sequence to rotate an HPKE keypair 76

C.6 Sequence to generate an MPK 77
C.7 Sequence to ready an MPK 79
C.8 Sequence to rotate the access key of an MPK 81
C.9 Sequence to mix an MPK into the MEK secret seed 82
C.10 Sequence to test the access key of an MPK 83
C.11 Sequence to load an MEK 85
C.12 Sequence to load MEK into the encryption engine key cache 86
C.13 Sequence to unload an MEK from the encryption engine key cache 87
C.14 Sequence to unload all MEKs (i.e., zeroize) from the encryption engine key cache ... 88
References **89**

List of Figures

1 OCP L.O.C.K. high level blocks 14
2 OCP L.O.C.K. key hierarchy 16
3 Preconditioned HKDF-Extract 17
4 Preconditioned AES-Encrypt 18
5 Preconditioned AES-Decrypt 18
6 HPKE unwrap for MPK access keys with ECDH 21
7 HPKE unwrap for MPK access keys with hybrid ML-KEM + ECDH 22
8 MPK generation 24
9 MPK readying 25
10 MPK access key rotation 26
11 MPK access key testing 27
12 Example drive flow to decrypt a DPK based on a host-provided Opal C_PIN 28
13 Example drive flow to accept an injected DPK 28
14 HEK derivation 30
15 HEK and SEK state machine 32
16 MEK secret derivation 34
17 Generating an MEK 35
18 Loading an MEK 35
19 Deriving an MEK 36
20 DRBG usage 37
21 KMB to encryption engine SFR interface 39
22 LBA range based metadata format 44
23 Key tag based metadata format 45
24 Auxiliary data format example 46
25 MEK format example for AES-XTS-256 47
26 Command execution example for loading an MEK 48
27 UML: Power on 72
28 UML: Get Status 73
29 UML: Get Supported Algorithms 74
30 UML: Endorsing an HPKE public key 75
31 UML: Rotating a KEM Encapsulation Key 76
32 UML: Generating an MPK 77
33 UML: Readying an MPK 79
34 UML: Rewrapping an MPK 81
35 UML: Mixing an MPK 82
36 UML: Testing an Access Key 83
37 UML: Loading an MEK 85
38 UML: Loading an MEK into the encryption engine key cache 86
39 UML: Unloading an MEK 87
40 UML: Unloading all MEKs 88

List of Tables

1 HEK lifecycle states 29
2 HEK lifecycle management commands 30
3 KMB to encryption engine SFRs 40
4 Offset SFR_Base + 0h: CTRL – Control 40
5 CTRL error codes 41
6 CTRL command codes 41
7 Offset SFR_Base + 10h: METD – Metadata 43
8 Offset SFR_Base + 20h: AUX – Auxiliary Data 45
9 Offset SFR_Base + 40h: MEK – Media Encryption Key 46
10 Values for the FIPS status field 48
11 GET_STATUS input arguments 49
12 GET_STATUS output arguments 49
13 GET_ALGORITHMS input arguments 49
14 GET_ALGORITHMS output arguments 50
15 CLEAR_KEY_CACHE input arguments 50
16 CLEAR_KEY_CACHE output arguments 50
17 ENDORSE_ENCAPSULATION_PUB_KEY input arguments 51
18 ENDORSE_ENCAPSULATION_PUB_KEY output arguments 51
19 ROTATE_ENCAPSULATION_KEY input arguments 52
20 ROTATE_ENCAPSULATION_KEY output arguments 52
21 GENERATE_MPK input arguments 52
22 GENERATE_MPK output arguments 53
23 REWRAP_MPK input arguments 53
24 REWRAP_MPK output arguments 54
25 READY_MPK input arguments 54
26 READY_MPK output arguments 54
27 MIX_MPK input arguments 55
28 MIX_MPK output arguments 55
29 TEST_ACCESS_KEY input arguments 55
30 TEST_ACCESS_KEY output arguments 56
31 GENERATE_MEK input arguments 56
32 GENERATE_MEK output arguments 57
33 LOAD_MEK input arguments 57
34 LOAD_MEK output arguments 58
35 DERIVE_MEK input arguments 58
36 DERIVE_MEK output arguments 58
37 UNLOAD_MEK input arguments 59
38 UNLOAD_MEK output arguments 59
39 ENUMERATE_KEM_HANDLES input arguments 59
40 ENUMERATE_KEM_HANDLES output arguments 60
41 KemHandle contents 60
42 ZEROIZE_CURRENT_HEK input arguments 60
43 ZEROIZE_CURRENT_HEK output arguments 60
44 PROGRAM_NEXT_HEK input arguments 61
45 PROGRAM_NEXT_HEK output arguments 61
46 ENABLE_PERMANENT_HEK input arguments 61

47 ENABLE_PERMANENT_HEK output arguments 62
48 REPORT_EPOCH_KEY_STATE input arguments 62
49 REPORT_EPOCH_KEY_STATE output arguments 62
50 SEK state values 63
51 HEK state values 63
52 SealedAccessKey contents 64
53 SealedCurrentAndNewAccessKey contents 64
54 WrappedKey contents 65
55 WrappedKey variants 65
56 KMB mailbox command result codes 66
57 Acronyms and abbreviations used throughout this document 67
58 Compliance requirements 68

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.

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:

- Andrés Lagar-Cavilla (Google)
- Amber Huffman (Google)
- Charles Kunzman (Google)
- Chris Sabol (Google)
- Jeff Andersen (Google)
- Srini Narayanamurthy (Google)
- Zach Halvorsen (Google)
- Anjana Parthasarathy (Microsoft)
- B 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)
- Artem Zankovich (Micron)
- Bharath Madanayakanahalli Gururaj (Micron)
- Jef Munsil (Micron)
- Jimmy Ruane (Micron)
- Jonathan Chang (Micron)
- Rob Strong (Micron)

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 drive 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 storage products for Google and Microsoft. This scale covers both a significant portion of the hyperscale and enterprise markets.

3.5 Sustainability

The goal of OCP L.O.C.K. is to eliminate the need for cloud service providers (CSPs) to destroy storage devices (e.g., SSDs) by providing a mechanism that increases the confidence that a media encryption key within the device is deleted during cryptographic 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 market serving CSPs, 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 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., cryptographic erase), 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 drive 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 erased, to cryptographically erase all data which was encrypted to the MEK. To erase an MEK, it is sufficient for the drive to erase all copies of it, or all copies of 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 CSP which manages an attached storage device.
- Accessing all device design documents, code, and RTL.

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

- MEKs managed by storage drive firmware are compromised due to implementation bugs or side channels.
- MEKs erased by storage drive 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 erase 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 encryption engine. The KMB will expose services to drive firmware which will allow the drive 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 component.
- 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 erase.
- Is able to be used in conjunction with the Opal or Key Per I/O [3] storage device specifications.

Note that Opal in the context of this document includes the Opal Family of SSCs: Opal SSC, Opalite SSC, Ruby SSC. Pyrite is excluded as Pyrite SSDs are non-SED.

4.4.1 Non-goals

The following areas are out of scope for this project.

- Compliance with IEEE 1619^{TM}-2025 [4] requirements around key scope, i.e. restricting a given MEK to only encrypt a maximum of 2^{44} 128-bit blocks. Drive firmware will be responsible for enforcing this.
- Compliance with Common Criteria requirement FCS_CKM.1.1(c) [5] 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 erase, before the injected DEK is used

to encrypt user data, the DEK 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 erase 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 drive 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 drive firmware correctly invokes OCP L.O.C.K. services.

4.5 Architecture

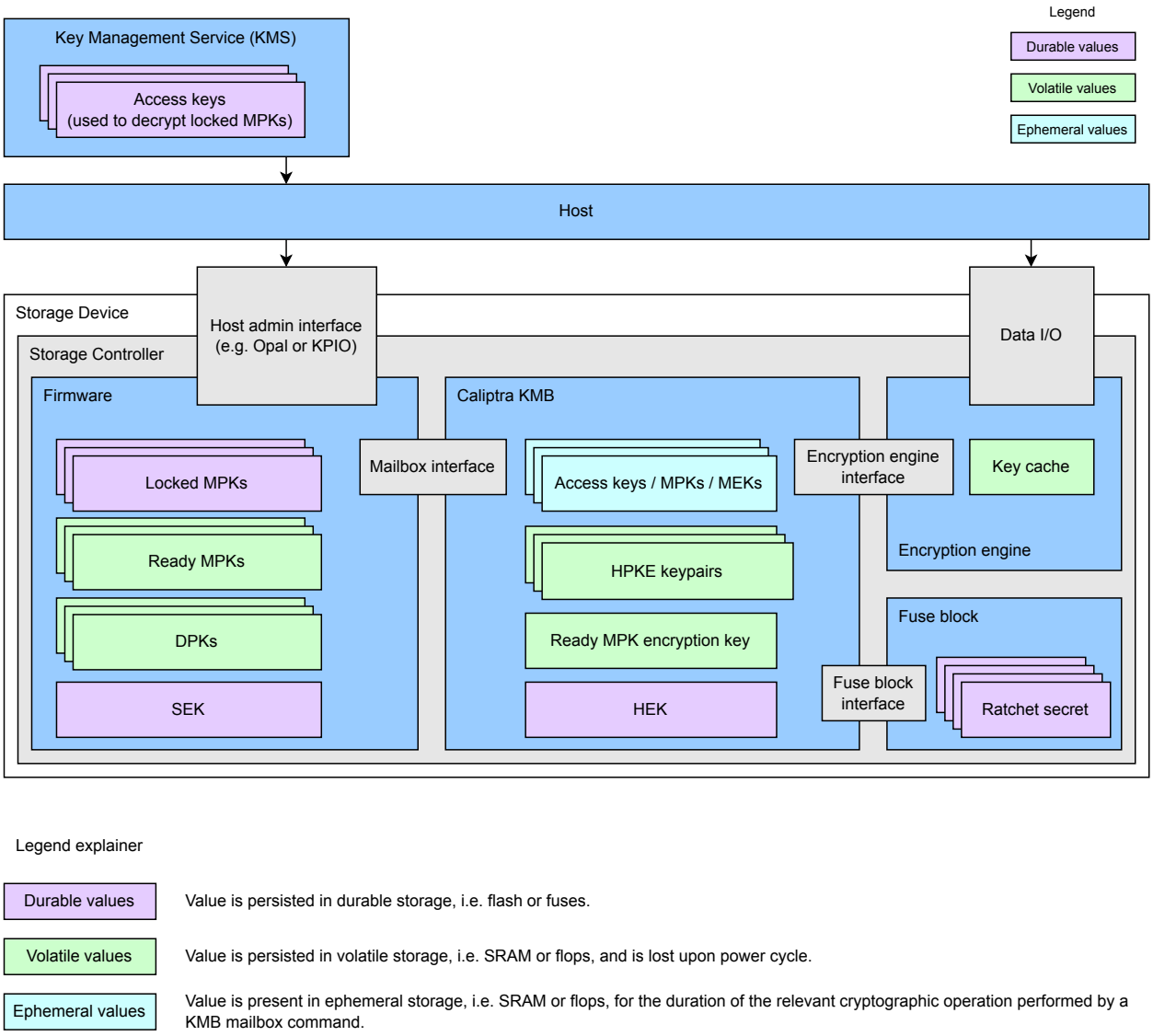

Figure 1: OCP L.O.C.K. high level blocks

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 [6]. 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 cryptographically erased 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 cryptographically erased state.
  - The SEK is managed by drive 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 drive 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 drive 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 drive firmware. Drive 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 upon request from drive firmware.

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

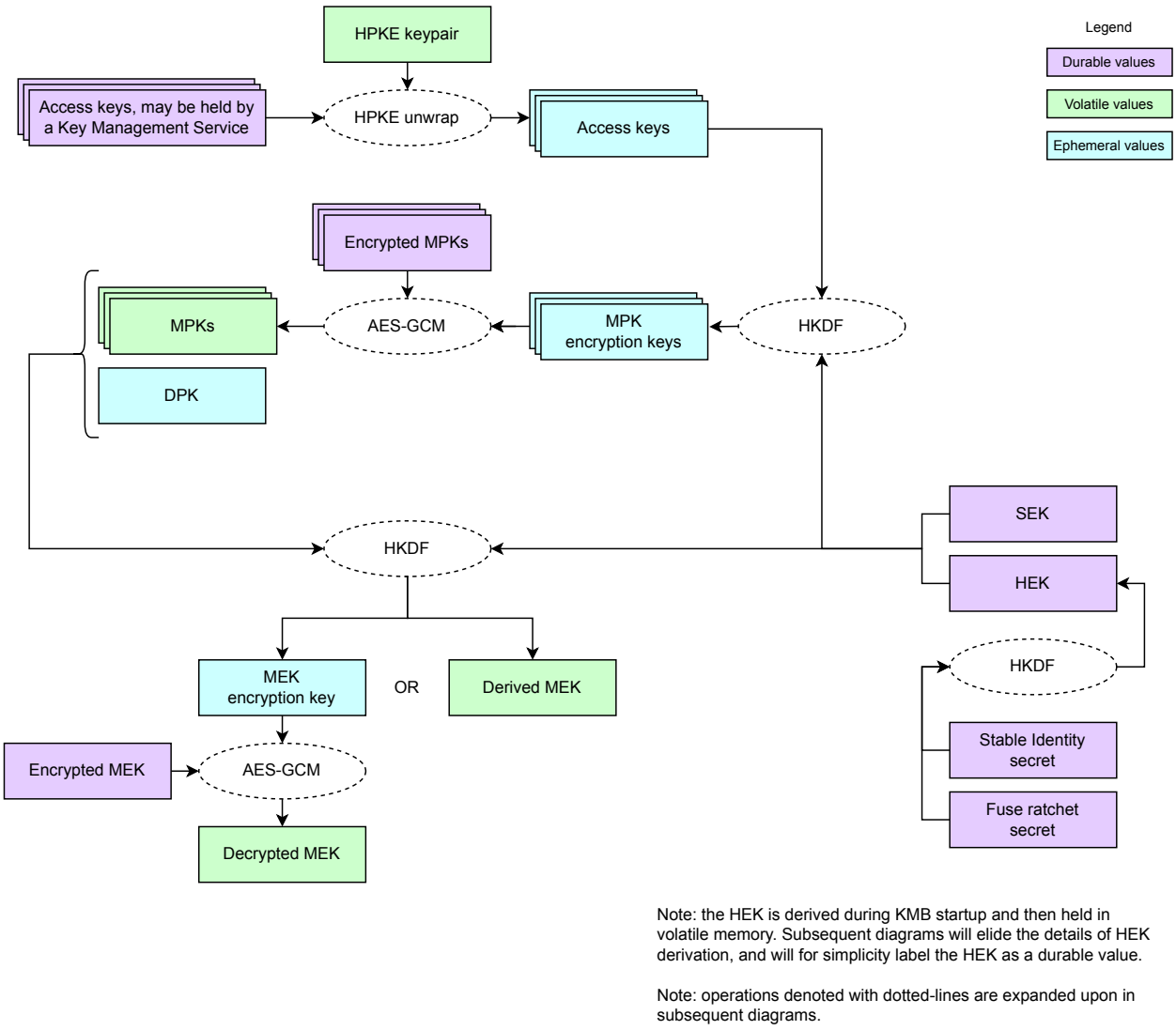

Figure 2: OCP L.O.C.K. key hierarchy

4.5.2.1 Preconditioned HKDF

In this specification, several flows involve combining together multiple symmetric keys. These keys have different scopes and lifespans. For example, the HEK and SEK are long-lived per-drive keys, while the DPK is unique to each MEK.

NIST SP 800-133 [7] section 6.3 provides a number of FIPS-approved methods for combining keys together. This specification leverages HKDF-Extract [8], which is equivalent to the key-extraction process detailed in SP 800-133 section 6.3. That section provides the following stipulation: “Each component key **shall** be kept secret and **shall not** be used for any purpose other than the computation of a specific symmetric key K (i.e., a given component key **shall not** be used to generate more than one key).”

A strict reading of this requirement would preclude, for example, combining the same SEK with different DPKs to produce different per-MEK output keys. Therefore, this specification defines the “preconditioned HKDF-Extract” operation, illustrated in Figure 3.

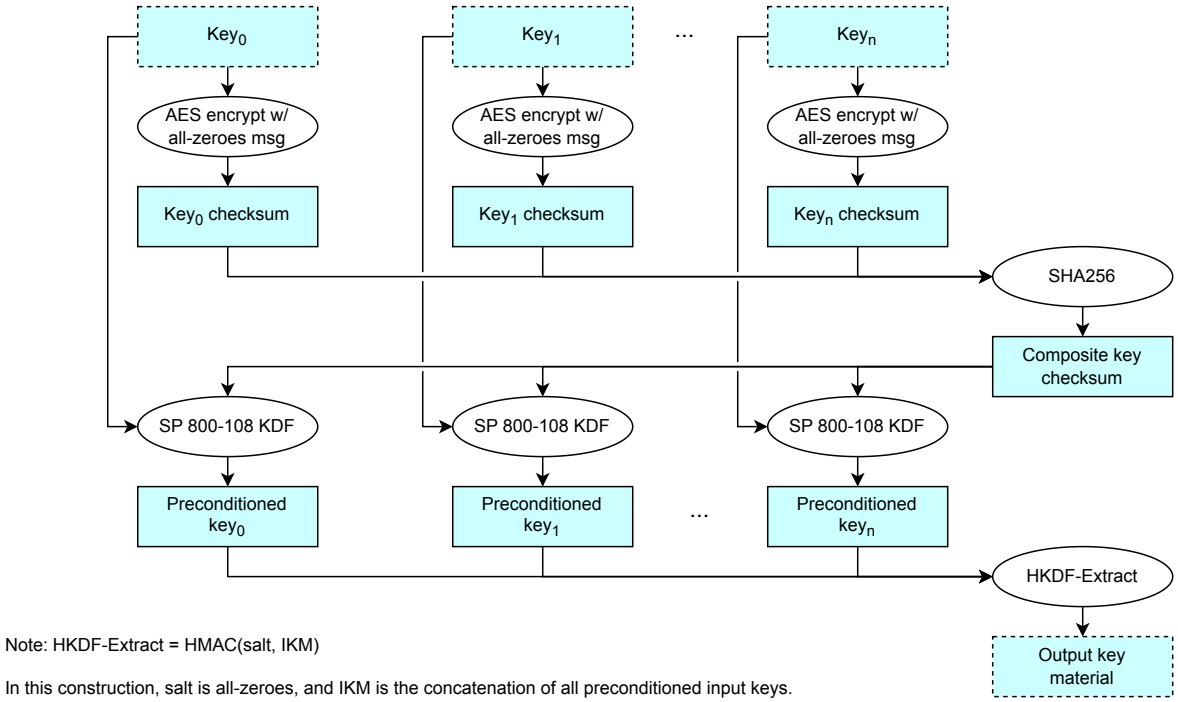

Figure 3: Preconditioned HKDF-Extract

Each input key is preconditioned using a unique identifier composed from each of the constituent keys, before being combined using HKDF-Extract. Therefore each key fed to HKDF-Extract is unique to the given key mixing operation and will not be re-used to produce different output keys.

Subsequent diagrams will use this operation when combining keys with different scopes or lifetimes.

4.5.2.2 Preconditioned AES

In this specification, several flows involve deriving a key for use in AES-GCM. In AES-GCM it is critical that IV collisions be avoided. To that end SP 800-38D [9] section 8.3 stipulates that AES-GCM-Encrypt may only be invoked at most 2^{32} times with a given AES key. One approach to address this constraint is to maintain counters to track the usage count of a given key. Such tracking would be difficult for Caliptra, which lacks direct access to rewritable storage which would be needed to maintain a counter that preserves its value across power cycles.

To elide the need for maintaining counters, this specification defines the “preconditioned AES-Encrypt” and “preconditioned AES-Decrypt” operations, illustrated in Figure 4 and Figure 5.

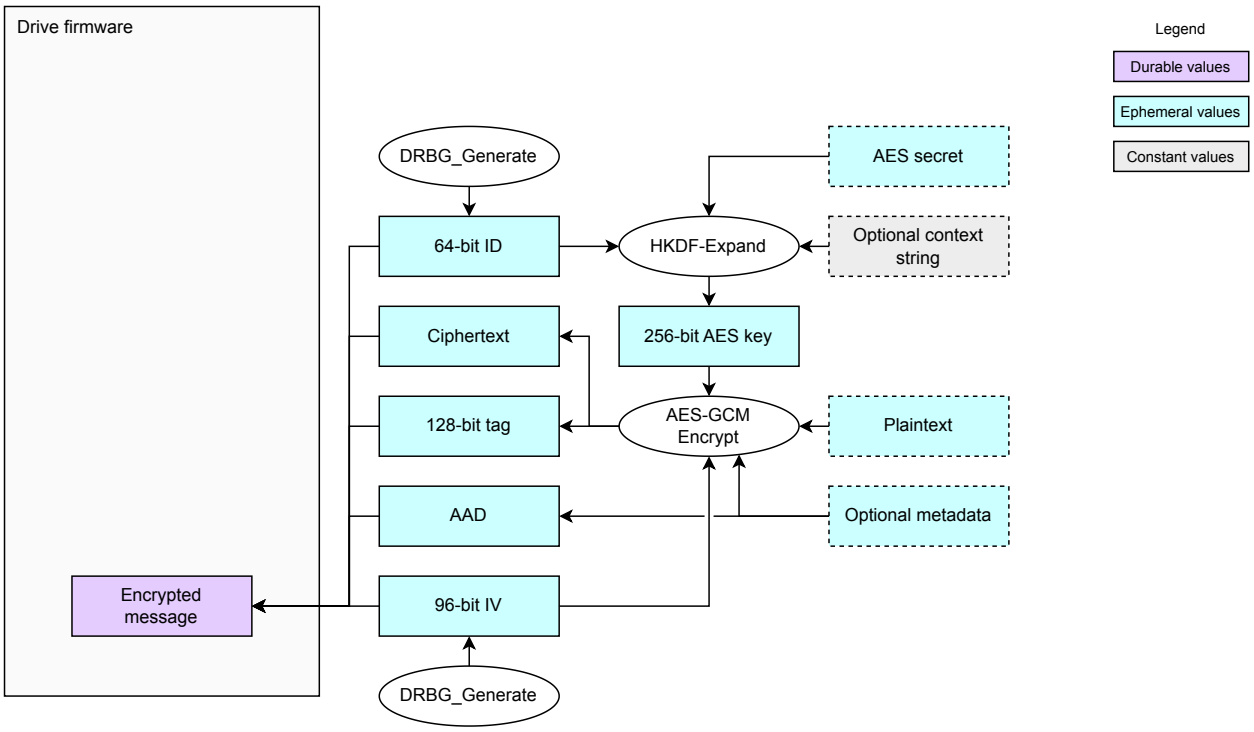

Figure 4: Preconditioned AES-Encrypt

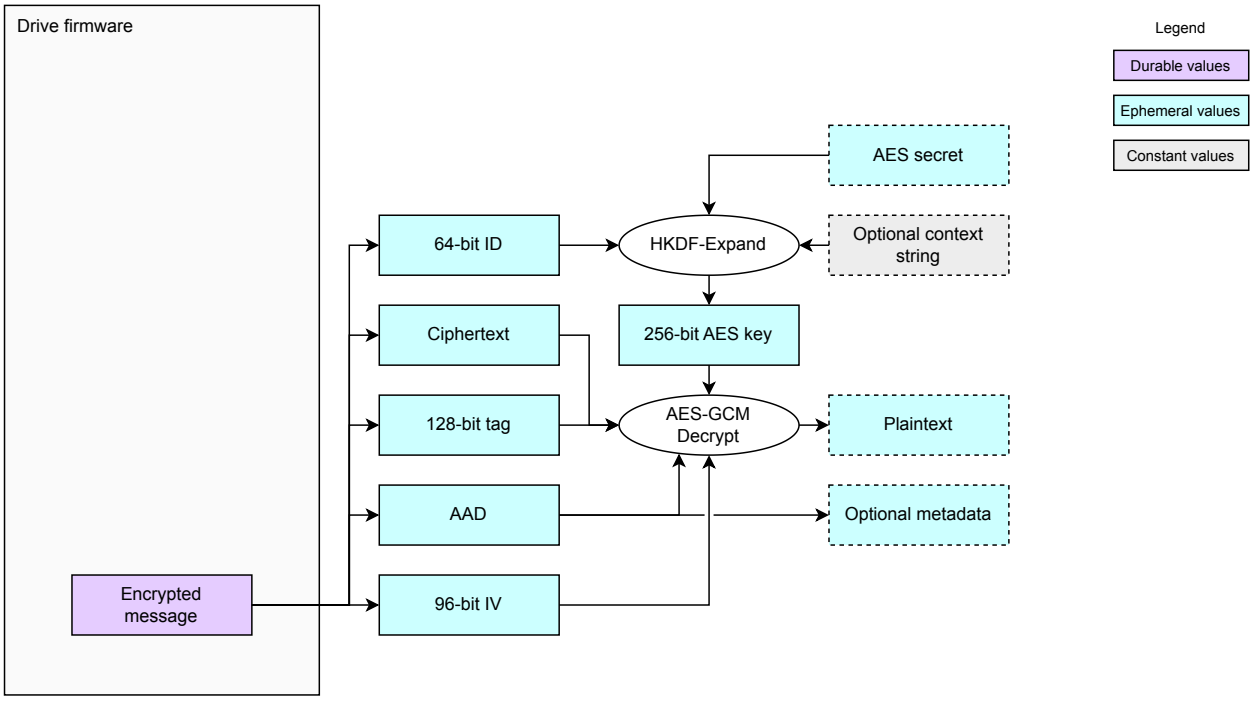

Figure 5: Preconditioned AES-Decrypt

Each AES secret is preconditioned using a randomly-generated 64-bit identifier, before being used with AES-GCM-Encrypt. This ensures that it is safe to use a given input AES secret in approximately 2^{64} preconditioned AES-Encrypt operations before an IV collision is expected to occur with probability greater than 2^{-32}. 2^{64} is large enough that a given storage device will experience hardware failure well before that limit is reached. Ergo, usage counters within the drive are not needed for these keys. See Appendix A for additional details on the calculations behind this figure.

In addition to a plaintext message, this operation supports optional metadata for the message, which is attached as AAD.

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, SEK, and an externally-supplied access key to which the MPK is bound. The locked MPK is held at rest by drive firmware.
- The Ready MPK Encryption Key is a volatile key held within KMB which is lazily generated and is lost when the drive shuts down. Lazy generation allows injected host entropy to contribute to the key's generation. Ready MPKs are held in drive firmware memory.

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 drive firmware passes that encrypted access key to KMB, along with the locked MPK, to produce the ready MPK which is cached in drive memory. This is done prior to the drive 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 [10] 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. Drive firmware is responsible for enumerating the available HPKE public keys and exposing them to the user via a host interface. The nature of the host interface is outside the scope of this specification.

When a user wishes to generate or ready an 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. Decode the HPKE capabilities of the storage device based on the HPKE certificate.
4. Seal their access key using the HPKE public key.
5. Transmit the sealed 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, defined in RFC 9180 [6]
- Hybridized ML-KEM-1024 with P-384 ECDH, defined in Hybrid PQ/T Key Encapsulation Mechanisms [11]

Figure 6 and Figure 7 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”.

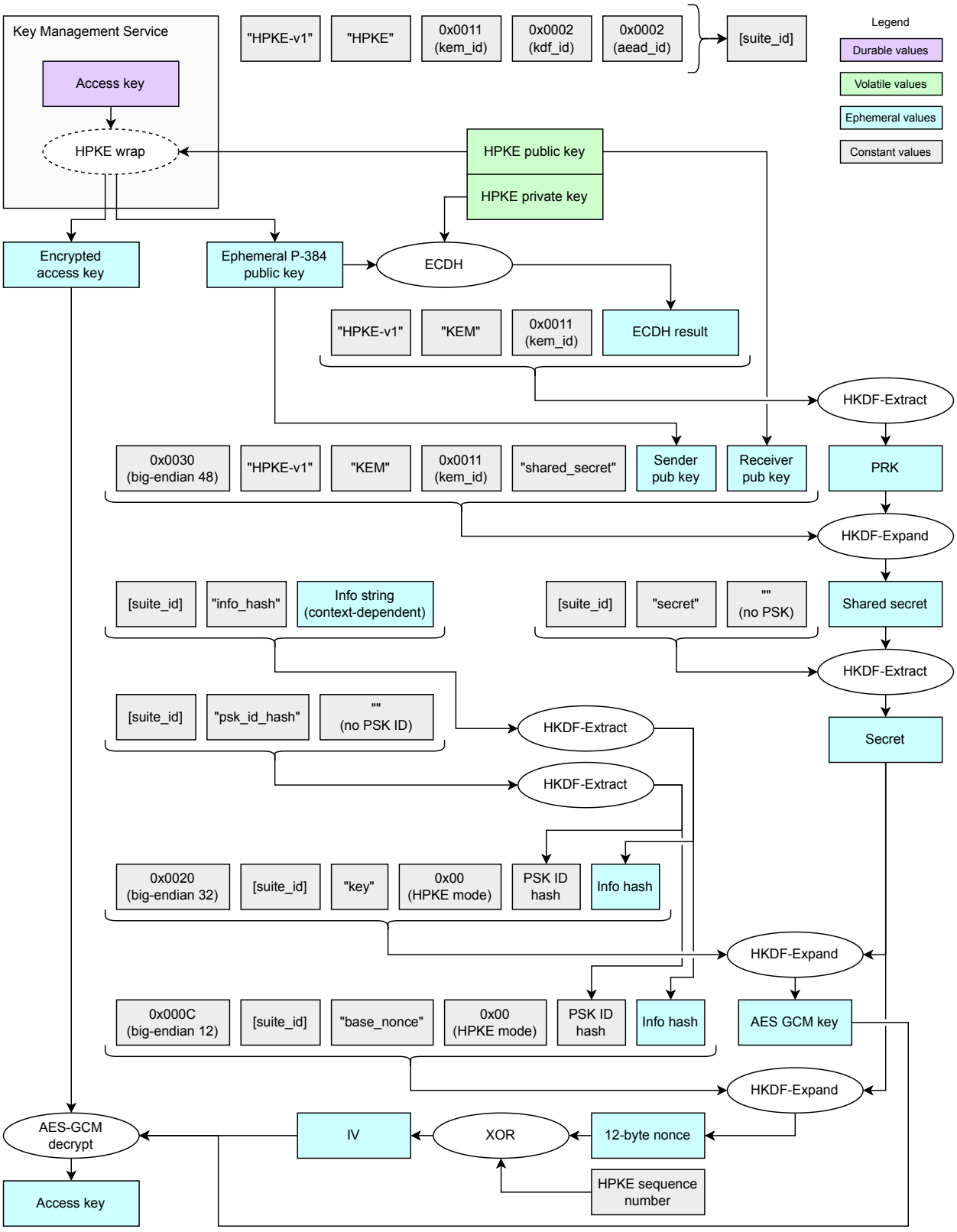

Figure 6: HPKE unwrap for MPK access keys with ECDH

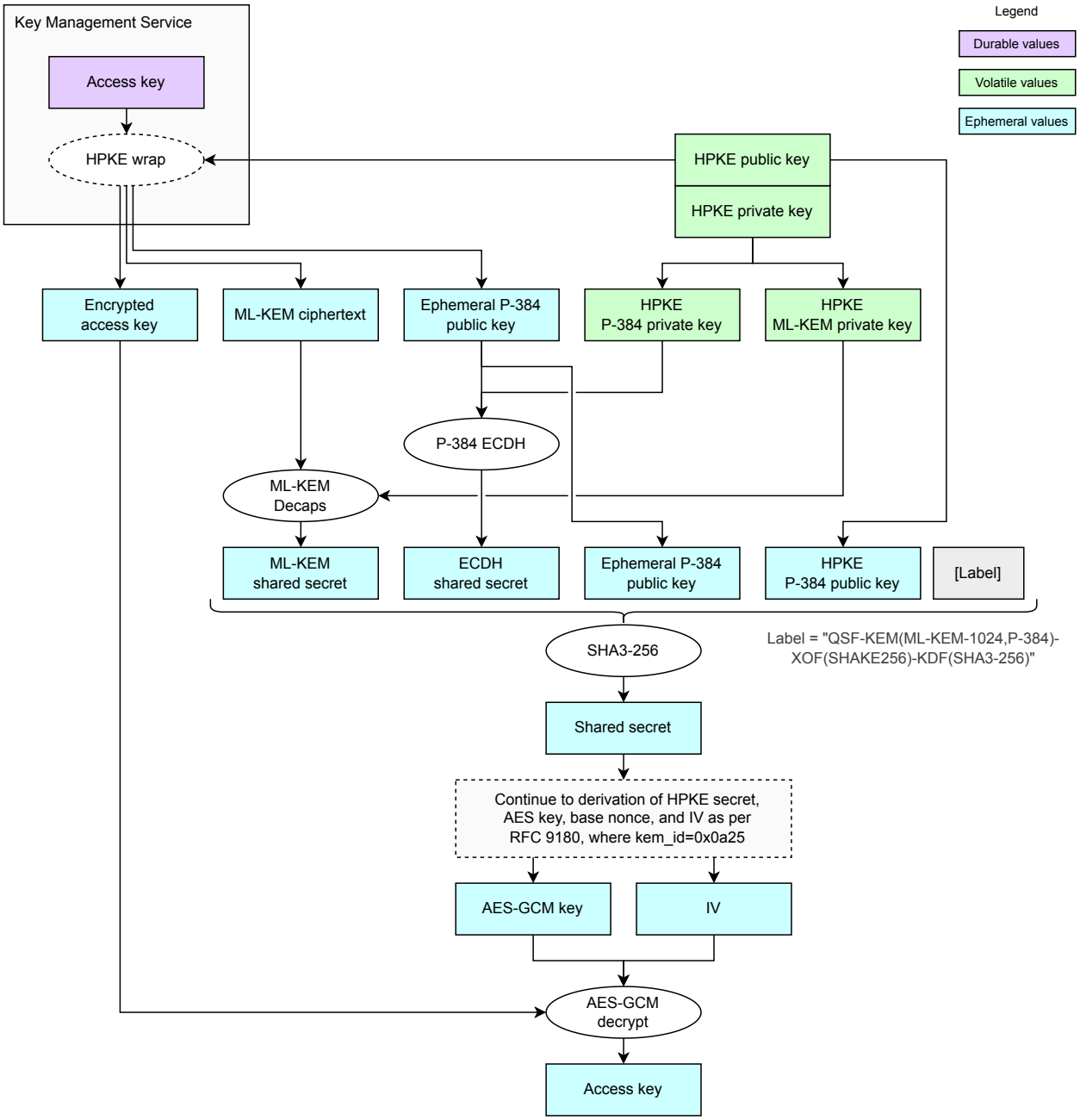

Figure 7: HPKE unwrap for MPK access keys with hybrid ML-KEM + ECDH

The only difference between Figure 6 and Figure 7 is how the shared secret is derived. Operations which produce an AES key and IV from that shared secret are identical, and are omitted from Figure 7 for brevity.

The sequence number is incremented for each decryption operation performed with keys derived from a given HPKE context. With the exception of MPK access key rotation as described in Section 4.5.3.2.3, all flows that accept HPKE payloads will perform a single decryption operation with the computed context. MPK access key rotation will perform two decryption operations and

will increment the sequence number between them.

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

Drive firmware may request that KMB generate an MPK, bound to a given access key.

To identify an MPK, at creation time the drive firmware can provide “metadata” for the MPK, which is included in the generated locked MPK and bound to the ciphertext as AAD. It is outside the scope of this specification how drive firmware populates the metadata field for the MPK. Note that “metadata” in this context refers to metadata about the MPK, and bears no relation to metadata about an MEK.

KMB performs the following steps:

1. Unwrap the given MPK access key using the HPKE keypair held within KMB.
2. Randomly generate an MPK.
3. Derive an MPK encryption key from the HEK, SEK, and decrypted access key.
4. Encrypt the MPK to the MPK encryption key, attaching the given metadata as AAD.
5. Return the encrypted MPK to the drive firmware.

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

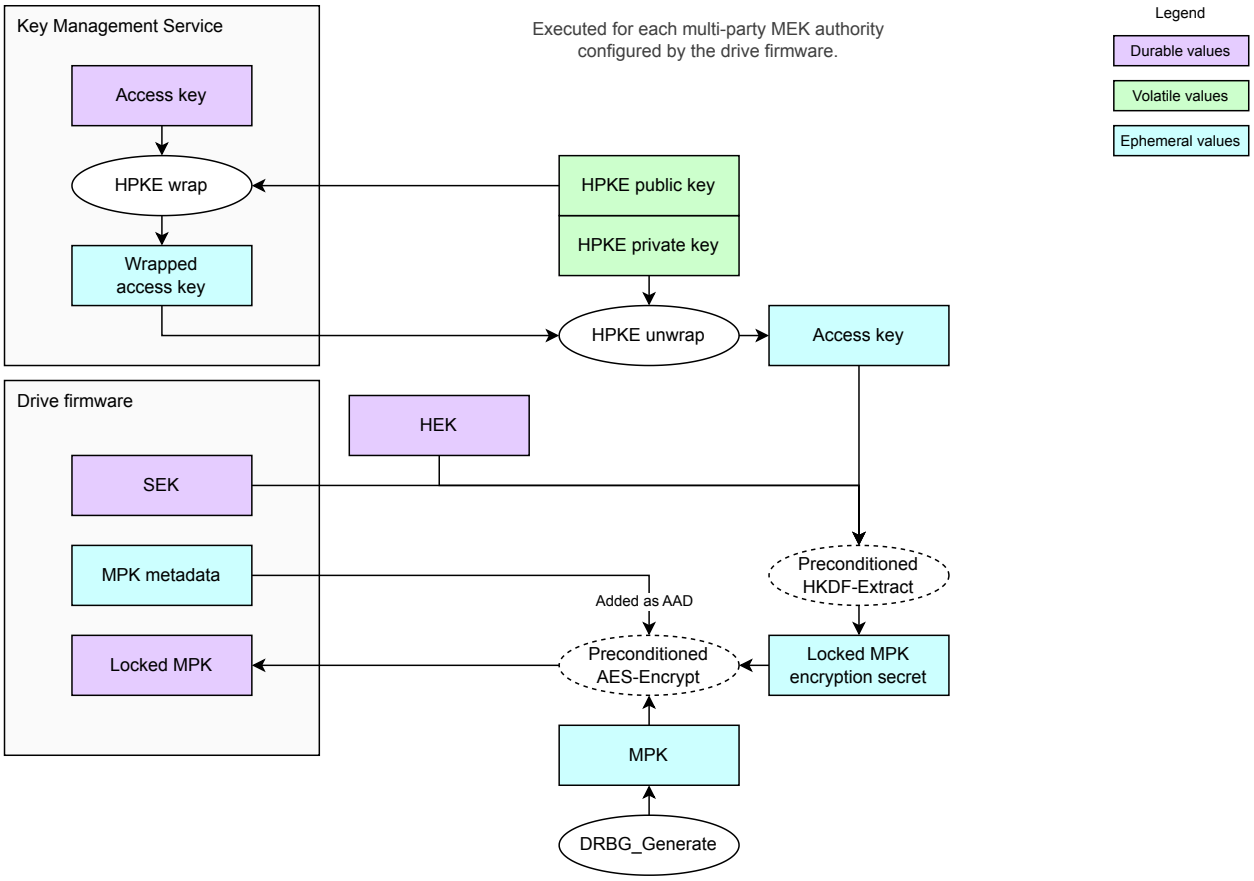

Figure 8: MPK generation

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

To ready an 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, SEK, and decrypted access key.
3. Decrypt the MPK using the MPK encryption key.
4. Encrypt the MPK using the Ready MPK Encryption Key, preserving the MPK metadata.
5. Return the re-encrypted “ready” MPK to the drive firmware.

Drive 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 4.5.6.

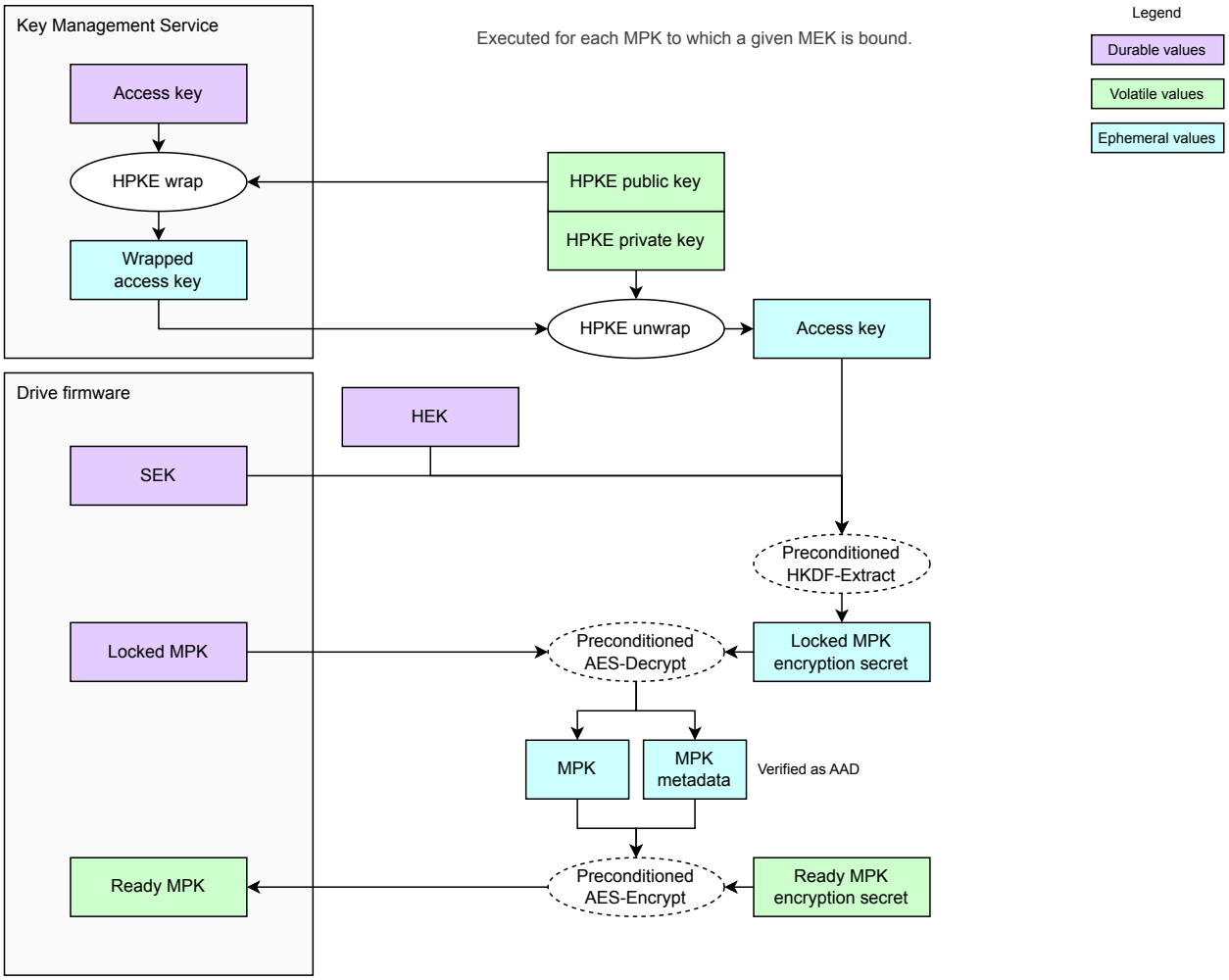

Figure 9: MPK readying

4.5.3.2.3 MPK access key rotation

The access key to which an MPK is bound may be rotated. The user must prove that they have knowledge of both the current and new access key before a rotation is allowed. This is accomplished by the user wrapping both their current and new access key in the same HPKE payload. KMB performs the following steps:

1. Unwrap the given current access key and new access key using the HPKE keypair held within KMB, incrementing the HPKE sequence number between each decryption operation. Note that the sequence number increment is a side-effect of the `ContextR.Open()` HPKE operation.
2. Derive the current MPK encryption key from the HEK, SEK, and decrypted current access key.
3. Derive the new MPK encryption key from the HEK, SEK, and decrypted new access key.
4. Decrypt the MPK using the current MPK encryption key.
5. Encrypt the MPK using the new MPK encryption key, preserving the MPK metadata.

6. Return the re-encrypted MPK to the drive firmware.

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

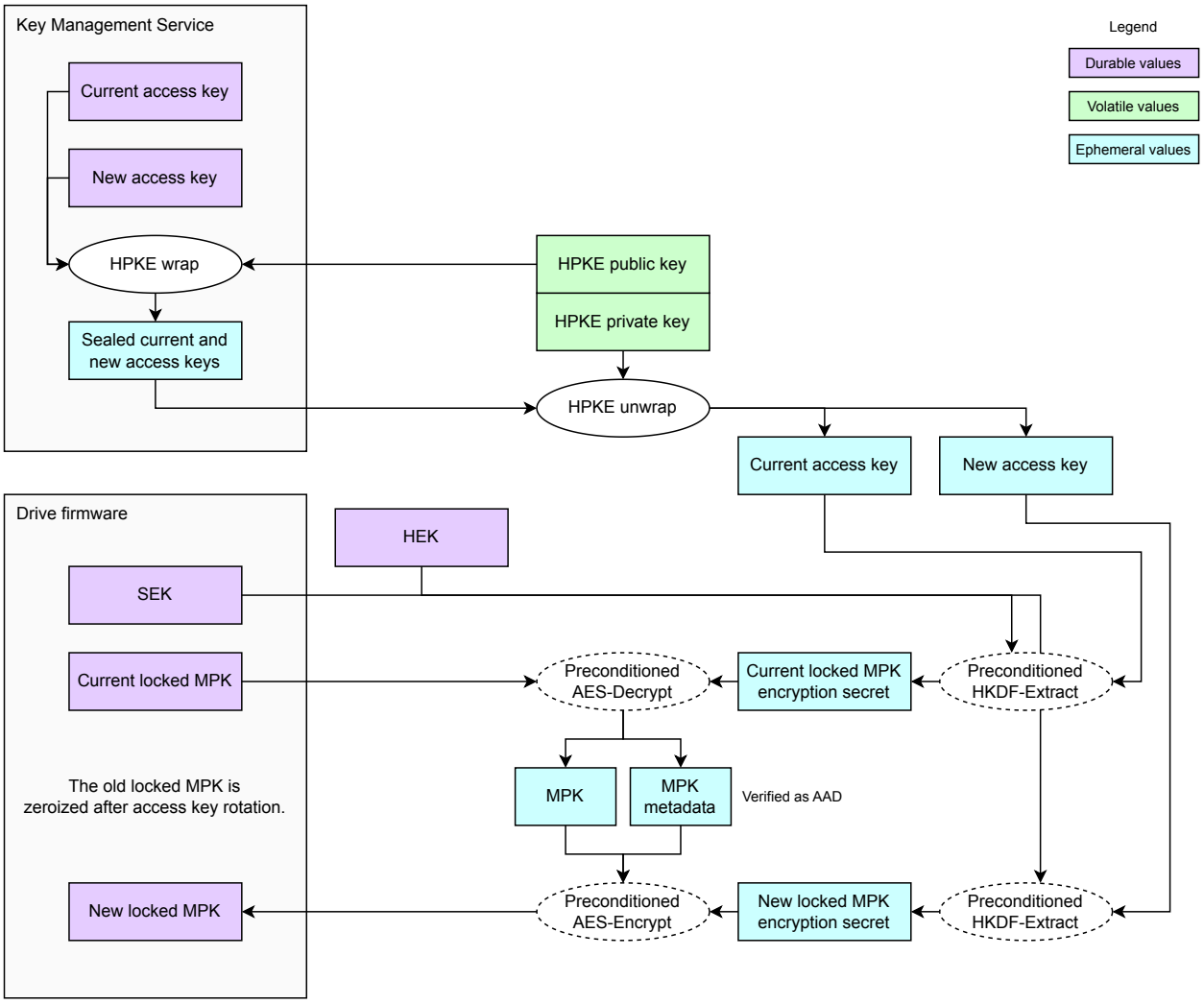

Figure 10: MPK access key rotation

4.5.3.2.4 MPK access key testing

A user that has bound an access key to an MPK may wish to confirm that their access key is in fact bound to the MPK. This can be useful after an access key rotation, to ensure that the rotation executed successfully on the storage device.

To test an MPK, KMB receives a wrapped MPK, wrapped MPK access key, and freshness nonce. KMB then 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, SEK, and decrypted access key.

3. Verify the integrity of the MPK ciphertext and AAD using the MPK encryption key, discarding the decrypted MPK in the process.
4. Calculate and return the digest SHA2-384(MPK metadata || access key || nonce).

The user can then verify that the returned digest matches their expectation.

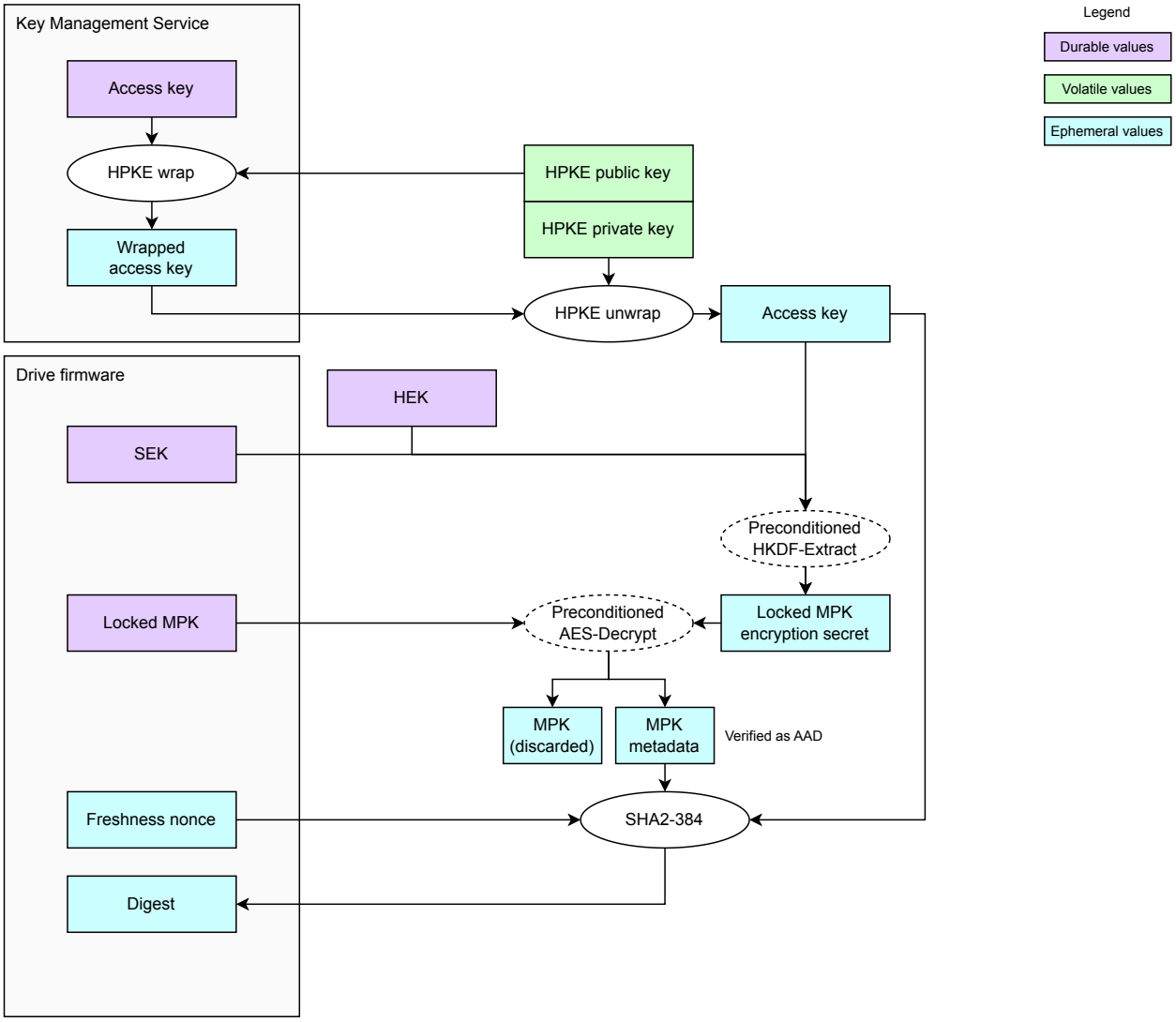

Figure 11: MPK access key testing

4.5.4 DPKs

Drive 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 4.5.6.

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. Drive firmware logic which decrypts MEKs can be repurposed to produce the DPK.

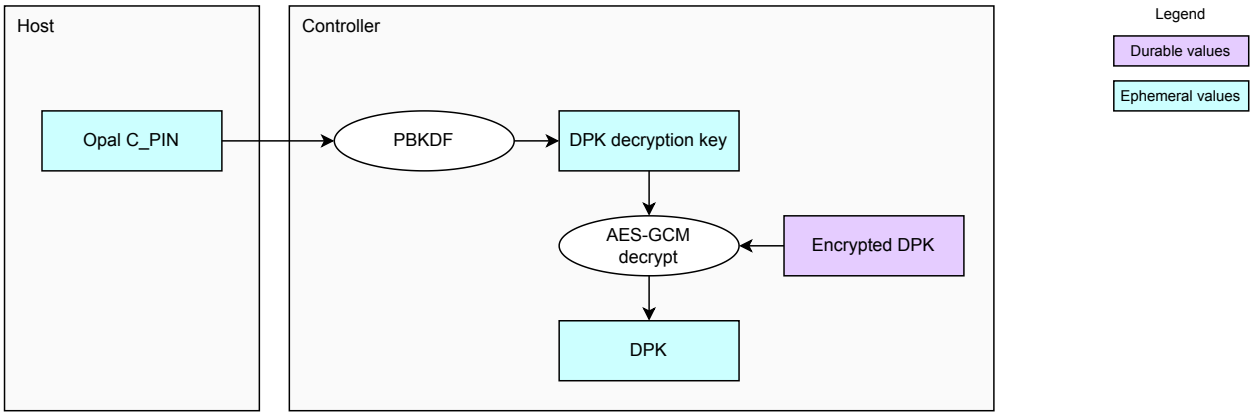

Figure 12: Example drive flow to decrypt a DPK based on a host-provided Opal C_PIN

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.

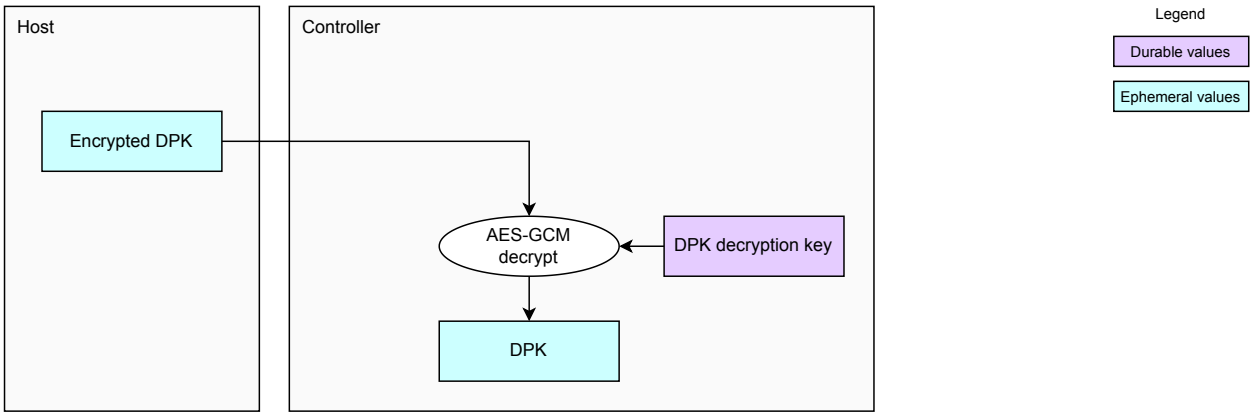

Figure 13: Example drive flow to accept an injected DPK

4.5.5 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 drive firmware, and may be stored in flash. If the SEK is zeroized, drive firmware is responsible for enforcing that MEKs are not loaded.

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

4.5.5.1 HEK derivation inputs

4.5.5.1.1 Stable Identity Key

Caliptra features a Stable Identity Key (SIK), which is a secret derived from Caliptra’s LDevID CDI 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 Caliptra firmware Security Version Numbers (SVNs), with the property that Caliptra firmware whose SVN is X can only obtain Stable Identity keys bound to SVNs less than or equal to X. The SIK used by KMB is the key ladder secret bound to SVN 0, ensuring that Caliptra firmware of any SVN can wield it.

4.5.5.1.2 Ratchet secrets

Caliptra with L.O.C.K. features a series of n 256-bit ratchet secrets present in a fuse bank, dubbed $R_0..R_{n-1}$, where $4 \leq n \leq 16$. The vendor is responsible for determining the number of ratchet secrets available in fuses. These ratchet secrets are only programmable and readable by Caliptra Core, via the Caliptra fuse controller. KMB transitions each ratchet secret individually from blank → randomized → zeroized. R_x is only randomized once R_{x-1} has been zeroized. KMB zeroizes a ratchet secret by blowing every bit.

4.5.5.2 HEK derivation

Table 1 describes the states between which the HEK transitions over the lifespan of the device. The lifecycle state of the HEK is determined by Caliptra’s lifecycle state as well as the state of the ratchet secret fuse bank.

Table 1: HEK lifecycle states

Caliptra lifecycle state	**Ratchet secret state, where x is the current ratchet slot**	**Derivation inputs**	**Reported HEK state**
Unprovisioned or Manufacturing	[Any]	SIK	HEK_AVAIL_PREPROD
Production	R_0 is unprogrammed.	N/A	HEK_UNAVAIL_EMPTY
	R_x is randomized.	SIK and R_x	HEK_AVAIL_PROD
	R_x is zeroized.	N/A	HEK_UNAVAIL_ZEROIZED
	R_x is corrupted.	N/A	HEK_UNAVAIL_CORRUPTED
	Every ratchet secret is zeroized, and the permanent-HEK fuse bit has been set.	SIK	HEK_AVAIL_PERMANENT

Table 2 describes the commands that KMB exposes to manage the HEK lifecycle.

Table 2: HEK lifecycle management commands

Command	Description
PROGRAM_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 ratchet secret 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 the SIK. May only be invoked once all ratchet secrets are zeroized. May be re-attempted if an error caused the permanent-HEK fuse indicator to be left in an invalid state. Upon success, if the SEK is programmed, MEKs may be loaded.

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

Figure 14 illustrates HEK derivation, for the states where the HEK is available.

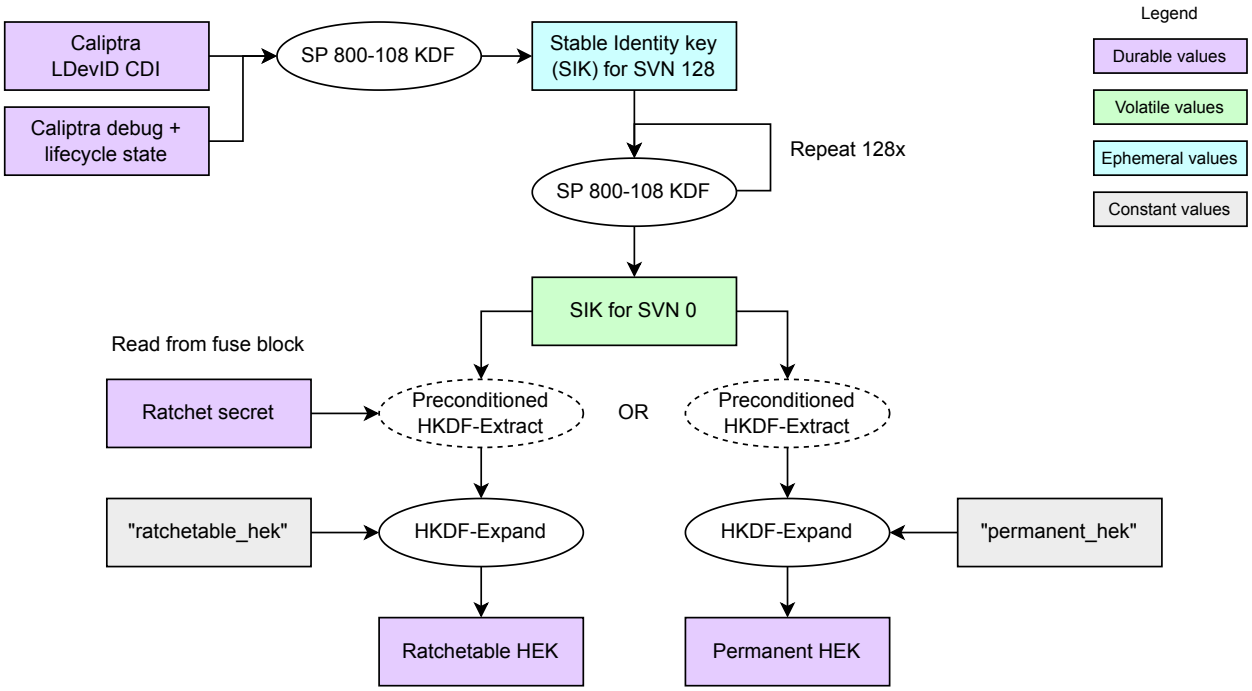

Figure 14: HEK derivation

4.5.5.3 HEK and SEK lifecycle

Figure 15 illustrates the case where a drive ships with four ratchet secrets.

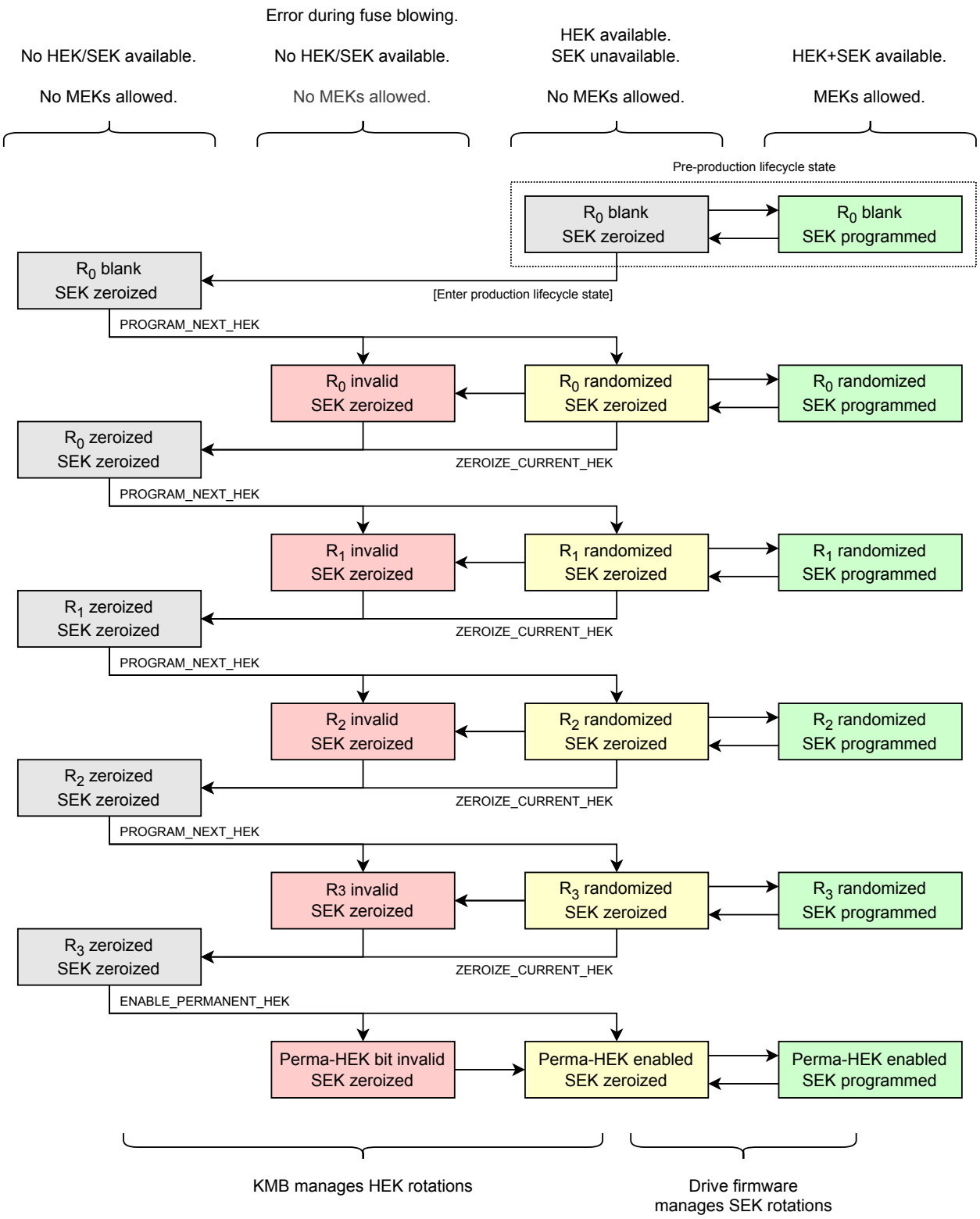

Figure 15: HEK and SEK state machine

A device out of manufacturing must be in the production lifecycle state and have a randomized ratchet secret.

Drive firmware is responsible for enforcing that SEK programming and zeroization follows the state machine illustrated in Figure 15. Specifically:

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

4.5.6 MEKs

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

Figure 16 provides details on how the SEK, HEK, DPK, and MPKs are used to produce the MEK secret, which then either encrypts/decrypts a random MEK or is used to compute a derived MEK.

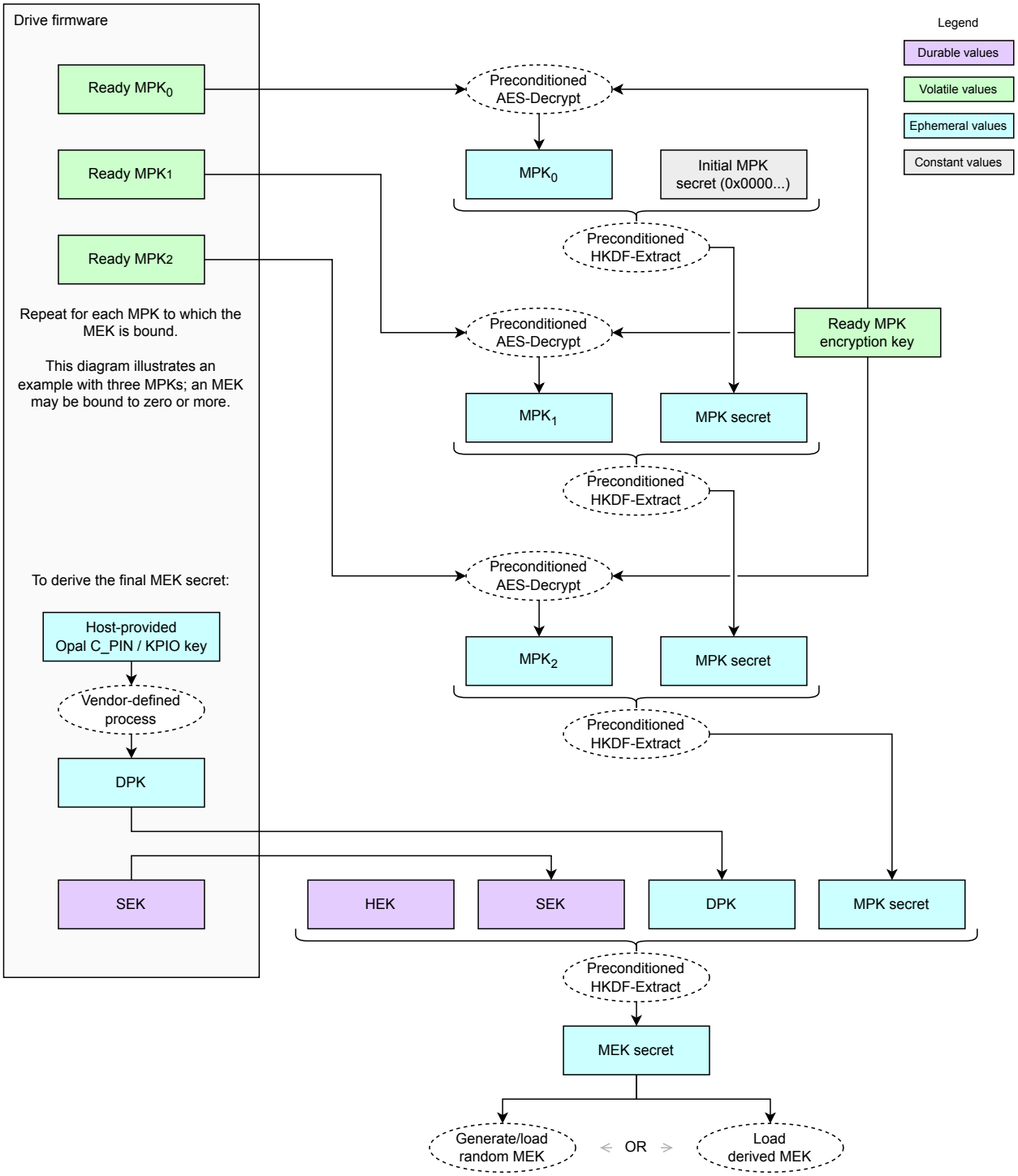

Figure 16: MEK secret derivation

Note: it is the drive firmware's responsibility to ensure that MPKs are mixed in the correct order for a given MEK.

See Section 4.5.6.1 for how the MEK secret is used to generate and load random MEKs. See

Section 4.5.6.2 for how the MEK secret is used to derive deterministic MEKs.

4.5.6.1 Generating and loading a random MEK

Figure 17 and Figure 18 elide the process of deriving the MEK secret, as those details are captured in Figure 16. When an MEK is generated or loaded, the inputs given in Figure 16 are provided by drive firmware so that KMB may derive the MEK secret.

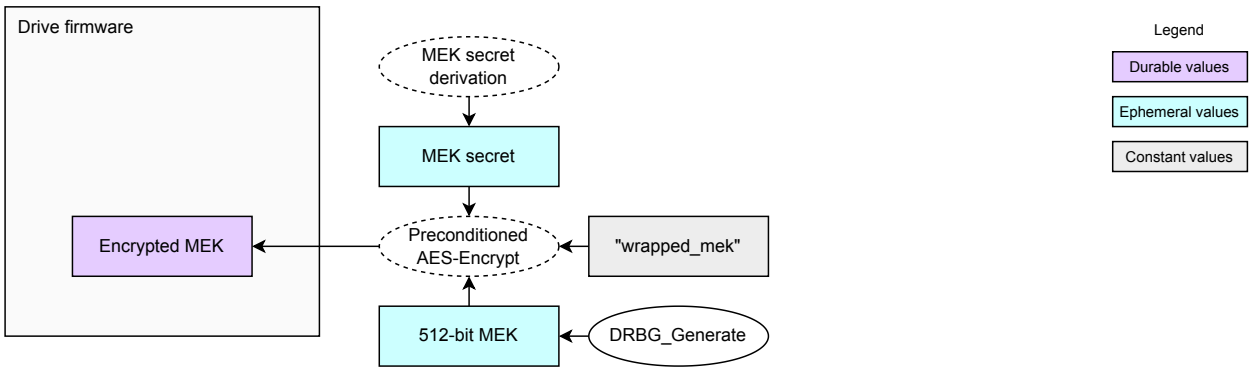

Figure 17: Generating an MEK

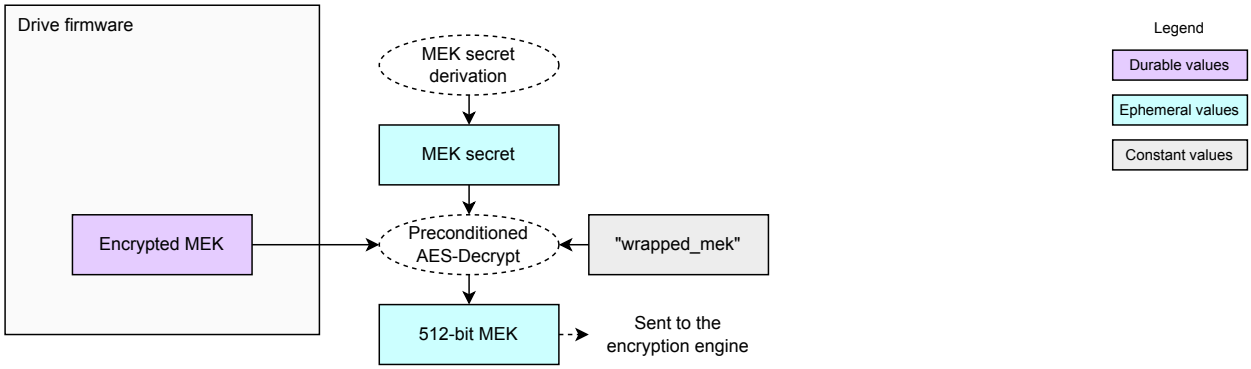

Figure 18: Loading an MEK

4.5.6.2 Deriving an MEK

Figure 19 elides the process of deriving the MEK secret, as those details are captured in Figure 16. When an MEK is derived, the inputs given in Figure 16 are provided by drive firmware so that KMB may derive the MEK secret.

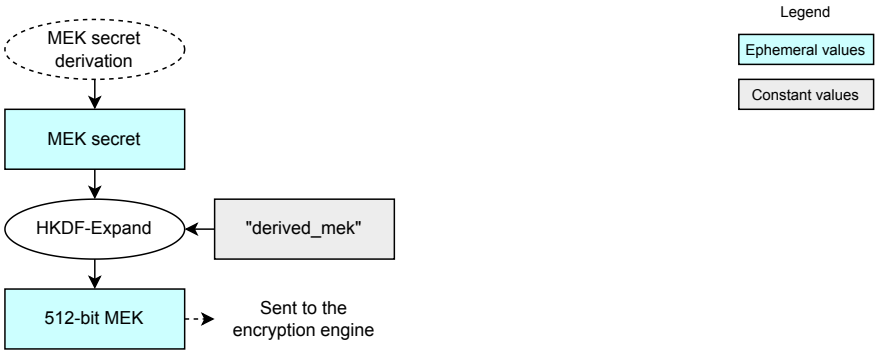

Figure 19: Deriving an MEK

4.5.6.3 AES-XTS considerations

The MEK sent to the encryption engine may be used as an AES-XTS key. FIPS 140-3 IG [12] section C.I 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… *Key_1* and *Key_2* shall be generated and/or established independently according to the rules for component symmetric keys from NIST **SP 800-133rev2**, Sec. 6.3. The module **shall** check explicitly that *Key_1* ≠ *Key_2*, regardless of how *Key_1* and *Key_2* are obtained."

SP 800-133 [7] section 6.3 states: "The independent generation/establishment of the component keys K_1, …, K_n is interpreted in a computational and a statistical sense; that is, the computation of any particular K_i value does not depend on any one or more of the other K_i values, and it is not feasible to use knowledge of any proper subset of the K_i values to obtain any information about the remaining K_i values."

SP 800-108 [13] states that "the output of a key-derivation function is called the derived keying material and may subsequently be segmented into multiple keys. Any disjoint segments of the derived keying material (with the required lengths) can be used as cryptographic keys for the intended algorithms."

As the MEK sent to the encryption engine is either randomly or pseudorandomly generated, it satisfies the above constraints. Disjoint segments of derived keying material computed from a pseudorandom function are statistically and computationally independent.

When in AES-XTS mode, the encryption engine will be responsible for performing the inequality check for Key_1 and Key_2 stipulated by FIPS 140-3 IG section C.I. If this check fails, the encryption engine must report a vendor-defined error. This document does not specify how drive firmware should handle this error case.

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.

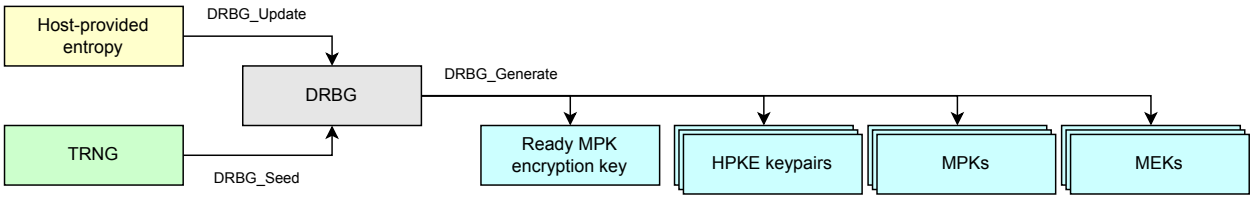

Figure 20: DRBG usage

Note that Caliptra hardware only supports the `DRBG_Update` operation when Caliptra is instantiated with an internal TRNG [14].

4.5.8 Boot-time initialization

At initial boot, Caliptra firmware performs the following steps:

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

The Ready MPK encryption key is not initialized at boot-up, as it is generated lazily upon first use. See Section 4.5.3 for details.

4.5.9 Authorization

KMB exposes commands that allows drive firmware to manage the lifecycle of MPKs, HEKs, and MEKs. These lifecycle events have the potential to (and are in many cases designed to) trigger user data loss. The host interface that allows the host to trigger these lifecycle events must therefore implement authorization controls to ensure user data is not improperly destroyed. Drive firmware is responsible for implementing this authorization layer, the details of which are outside the scope of this specification.

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 drive firmware, and enables the drive to manage MEKs and associated keys.

4.6.1 Encryption engine interface

This section defines the interface between KMB and an encryption engine. An encryption engine is used to encrypt/decrypt user data. 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. The interface defined in this section is used to load MEKs from KMB to the encryption engine or to cause the encryption engine to unload (i.e., remove) loaded MEKs. MEKs shall not be accessible to the drive firmware as they are being transferred between KMB and the encryption engine.

4.6.1.1 Overview

The encryption engine uses an MEK stored in volatile memory 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. Drive 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. Drive firmware has full control on metadata and optional aux. Figure 21 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.

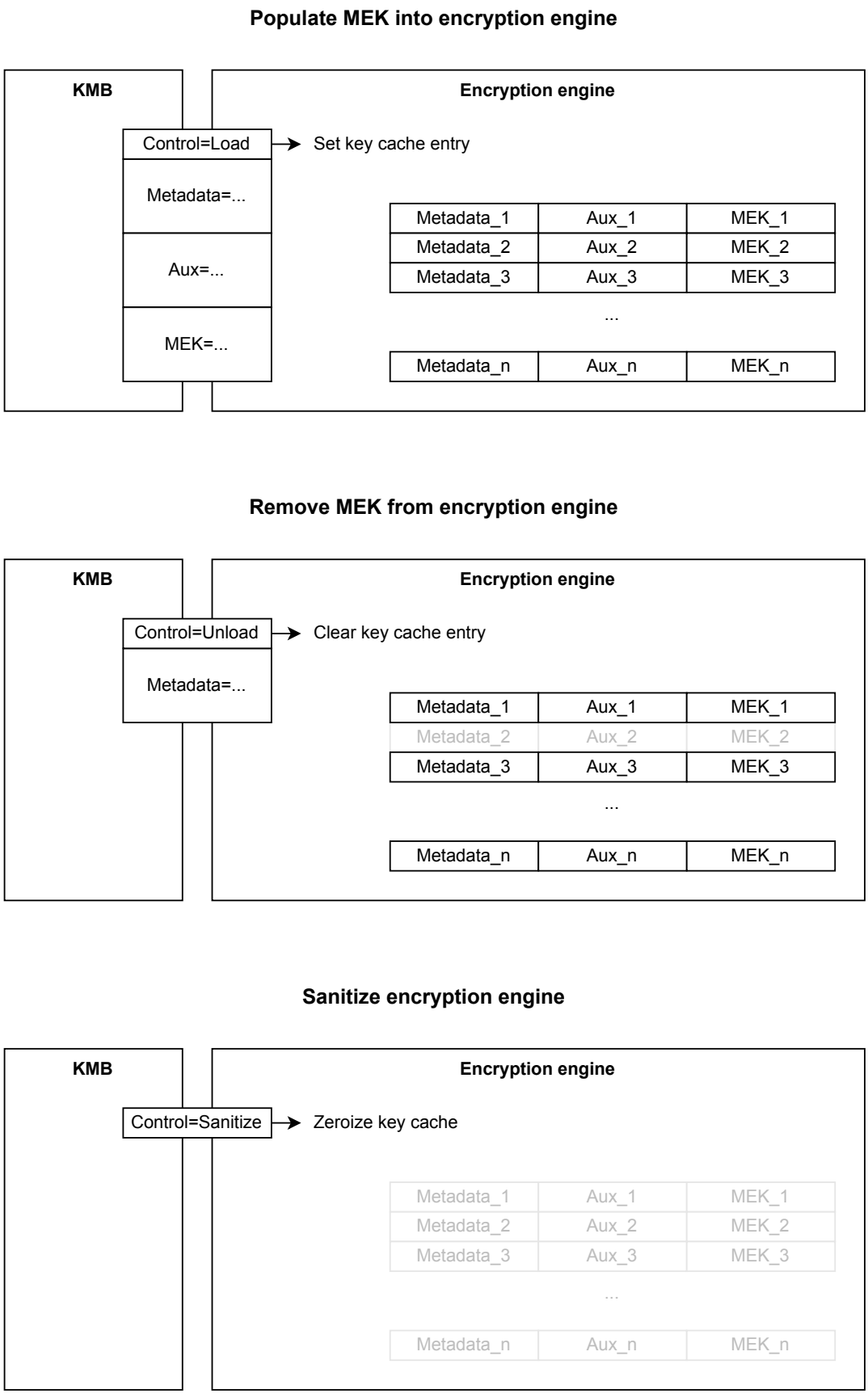

Figure 21: KMB to encryption engine SFR interface

The behavior of I/O through the encryption engine during the execution of an SFR command is vendor-defined.

4.6.1.2 Special Function Registers

KMB uses Special Function Registers (SFRs) to communicate with the encryption engine as shown in Table 3 and each of the following subsections which describe the registers.

Table 3: 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 4 defines the Control register used to sequence the execution of a command and obtain the status of that command.

Table 4: Offset SFR_Base + 0h: CTRL – Control

Bits	Type	Reset	Description
31	RO	0h	**Ready (RDY):** After an NVM Subsystem Reset occurs, the encryption engine sets this bit to 1b, indicating that the encryption engine is ready to execute commands. If this bit is 0b, then the encryption engine is not ready to execute commands.
30:20	RO	0h	Reserved
19:16	RO	0h	**Error (ERR):** When the encryption engine sets the DONE bit to 1b, the encryption engine sets this field to 0000b if the command specified by the CMD field succeeded, or a non-zero value if the command failed. See Table 5. Encryption engine error codes are surfaced back to drive firmware. If the DONE bit is set to 1b by KMB, then this 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 6.

(continued on next page)

(continued from previous page)

Bits	Type	Reset	Description
1	RW	0b	**Done (DONE):** 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 1b and this bit is 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 by KMB 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. When the encryption engine writes the value of 1b to the DONE field, the value of the EXE bit does not change.
0	RW	0b	**Execute (EXE):** A write by KMB 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 encryption engine sets the EXE bit to 0b.

Table 5: CTRL error codes

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

Table 6: 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 **DONE**
    1. **ERR:** either 0b or a non-zero value depending on the execution result
    2. **DONE:** 1b
3. The KMB writes **DONE**
    1. **DONE:** 1b
4. The encryption engine writes **CMD**, **ERR**, **DONE** and **EXE**
    1. **CMD:** 0h
    2. **ERR:** 0h
    3. **DONE:** 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 **DONE** to be 1
3. The KMB writes **DONE**
    1. **DONE:** 1b
4. The KMB waits **DONE** to be 0

Since the Control register is a part of the encryption engine whose implementation can be unique to each vendor, behaviors of the Control register with respect to unexpected flows 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 the EXE bit. Vendors shall ensure that any customizations remain compatible with the interface defined in this specification.

4.6.1.2.2 Metadata register

Table 7 defines the Metadata register used to pass additional data related to the MEK.

Table 7: 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 drive 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 22.

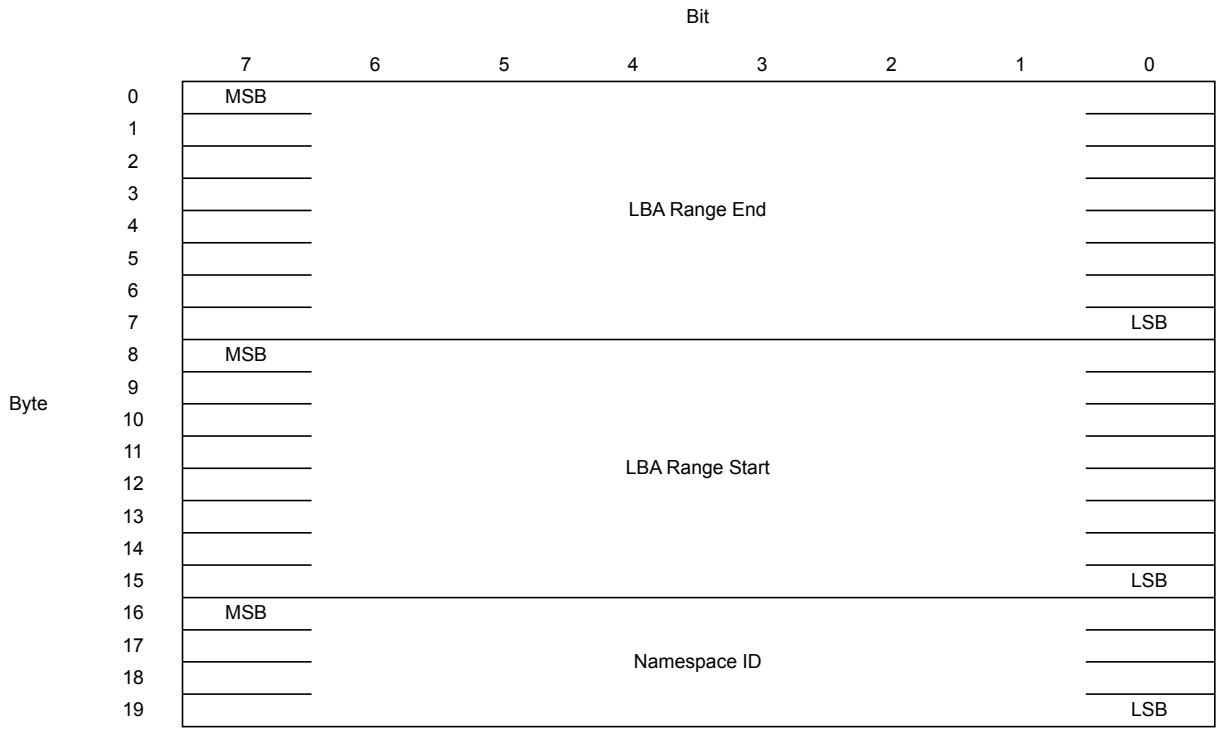

Figure 22: LBA range based metadata format

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 23 shows an example of **METD** in such cases.

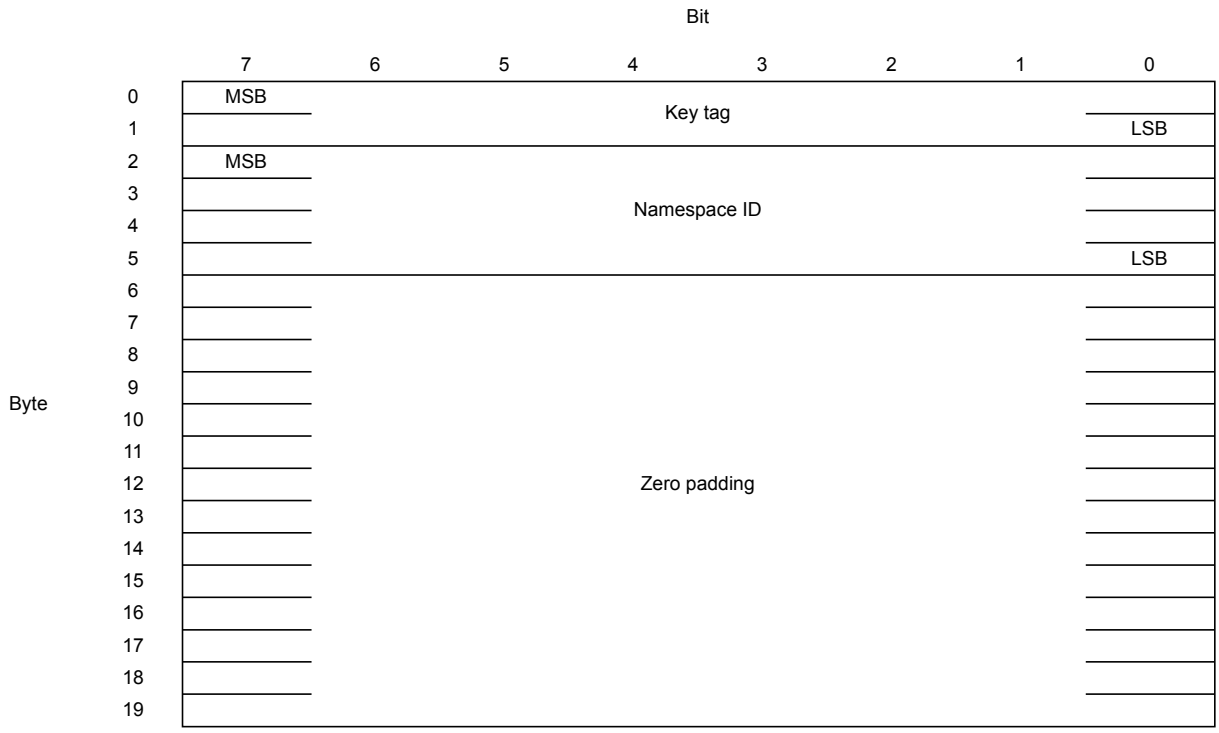

Figure 23: Key tag based metadata format

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

Table 8: 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 drive firmware instructs the KMB to load an MEK, the drive 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 24 can be used.

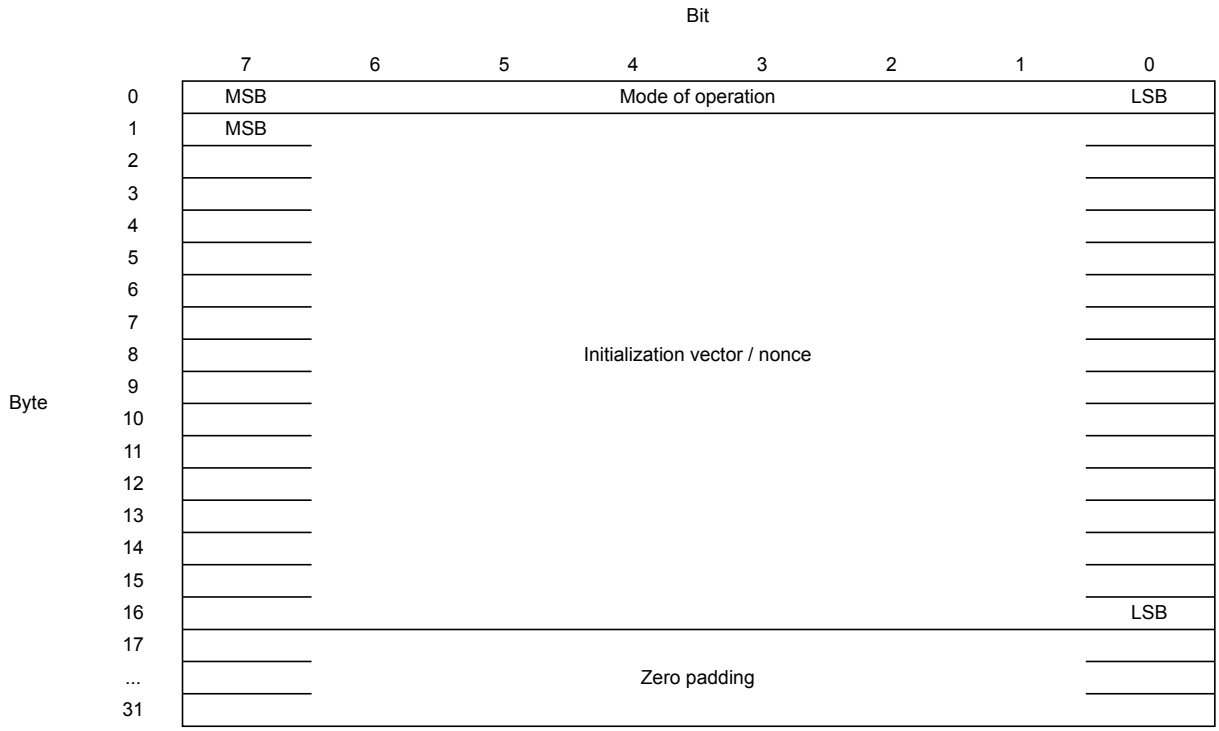

Figure 24: Auxiliary data format example

When the drive firmware instructs KMB to load an MEK, the drive 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 9: 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-256 is presented in Figure 25.

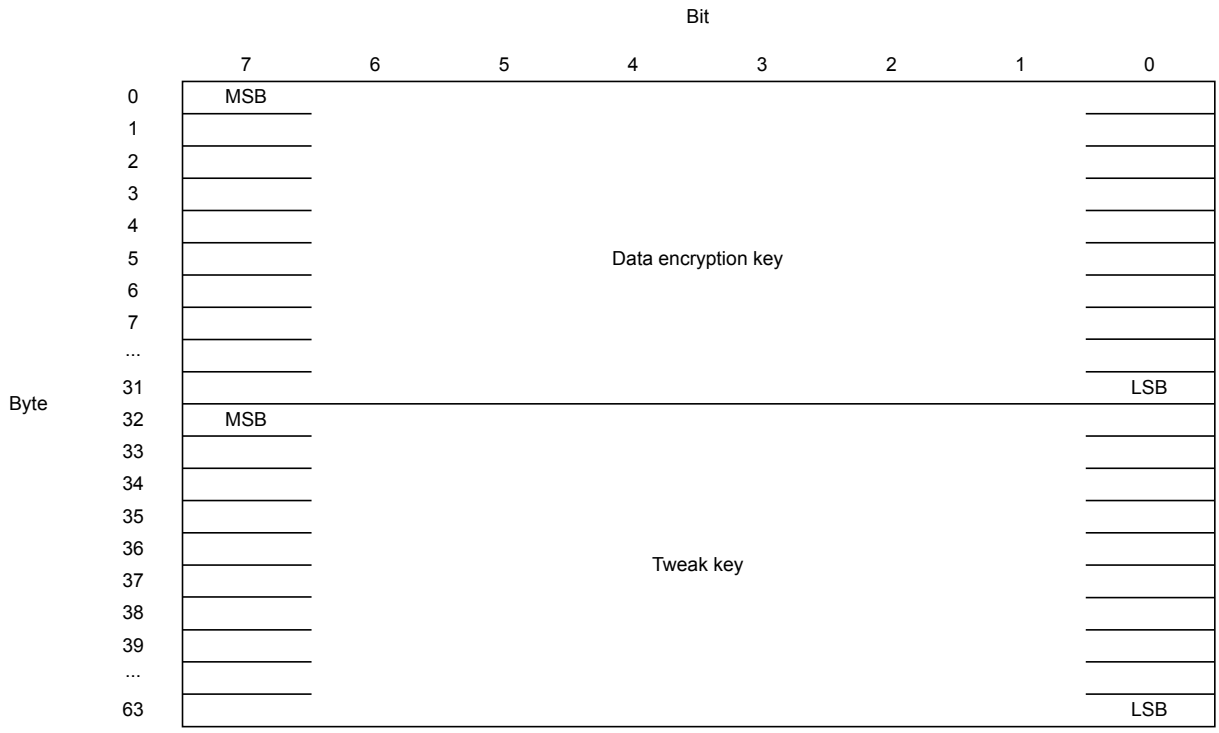

Figure 25: MEK format example for AES-XTS-256

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 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 KMB command sequence

Figure 26 shows a sample command execution. This is an expected sequence when the drive firmware instructs the KMB to load an MEK. This sequence would occur as part of the DERIVE_MEK or LOAD_MEK mailbox commands. The internal behavior of the encryption engine is one of several possible mechanisms, and can be different per vendor.

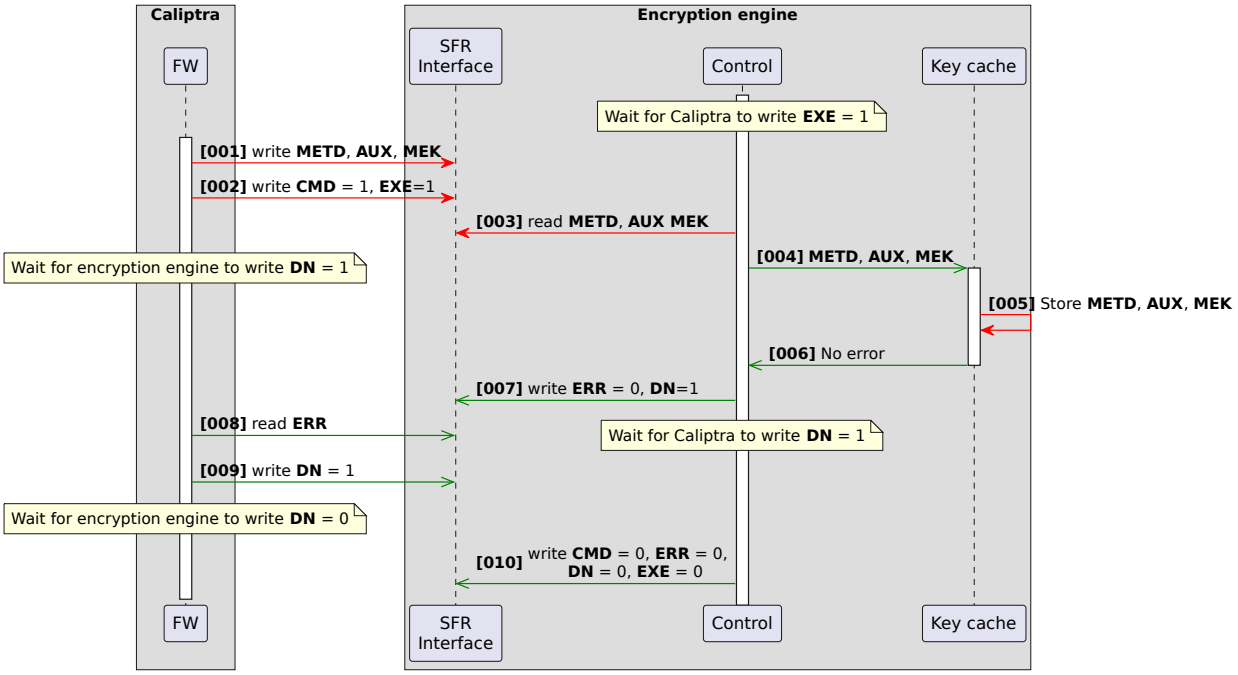

Figure 26: Command execution example for loading an MEK

4.6.2 Mailbox interface

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

4.6.2.1 FIPS status indicator

Each mailbox command returns a `fips_status` field. This provides an indicator of whether KMB is operating in FIPS mode. Table 10 provides the possible values for this field.

Note: all multi-byte fields (i.e. `u16` and `u32`) that are not byte arrays are interpreted as little endian.

Table 10: Values for the FIPS status field

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

4.6.2.2 Encryption engine timeout

Each mailbox command that causes a command to execute on the encryption engine includes a `cmd_timeout` value indicating the amount of time KMB firmware will wait until the command has completed. If this timeout is exceeded, KMB aborts the command and reports a `LOCK_ENGINE_TIMEOUT` result code.

For such commands, KMB firmware will return a `LOCK_EE_NOT_READY` error immediately if the encryption engine is not ready to execute a command.

4.6.2.3 Side-channel mitigations

Several mailbox commands invoke ECDH and/or ML-KEM Decaps. These operations involve deterministic data-dependent operations and are potentially susceptible to timing attacks and power analysis. To mitigate such attacks, Caliptra leverages masking in hardware, and introduces random jitter delays in firmware before handling mailbox commands.

4.6.2.4 GET_STATUS

Exposes a command that allows drive firmware to determine if the encryption engine is ready to process commands as well vendor-defined drive encryption engine status data.

See Figure 28 for the sequence diagram.

Command Code: 0x4753_5441 (“GSTA”)

Table 11: GET_STATUS input arguments

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

Table 12: GET_STATUS output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32[4]	Reserved.
ctrl_register	u32	Value of the CTRL register from the SFR interface.

4.6.2.5 GET_ALGORITHMS

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

See Figure 29 for the sequence diagram.

Command Code: 0x4741_4C47 (“GALG”)

Table 13: GET_ALGORITHMS input arguments

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

Table 14: GET_ALGORITHMS output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32[4]	Reserved.
endorsement_algorithms	u32	Identifies the supported endorsement algorithms: • Byte 0 bit 0: ecdsa_secp384r1_sha384 [15] • Byte 0 bit 1: ml-dsa-87 [16]
hpke_algorithms	u32	Identifies the supported HPKE algorithms: {kem/aead/kdf}_id • Byte 0 bit 0: 0x0011, 0x0002, 0x0002 [6] • Byte 0 bit 1: 0x0a25, 0x0002, 0x0002 [11]
access_key_sizes	u32	Indicates the length of plaintext access keys: • Byte 0 bit 0: 256 bits

Each of the `endorsement_algorithms`, `hpke_algorithms`, and `access_key_sizes` fields shall be reported as a non-zero value.

4.6.2.6 CLEAR_KEY_CACHE

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

Command Code: 0x434C_4B43 (“CLKC”)

Table 15: CLEAR_KEY_CACHE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
cmd_timeout	u32	Timeout in ms for command to encryption engine to complete.

Table 16: CLEAR_KEY_CACHE output arguments

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

(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.

4.6.2.7 ENDORSE_ENCAPSULATION_PUB_KEY

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

See Figure 30 for the sequence diagram.

Command Code: 0x4E45_505B (“EEPK”)

Table 17: ENDORSE_ENCAPSULATION_PUB_KEY input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
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 18: ENDORSE_ENCAPSULATION_PUB_KEY output arguments

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

4.6.2.8 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.

See Figure 31 for the sequence diagram.

Command Code: 0x5245_4E4B (“RENK”)

Table 19: ROTATE_ENCAPSULATION_KEY input arguments

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

Table 20: ROTATE_ENCAPSULATION_KEY output arguments

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

4.6.2.9 GENERATE_MPK

This command unwraps the specified access key, generates a random MPK, then encrypts the MPK with KDF(HEK || SEK || access_key, “MPK”) which is returned for the drive to persistently store. The given `metadata` is placed in the `metadata` field of the returned MPK to cryptographically tie them together.

See Figure 32 for the sequence diagram.

Command Code: 0x474D_504B (“GMPK”)

Table 21: GENERATE_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
metadata_len	u32	Length of the metadata argument.
info_len	u32	Length of the info argument.
metadata	u8[metadata_len]	Metadata for the MPK.
info	u8[info_len]	Info argument to use with HPKE unwrap.

(continued on next page)

(continued from previous page)

Name	Type	Description
sealed_access_key	SealedAccessKey	HPKE-sealed access key.

Table 22: GENERATE_MPK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32	Reserved.
encrypted_mpk	LockedMpk	MPK encrypted to access_key.

4.6.2.10 REWRAP_MPK

This command unwraps current_access_key and encrypted new_access_key from sealed_access_keys. Then current_access_key is used to decrypt new_access_key. The specified MPK is decrypted using KDF(HEK || SEK || current_access_key, “MPK”). A new MPK is encrypted with the output of KDF(HEK || SEK || new_access_key, “MPK”). The new encrypted MPK is returned.

The drive stores the returned new encrypted MPK and zeroizes the old encrypted MPK.

See Figure 34 for the sequence diagram.

Command Code: 0x5245_5750 (“REWP”)

Table 23: REWRAP_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
info_len	u32	Length of the info argument.
info	u8[info_len]	Info argument to use with HPKE unwrap.
current_locked_mpk	LockedMpk	Current MPK to be rewrapped.
sealed_access_keys	SealedCurrentAndNewAccessKey	HPKE-sealed current and new access keys.

Table 24: REWRAP_MPK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32	Reserved.
new_locked_mpk	LockedMpk	MPK encrypted to new_access_key.

4.6.2.11 READY_MPK

This command decrypts `sealed_access_key`. Then the decrypted access_key is used to decrypt `locked_mpk` using KDF(HEK || SEK || access_key, “MPK”). A “ready” MPK is encrypted with the Ready MPK Encryption Key. The encrypted ready MPK is returned.

See Figure 33 for the sequence diagram.

Command Code: 0x524D_504B (“RMPK”)

Table 25: READY_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
info_len	u32	Length of the info argument.
info	u8[info_len]	Info argument to use with HPKE unwrap.
sealed_access_key	SealedAccessKey	HPKE-sealed access key.
locked_mpk	LockedMpk	MPK encrypted to the HEK and access key.

Table 26: READY_MPK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32	Reserved.
ready_mpk	ReadyMpk	MPK encrypted to Ready MPK Encryption Key.

4.6.2.12 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.

See Figure 35 for the sequence diagram.

Command Code: 0x4D4D_504B (“MMPK”)

Table 27: MIX_MPK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
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.
ready_mpk	ReadyMpk	MPK encrypted to the Ready MPK Encryption Key.

Table 28: MIX_MPK output arguments

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

4.6.2.13 TEST_ACCESS_KEY

This command is used by the host to check the input access key is associated with the given MPK. The `nonce` is a random value to be included in the digest calculation to prevent response replay attacks. The output is calculated as SHA2-384(metadata || decrypted access key || nonce). The metadata is taken from the provided MPK’s `metadata` field.

See Figure 36 for the sequence diagram.

Command Code: 0x5441_434B (“TACK”)

Table 29: TEST_ACCESS_KEY input arguments

Name	Type	Description

(continued on next page)

(continued from previous page)

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
info_len	u32	Length of the info argument.
info	u8[info_len]	Info argument to use with HPKE unwrap.
nonce	u8[32]	Host-provided random value.
locked_mpk	LockedMpk	Locked MPK associated with the access key.
sealed_access_key	SealedAccessKey	HPKE-sealed access key.

Table 30: TEST_ACCESS_KEY output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
digest	u8[48]	SHA-384 hash of the MPK's metadata, decrypted access key, and nonce.

4.6.2.14 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 31: GENERATE_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
dpk	u8[32]	Data Protection Key.

Table 32: GENERATE_MEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
fips_status	u32	Indicates if the command is FIPS approved or an error.
reserved	u32	Reserved.
wrapped_mek	WrappedMek	MEK encrypted to the derived MEK encryption key.

4.6.2.15 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 format is vendor-defined and specifies the information to the encryption engine on where within the key cache the MEK is loaded.

See Figure 37 for the sequence diagram.

Command Code: 0x4C4D_454B (“LMEK”)

Table 33: LOAD_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
dpk	u8[32]	Data Protection Key.
metadata	u8[20]	Metadata for MEK to load into the drive encryption engine (i.e. NSID + LBA range).
aux_metadata	u8[32]	Auxiliary metadata for the MEK (optional; i.e. operation mode).
wrapped_mek	WrappedMek	MEK encrypted to the derived MEK encryption key.
cmd_timeout	u32	Timeout in ms for command to encryption engine to complete.

Table 34: LOAD_MEK output arguments

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

4.6.2.16 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 format is vendor-defined and specifies the information to the encryption engine on where within the key cache the MEK is loaded.

Command Code: 0x444D_454B (“DMEK”)

Table 35: DERIVE_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek	u8[32]	Soft Epoch Key.
dpk	u8[32]	Data Protection Key.
metadata	u8[20]	Metadata for MEK to load into the drive encryption engine (i.e. NSID + LBA range).
aux_metadata	u8[32]	Auxiliary metadata for the MEK (optional; i.e. operation mode).
cmd_timeout	u32	Timeout in ms for command to encryption engine to complete.

Table 36: DERIVE_MEK output arguments

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

4.6.2.17 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 format is vendor-defined and specifies the information to the encryption engine on where within the key cache, the MEK is loaded.

See Figure 39 for the sequence diagram.

Command Code: 0x554D_454B (“UMEK”)

Table 37: UNLOAD_MEK input arguments

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

Table 38: UNLOAD_MEK output arguments

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

4.6.2.18 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 39: ENUMERATE_KEM_HANDLES input arguments

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

Table 40: ENUMERATE_KEM_HANDLES output arguments

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

Table 41: KemHandle contents

Name	Type	Description
handle	u32	Handle for KEM keypair held in KMB memory.
hpke_algorithm	u32	HPKE algorithm. Shall be a bit value indicated as supported in Table 14.

4.6.2.19 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. This command does not modify the `active_hek_slot` reported in REPORT_EPOCH_KEY_STATE.

Command Code: 0x5A43_484B (“ZCHK”)

Table 42: ZEROIZE_CURRENT_HEK input arguments

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

Table 43: ZEROIZE_CURRENT_HEK output arguments

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

4.6.2.20 PROGRAM_NEXT_HEK

This command generates a random key and programs it into the next-available HEK slot. The `active_hek_slot` reported in Section 4.6.2.22 is incremented after successfully programming the random key, with the exception that when this command is used to provision the first ratchet secret, the `active_hek_slot` is not incremented, because it is already set to the first slot.

Command Code: 0x504E_484B (“PNHK”)

Table 44: PROGRAM_NEXT_HEK input arguments

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

Table 45: PROGRAM_NEXT_HEK output arguments

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

4.6.2.21 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: 0x4550_484B (“EPHK”)

Table 46: ENABLE_PERMANENT_HEK input arguments

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

Table 47: ENABLE_PERMANENT_HEK output arguments

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

4.6.2.22 REPORT_EPOCH_KEY_STATE

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

Command Code: 0x5245_4B53 (“REKS”)

Table 48: REPORT_EPOCH_KEY_STATE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller.
reserved	u32	Reserved.
sek_state	u16	SEK state. See Table 50.
nonce	u8[16]	Freshness nonce to be included in the signed IETF EAT.

Table 49: REPORT_EPOCH_KEY_STATE output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra.
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 51.
sek_state	u16	SEK state from the input argument.
nonce	u8[16]	Nonce from the input argument.
eat_len	u16	Total length of the IETF EAT.
eat	u8[eat_len]	CBOR-encoded and signed IETF EAT. See Section B for the format.

Table 50: SEK state values

Value	Description
0h	SEK_ZEROIZED
1h	SEK_PROGRAMMED
2h to FFFFh	Reserved

Table 51: HEK state values

Value	State	Description
0h	HEK_AVAIL_PREPROD	HEK is available for pre-production testing.
1h	HEK_UNAVAIL_EMPTY	The drive has transitioned to the Production life cycle state and a ratchet secret has not yet been provisioned. The PROGRAM_NEXT_HEK command can be used to provision the first secret.
2h	HEK_AVAIL_PROD	A randomized ratchet secret is provisioned in the current slot. The ZEROIZE_CURRENT_HEK command can be used to zeroized the current slot.
3h	HEK_UNAVAIL_ZEROIZED	The current ratchet secret has been zeroized. The PROGRAM_NEXT_HEK command can be used to provision a new rachet secret, if there remains at least one available blank ratchet secret slot. If all slots have been zeroized, the ENABLE_PERMANENT_HEK command can be used to make the HEK available.
4h	HEK_UNAVAIL_CORRUPTED	The current ratchet secret slot is corrupted. The ZEROIZE_CURRENT_HEK command can be used to zeroized the current slot.
5h	HEK_AVAIL_PERMANENT	All ratchet secrets have been zeroized and permanent HEK mode has been enabled. Owners may opt into this state once every ratchet secret has been zeroized. In this state, data written to the storage device can no longer be cryptographically erased via ratchet secret zeroization.
6h to FFFFh	Reserved	Reserved.

Operations that use the HEK are disallowed in all `HEK_UNAVAIL_*` states.

See Table 1 for more information about the relationship between HEKs and Caliptra lifecycle states.

HEK_AVAIL_PREPROD → HEK_UNAVAIL_EMPTY is a one-way transition that the device vendor is expected to trigger by advancing the Caliptra lifecycle state to Production.

HEK_UNAVAIL_ZEROIZED → HEK_AVAIL_PERMANENT is a one-way transition that the device owner can trigger using the ENABLE_PERMANENT_HEK command once all ratchet secret slots have been zeroized.

See Figure 15 for more details about the transitions between HEK states.

4.6.2.23 Common mailbox types

This section defines common types used to interface between the drive firmware and KMB. These types are common patterns found in both requests and responses.

Table 52: SealedAccessKey contents

Name	Type	Description
kem_handle	u32	Handle for KEM keypair held in KMB memory.
hpke_algorithm	u32	HPKE algorithm. Must be a bit value indicated as supported in Table 14.
access_key_len	u32	Access key length in bytes. Must be the scalar value associated with a bit value indicated as supported in Table 14.
kem_ciphertext	u8[Nenc]	HPKE encapsulated key.
ak_ct	u8[access_key_len+Nt]	Access key ciphertext and authentication tag.

`Nenc` and `Nt` are HPKE values associated with the `kem_id` and `aead_id` identifiers from the given `hpke_algorithm`. For example, if byte 0 bit 0 of `hpke_algorithm` is set (indicating `kem_id` 0x0011 and `aead_id` 0x0002), then according to [6], `Nenc` and `Nt` would be 97 and 16, respectively.

Table 53: SealedCurrentAndNewAccessKey contents

Name	Type	Description
current_access_key	SealedAccessKey	Current access key.
new_access_key	u8[access_key_len+Nt]	New access key.

`Nt` is the HPKE value associated with the `aead_id` identifier from the SealedAccessKey's `hpke_algorithm`.

4.6.2.23.1 WrappedKey type

AES-256-GCM is used for all wrapping and unwrapping.

Table 54: WrappedKey contents

Name	Type	Description
key_type	u16	Type of the wrapped key. • 0h: Reserved • 1h: LOCKED_MPK (ciphertext held at rest) • 2h: READY_MPK (ciphertext held in RAM) • 3h: WRAPPED_MEK • 4h to FFFFh: Reserved
reserved	u16	Reserved.
id	u64	Random identifier for the given wrapped key.
metadata_len	u32	Length of the metadata field.
ct_len	u32	Length of the ct field.
iv	u8[12]	Initialization vector for AES operation.
metadata	u8[metadata_len]	Metadata associated with the wrapped key.
ct	u8[ct_len]	Key ciphertext and authentication tag.

The AAD for the encrypted message is constructed as `key_type` || `id` || `metadata_len` || `metadata`.

Variants of WrappedKey will be used to reduce duplicating information in commands. The following names will be used for WrappedKeys of a specific `key_type` and `ct_len`:

Table 55: WrappedKey variants

Name	`key_type`	`ct_len`
LockedMpk	LOCKED_MPK	32
ReadyMpk	READY_MPK	32
WrappedMek	WRAPPED_MEK	64

4.6.2.24 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, drive firmware may resubmit the mailbox command.

Table 56: KMB mailbox command result codes

Name	Value	Description
LOCK_ENGINE_TIMEOUT	0x4C45_544F (“LETO”)	Timeout occurred when communicating with the drive encryption 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
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	0x4C48_4953 (“LHIS”)	Incorrect HEK slot when programming or zeroizing
LOCK_EE_NOT_READY	0x4C45_4E52 (“LENR”)	Encryption engine was not ready when the command was received.
LOCK_HEK_NOT_AVAILABLE	0x4C48_4E41 (“LHNA”)	The operation requires the HEK, which is unavailable.
LOCK_HEK_NOT_ZEROIZED	0x4C48_4E5A (“LHNZ”)	Next HEK slot cannot be programmed because the current slot is programmed or corrupted, and must be zeroized.
LOCK_HEK_ZEROIZED	0x4C48_5A44 (“LHZD”)	Current HEK slot cannot be zeroized because it is already zeroized.
LOCK_HEK_SLOTS_FULL	0x4C48_5346 (“LHSF”)	Next HEK slot cannot be programmed because all slots are exhausted.

(continued on next page)

(continued from previous page)

Name	Value	Description
LOCK_HEKS_UNZEROIZED	0x4C48_555A (“LHUZ”)	Permanent-HEK mode cannot be enabled because one or more HEK slots are not zeroized.

4.6.2.24.1 Fatal errors

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

4.6.2.24.2 Non-fatal errors

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

4.7 Terminology

Table 57: Acronyms and abbreviations used throughout this document

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
EPK	Epoch Protection Key
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

(continued on next page)

(continued from previous page)

Abbreviation	Description
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
SVN	Security Version Number
TCG	Trusted Computing Group
UART	Universal asynchronous receiver-transmitter
XTS	XEX-based tweaked-codebook mode with ciphertext stealing

4.8 Compliance

This section enumerates requirements for devices that integrate OCP L.O.C.K.

Table 58: Compliance requirements

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. See Section 4.5.	Yes
4	The encryption engine shall remove all MEKs from the encryption engine on a power cycle or during zeroization of the storage device. See Section 4.5.	Yes
5	The SEK shall only be programmed once the HEK is available, and the HEK shall only be zeroized once the SEK is zeroized. See Section 4.5.5.3.	Yes
6	MEKs shall not be programmed while the SEK is zeroized. See Section 4.5.5.	Yes
7	Drive firmware shall implement authorization controls to gate lifecycle events that have the potential to trigger data loss. See Section 4.5.9.	Yes
8	The encryption engine shall ensure that AES-XTS Key_1 and Key_2 are not equal. See Section 4.5.6.3.	Yes

4.9 Repository location

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

Appendix A: Preconditioned AES-Encrypt calculations

This appendix expands on Section 4.5.2.2 and provides additional details behind the claim that approximately 2^{64} preconditioned AES-Encrypt operations for a given secret are needed before an IV collision is expected to occur with probability greater than 2^{-32}.

To satisfy FIPS, it is sufficient to demonstrate that no single AES-GCM key will be used more than 2^{32} times. This can be put in terms of the “Balls into bins” problem [17], where we are looking for the number of balls (m) required for the maximum load across 2^{64} bins (n) to be 2^{32}. An approximation for the maximum load where $m > n$ is given as $m/n + \sqrt{m \cdot log(n)/n}$. Rearranging to solve for m via the quadratic formula yields $m \approx 2^{96}$. Therefore a given input secret may be used in at most 2^{96} preconditioned AES-Encrypt operations before any key derived from that secret is used in more than 2^{32} AES-GCM-Encrypt operations.

However, the 2^{32} encryption limit is a means to an end, namely ensuring a low probability of IV collisions. The Birthday paradox [18] implies that the probability of a collision between n generated IVs where d is the number of possible IVs is approximately $n^2/(2d)$. The chances of a 96-bit IV collision across 2^{32} IVs is therefore approximately $2^{2 \cdot 32}/(2^{96+1}) = 2^{-33}$ for a given key. Across 2^{64} keys derived from an input secret, the expected number of keys that experience a collision is approximately $2^{64} \cdot 2^{-33} = 2^{31}$. Clearly a limit of 2^{96} encryption operations for a given input secret, while meeting the letter of the FIPS requirements, is too large for safety.

We can calculate the safe margin by simply considering the 64-bit ID concatenated to the 96-bit IV. What we are really after is the maximum number of preconditioned AES-Encrypt invocations with a given input secret such that the likelihood of experiencing a collision of the 160-bit (ID || IV) pair is at most 2^{-32}. Leveraging the Birthday paradox equation, $n \approx \sqrt{2^{160+1} \cdot 2^{-32}} = 2^{64.5}$.

Therefore a given input secret may safely be used in at most $2^{64.5}$ preconditioned AES-Encrypt operations before an IV collision is expected to occur with probability greater than 2^{-32} across all of the AES-GCM keys derived from the input secret.

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

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

Appendix C: Sequence diagrams

C.1 Sequence of events at boot

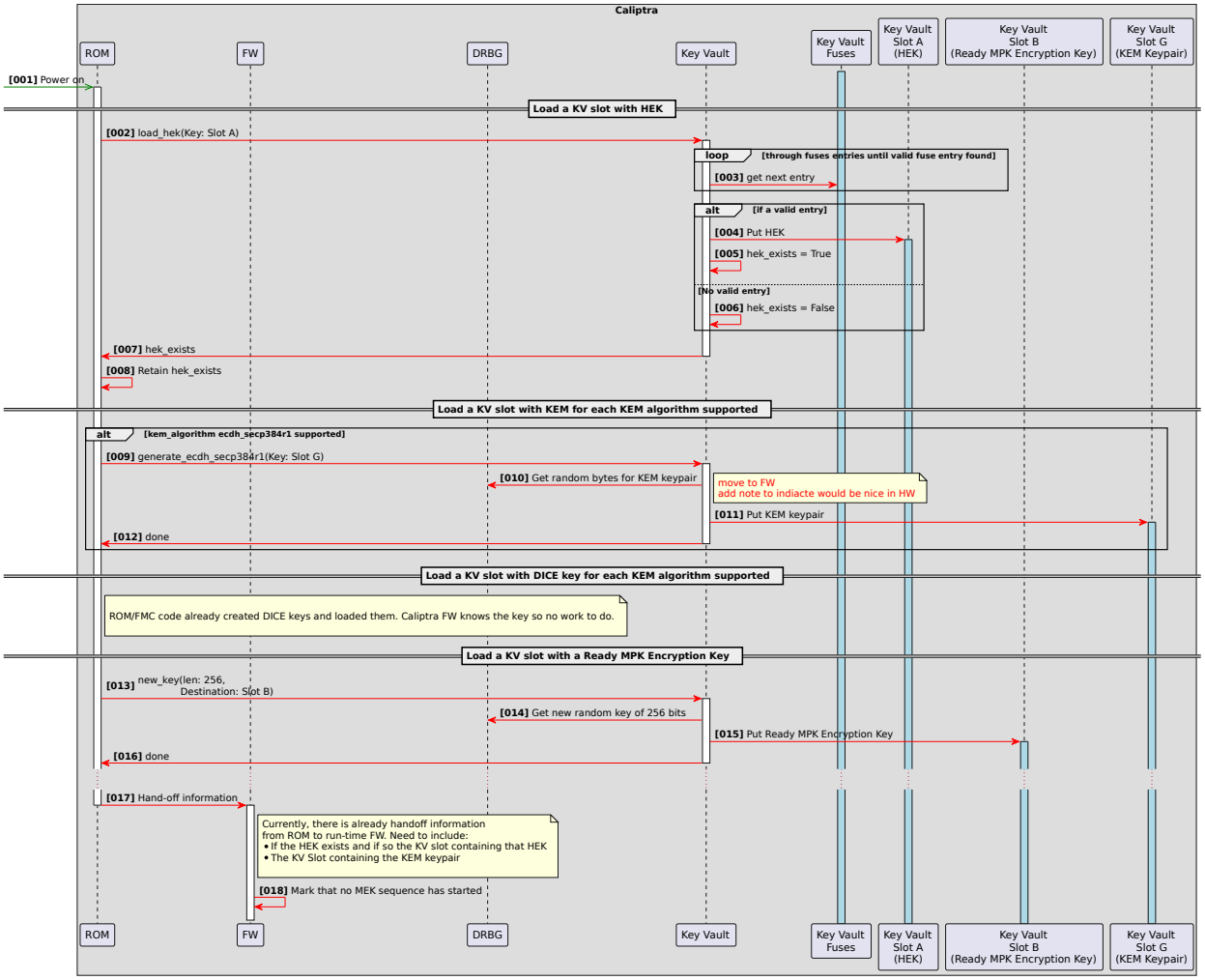

Figure 27: UML: Power on

C.2 Sequence to obtain the current status of KMB

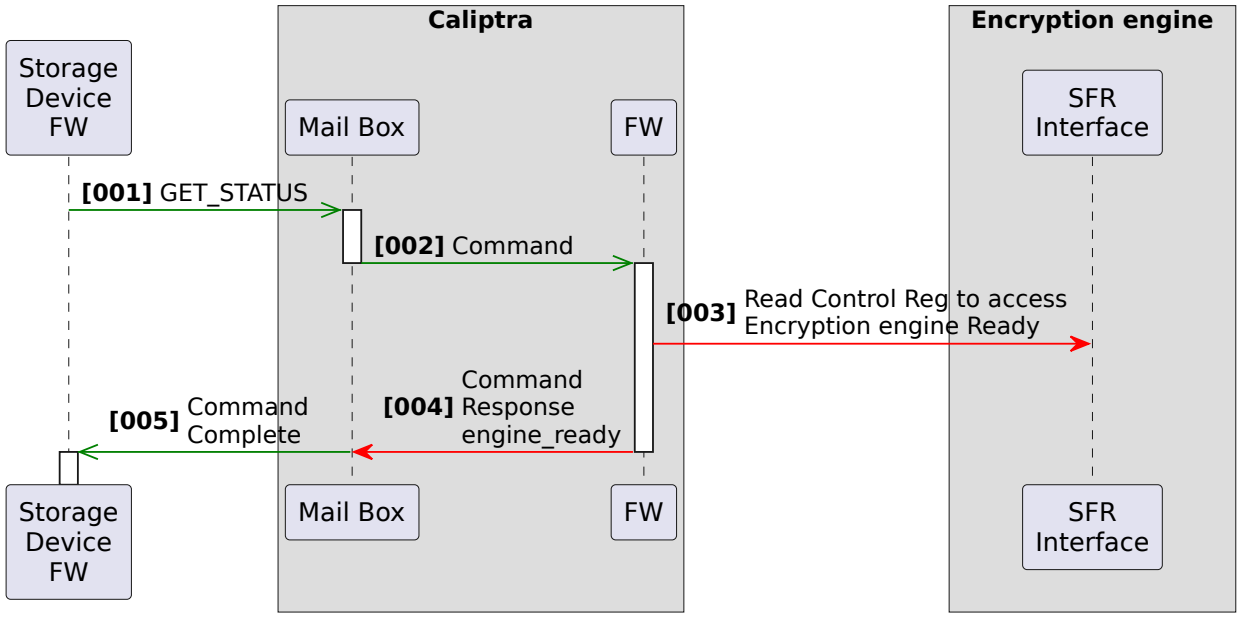

Figure 28: UML: Get Status

C.3 Sequence to obtain the supported algorithms

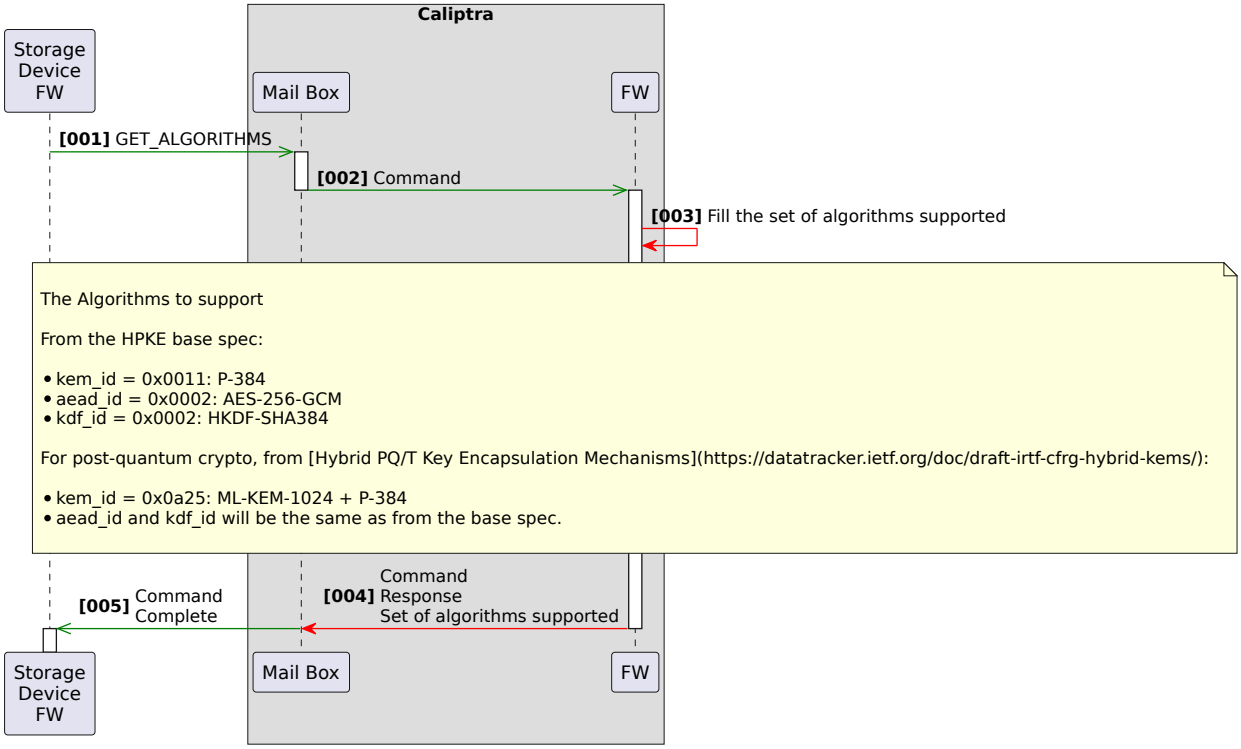

Figure 29: UML: Get Supported Algorithms

C.4 Sequence to endorse an HPKE public key

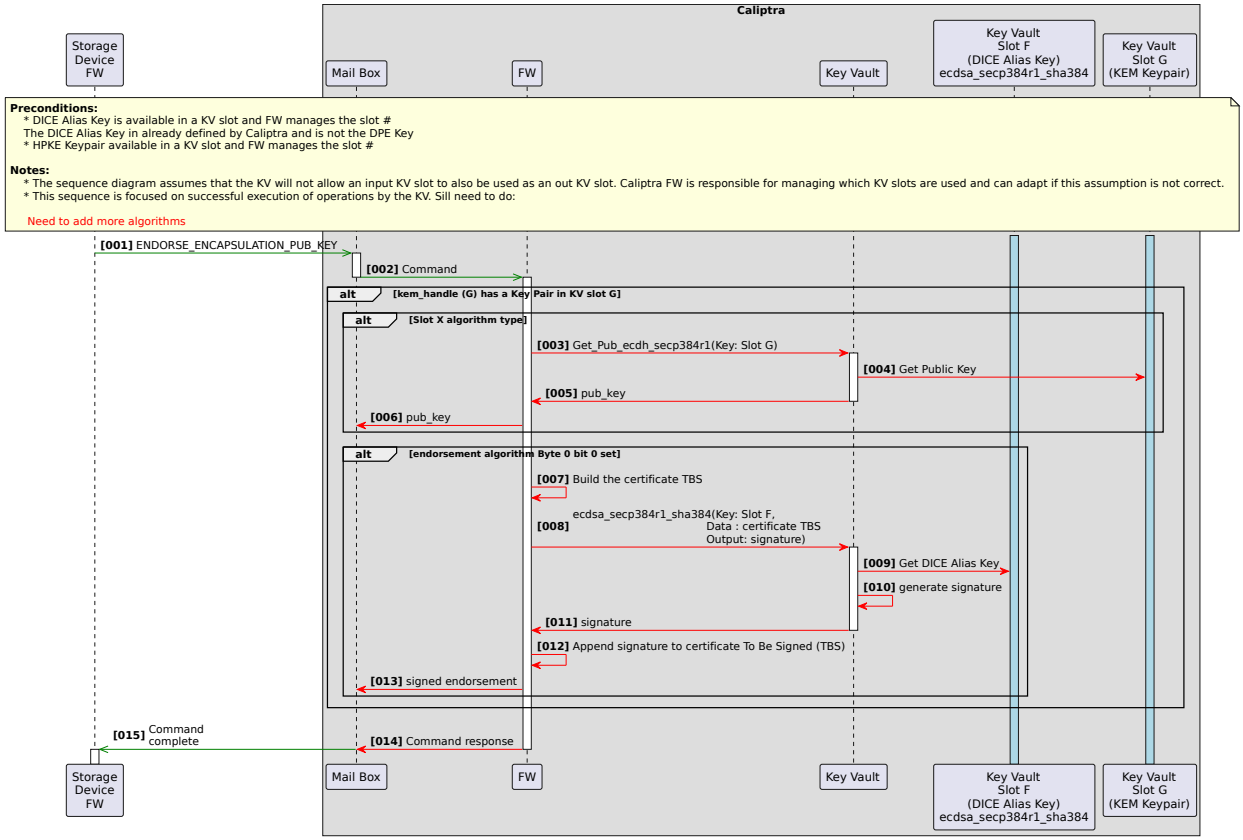

Figure 30: UML: Endorsing an HPKE public key

C.5 Sequence to rotate an HPKE keypair

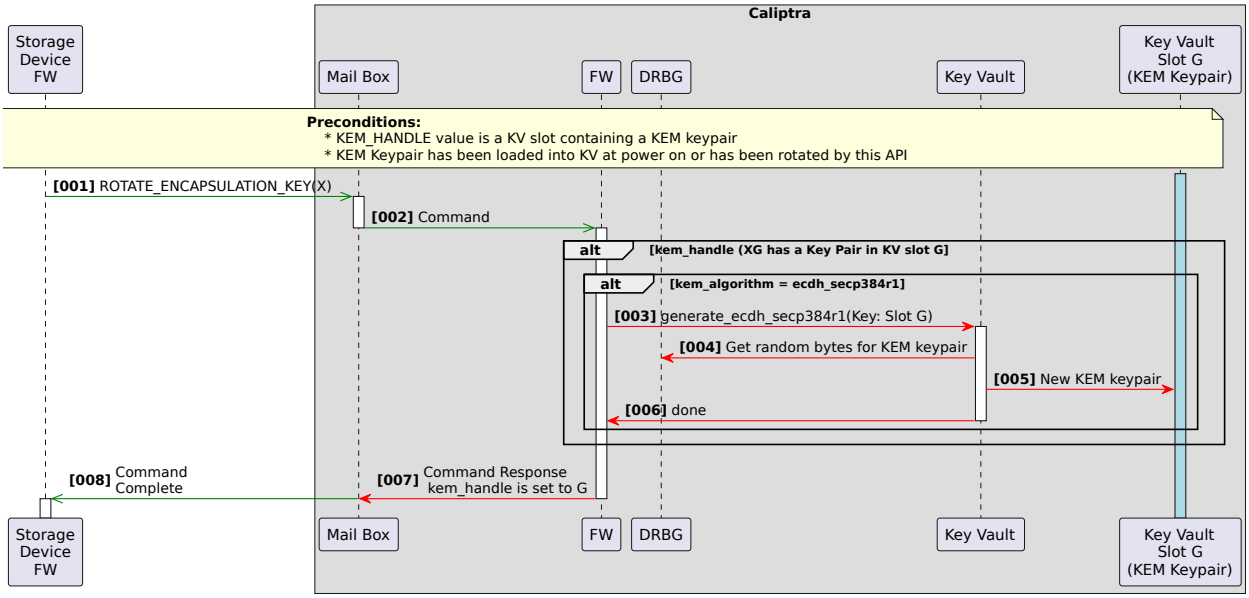

Figure 31: UML: Rotating a KEM Encapsulation Key

C.6 Sequence to generate an MPK

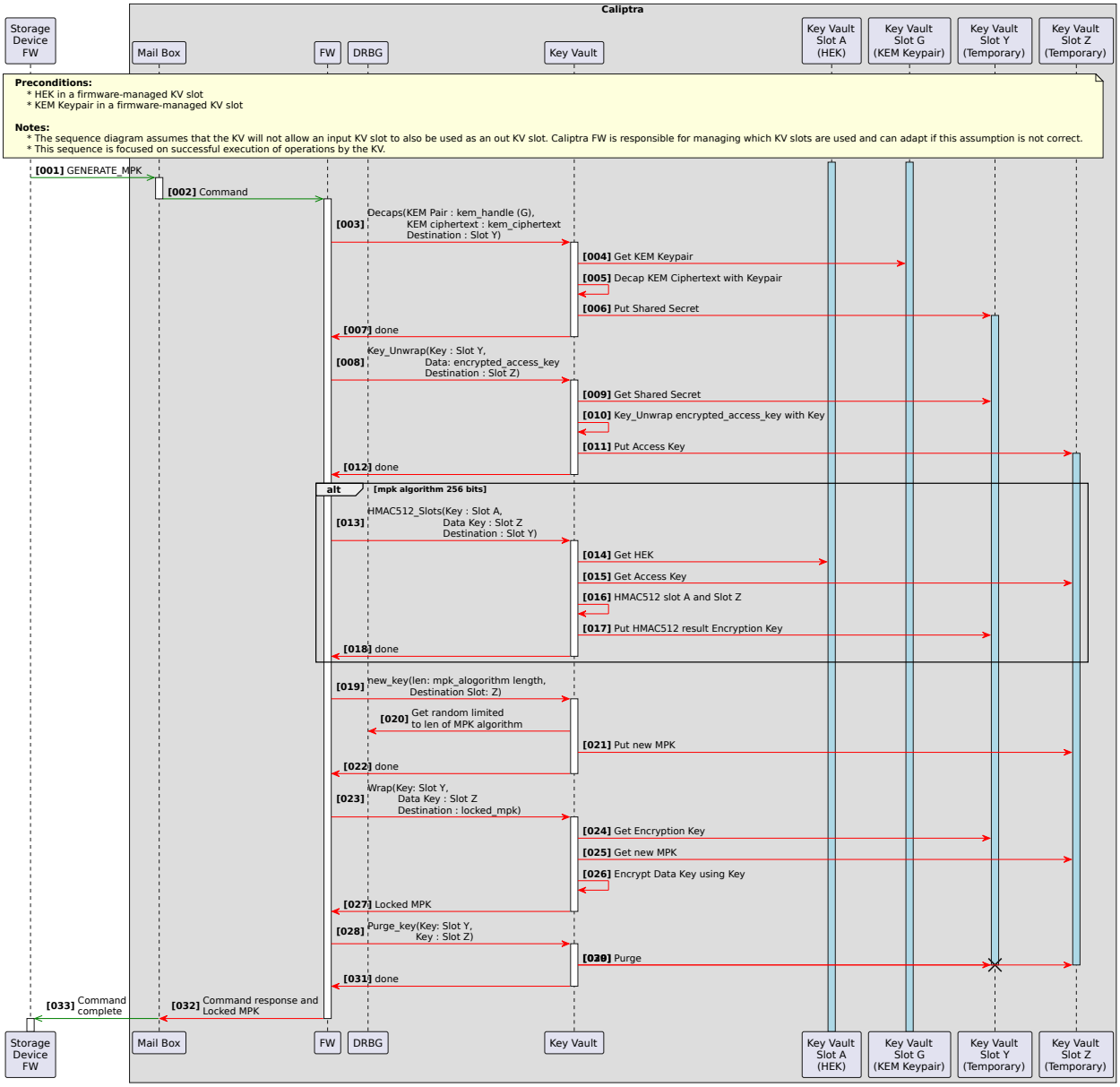

Figure 32: UML: Generating an MPK

C.7 Sequence to ready an MPK

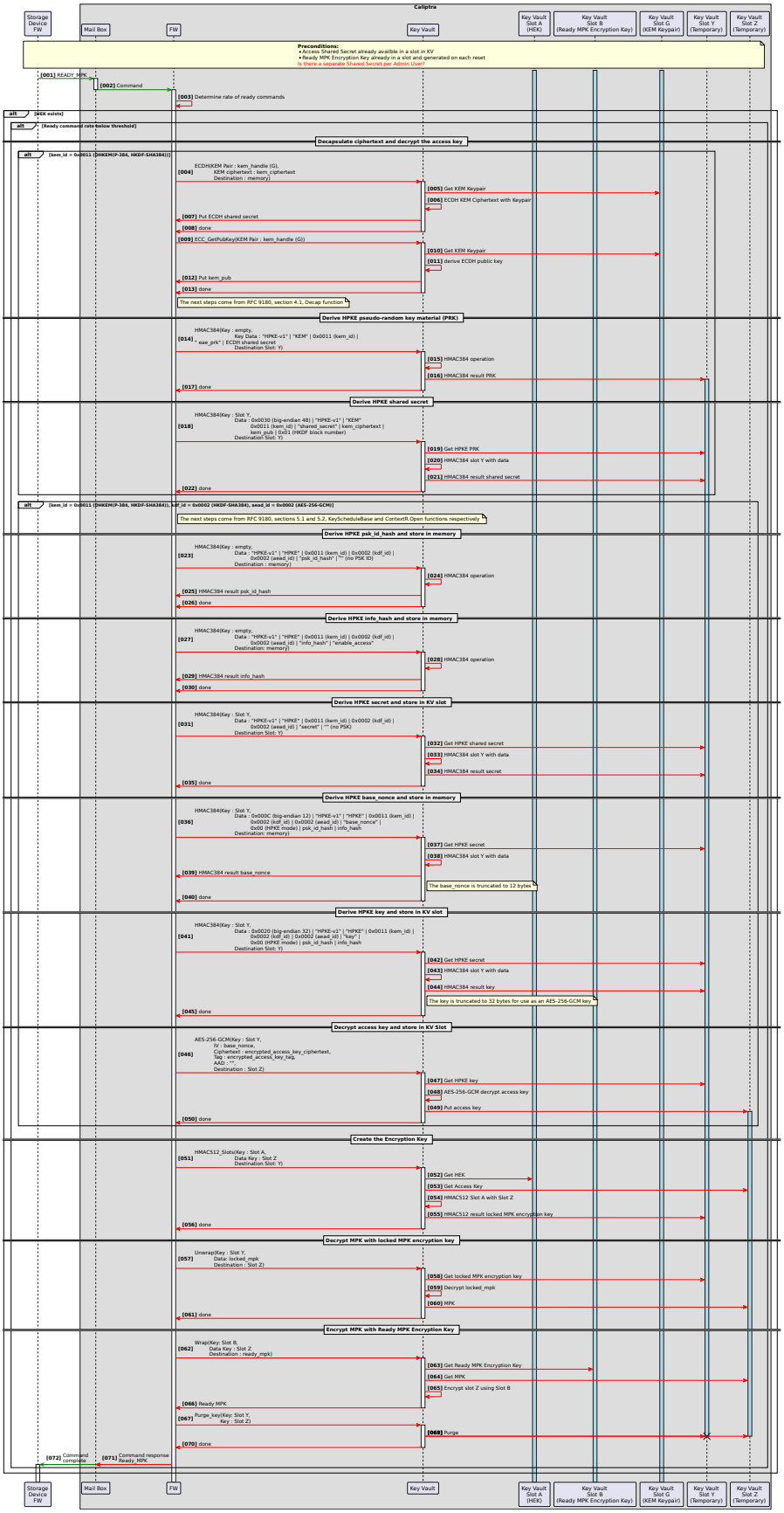

Figure 33: UML: Readying an MPK

C.8 Sequence to rotate the access key of an MPK

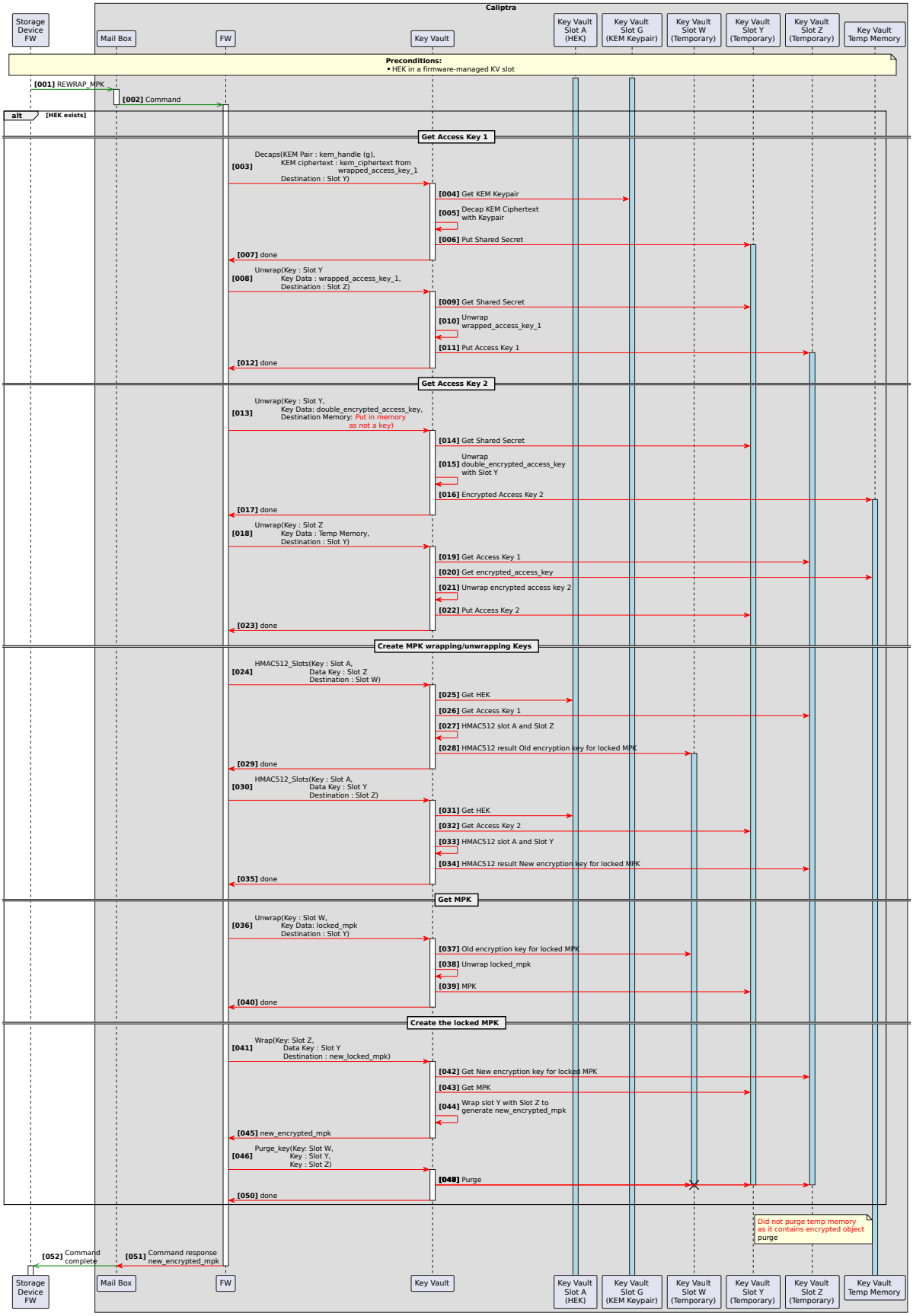

Figure 34: UML: Rewrapping an MPK

C.9 Sequence to mix an MPK into the MEK secret seed

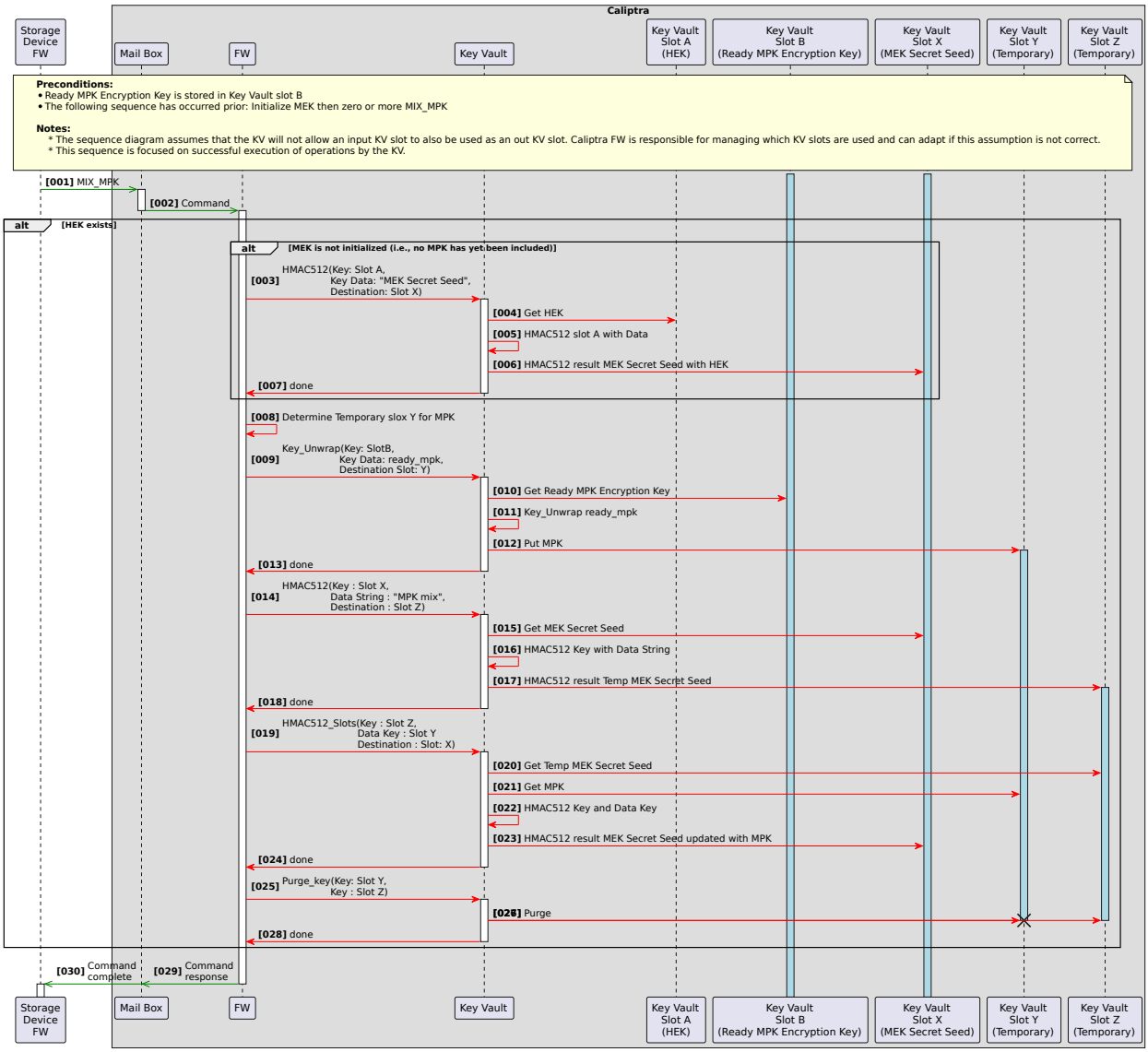

Figure 35: UML: Mixing an MPK

C.10 Sequence to test the access key of an MPK

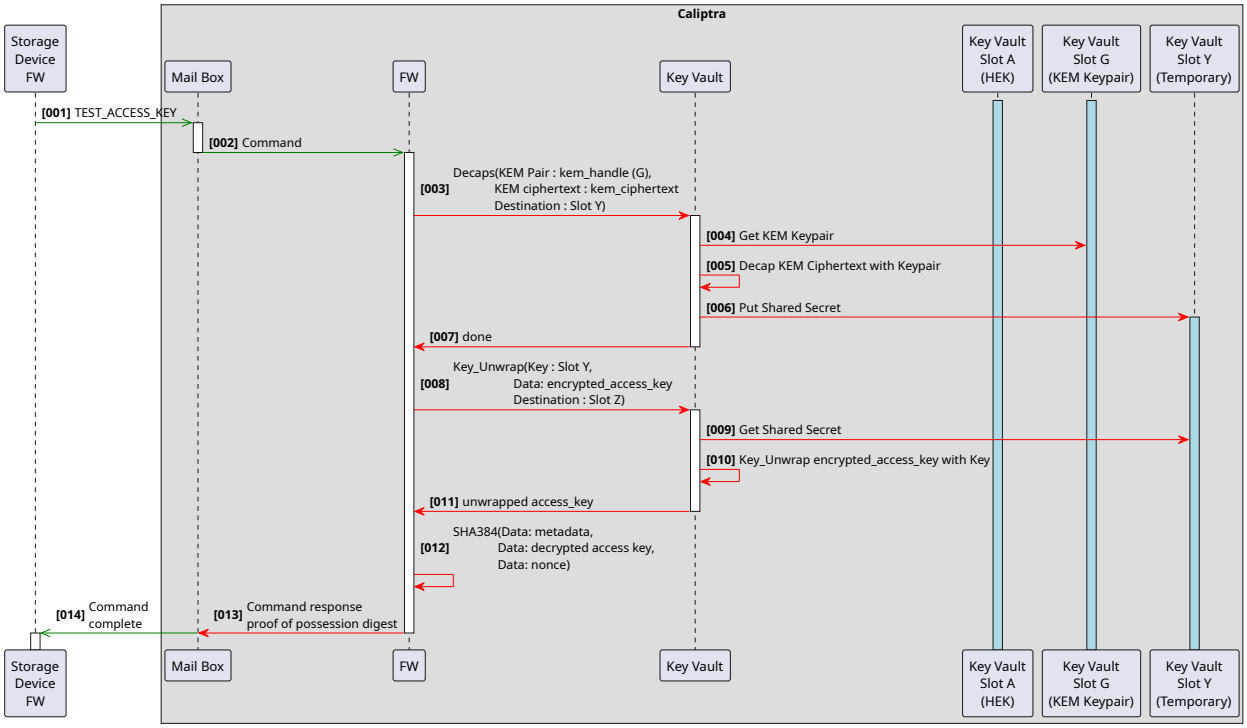

Figure 36: UML: Testing an Access Key

C.11 Sequence to load an MEK

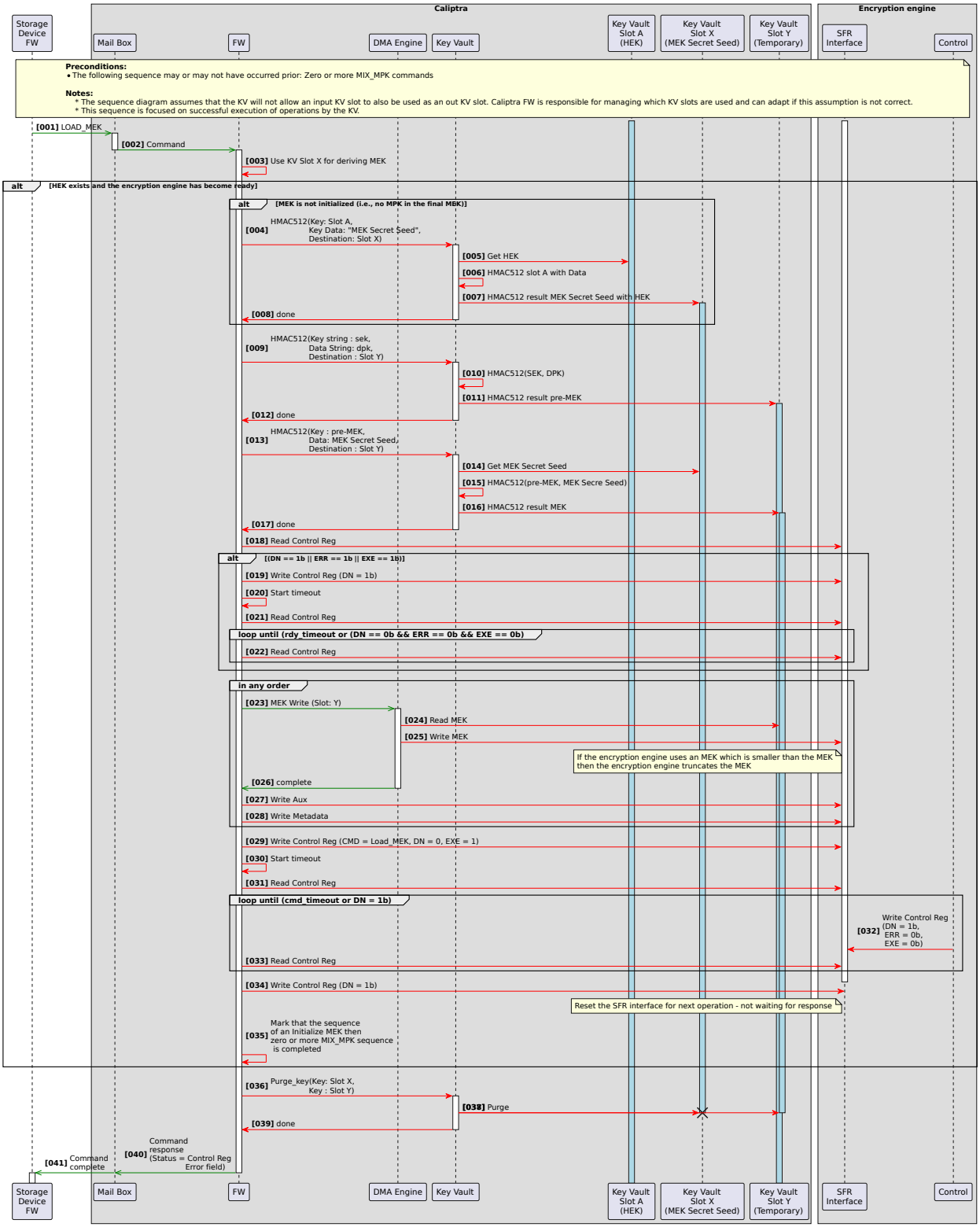

Figure 37: UML: Loading an MEK

C.12 Sequence to load MEK into the encryption engine key cache

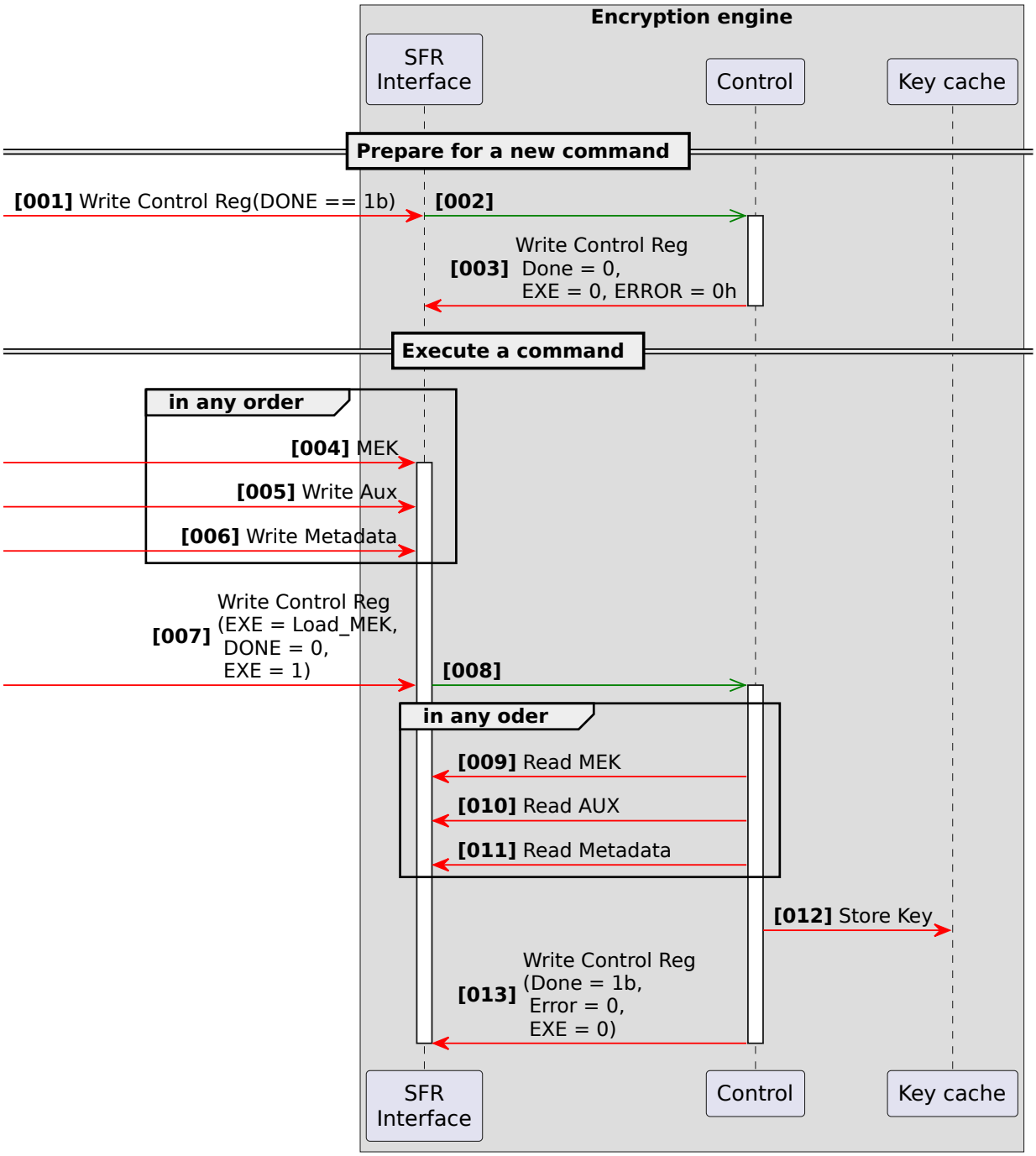

Figure 38: UML: Loading an MEK into the encryption engine key cache

C.13 Sequence to unload an MEK from the encryption engine key cache

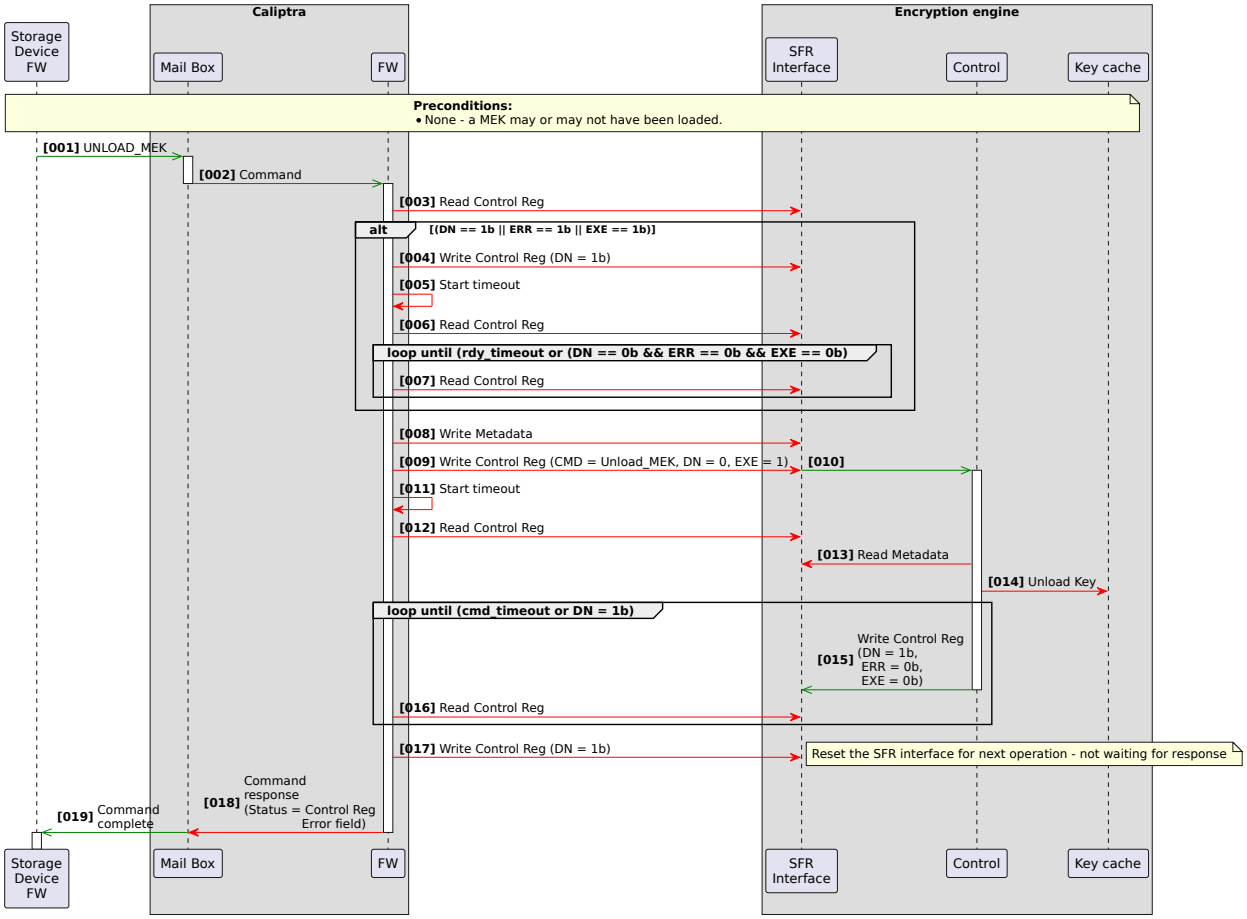

Figure 39: UML: Unloading an MEK

C.14 Sequence to unload all MEKs (i.e., zeroize) from the encryption engine key cache

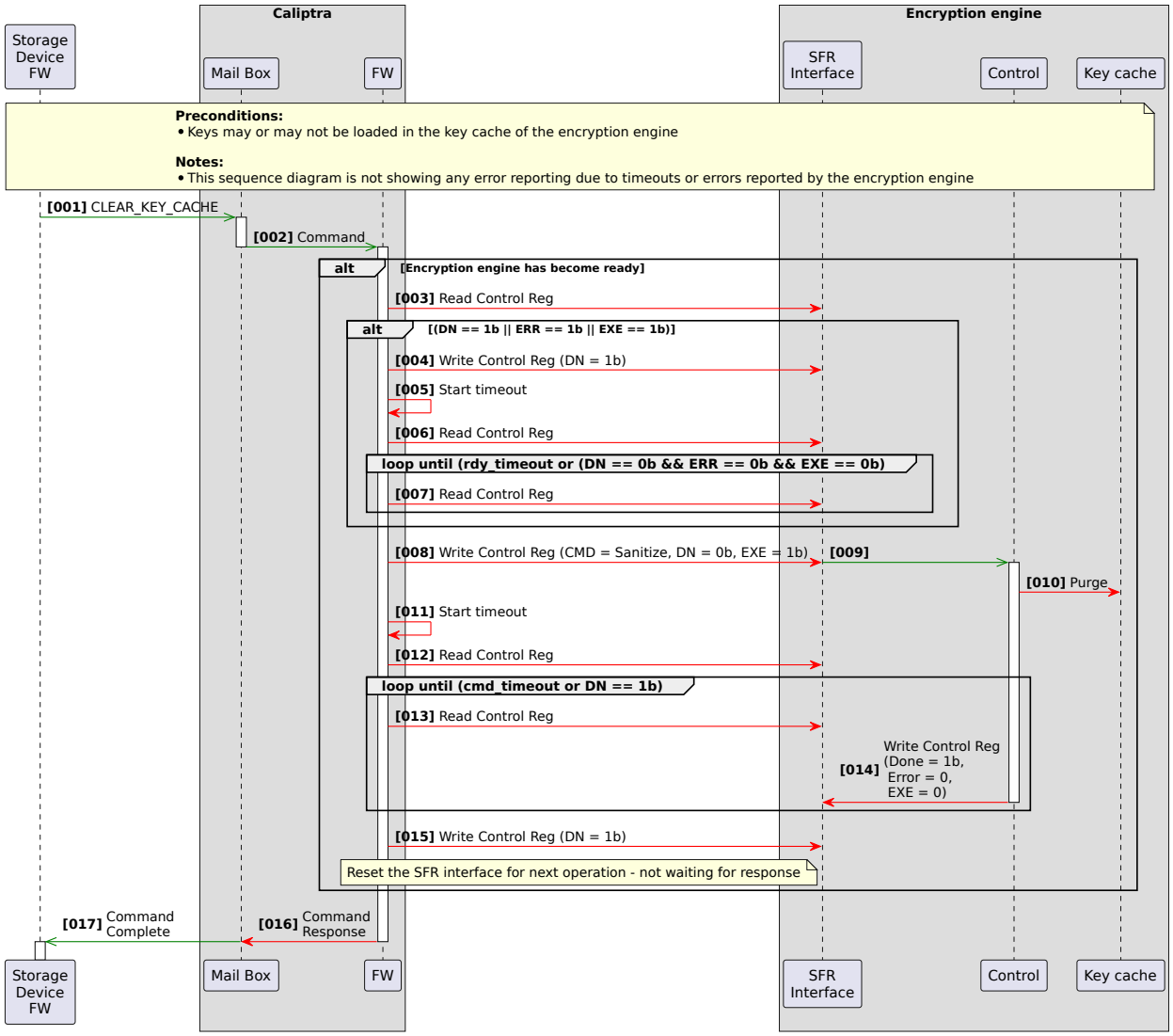

Figure 40: UML: Unloading all MEKs

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] “IEEE Draft Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices.” IEEE, Feb. 2025. Available: https://standards.ieee.org/ieee/1619/11552/

[5] “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

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

[7] “Recommendation for Cryptographic Key Generation (Rev. 2).” NIST, Jun. 2020. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-133r2.pdf

[8] “HMAC-based Extract-and-Expand Key Derivation Function (HKDF).” IETF, May 2010. Available: https://datatracker.ietf.org/doc/html/rfc5869

[9] “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.” NIST, Nov. 2007. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

[10] “TCG DICE.” Trusted Computing Group. Available: https://trustedcomputinggroup.org/work-groups/dice-architectures/

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

[12] “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

[13] “Recommendation for Key Derivation Using Pseudorandom Functions (Rev. 1).” NIST, Aug. 2022. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-108r1-upd1.pdf

[14] “Caliptra 2.0 RTL Specification.” CHIPS Alliance. Available: https://github.com/chipsalliance/caliptra-rtl/blob/ea416cb/docs/CaliptraHardwareSpecification.md

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

[16] 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

[17] “Balls into bins problem.” Wikipedia. Accessed: Jun. 2025. [Online]. Available: https://en.wikipedia.org/wiki/Balls_into_bins_problem

[18] “Birthday problem.” Wikipedia. Accessed: Jun. 2025. [Online]. Available: https://en.wikipedia.org/wiki/Birthday_problem