OPEN
Compute Project

NVMe™ Project

# OCP L.O.C.K.

1.0 (RC2)

**Modular Base Specification**
Effective September 3, 2025

Author: (See Acknowledgements section)

## Current Template Version:

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

# Table of Contents

# List of Figures

# List of Tables

# 1 License

## 1.1 Open Web Foundation (OWF) CLA

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

- Google
- Microsoft
- Samsung
- Kioxia
- Solidigm

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

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

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

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)
- Carl Lundin (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 providers 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 cloud providers, 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) 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™. OCP L.O.C.K. is designed for compatibility with the TCG Storage Architecture [1], including support for TCG Opal[1] [2], TCG Enterprise [3], and Key Per I/O [4] specifications.

## 4.2 Background

In the life of a storage device in a datacenter, the device leaves the supplier, a customer writes user data to the device, and then the device is decommissioned. Customer data is not allowed to leave the data center. The cloud provider needs high confidence that the storage device leaving the datacenter is secure. The current default cloud provider 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 cloud providers [5]. This produces significant e-waste and inhibits any re-use/recycling.

Self-encrypting drives (SEDs) store data encrypted at rest using 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 and Enterprise specifications.

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.

[1] 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.

- Theft of a storage device from a data center.
- Destructively inspecting a stolen device.
- Running arbitrary firmware on a stolen device.
  - This includes attacks where vendor firmware signing keys have been compromised.
- Attempting to glitch execution of code running on general-purpose cores.
- Stealing debug core dumps or UART/serial logs from a device while it is operating in a data center, and later stealing the device.
- Gaining access to any class secrets, global secrets, or symmetric secrets shared between the device and an external entity.
- Executing code within a virtual machine on a multi-tenant host offered by the cloud provider 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 integrates 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.

### 4.4.1 Non-goals

The following areas are out of scope for this project.

- Compliance with IEEE 1619$^{TM}$-2025 [6] 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) [7] 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 integrates 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.
- Mitigation against availability attacks carried out by attackers with physical presence.

## 4.5 Integrating OCP L.O.C.K.

OCP L.O.C.K. is a feature set conditionally compiled into Caliptra Subsystem 2.1+. It consists of functionality in Caliptra ROM and Runtime Firmware along with a hardware interface from Caliptra Core to a vendor-implemented encryption engine, which is the path by which Caliptra Core programs MEKs. This hardware interface leverages the AXI Manager which is only available in Caliptra Subsystem.

OCP L.O.C.K.'s KMB offers cryptographic services for drive firmware, which implements a host-facing storage interface, such as TCG Opal, Enterprise, or Key Per I/O. The host will interact with drive firmware to configure the storage device and actuate storage lifecycle events, such as MEK loading / unloading and cryptographic erase. Certain of these lifecycle events are handled by KMB on behalf of drive firmware.

In Caliptra Subsystem, drive firmware runs within the Manufacturer Control Unit (MCU) [8].

### 4.5.1 Integration verification

A product that integrates OCP L.O.C.K. will be expected to undergo an OCP S.A.F.E. review to ensure that L.O.C.K. is correctly integrated and that the drive firmware correctly invokes L.O.C.K. services.

## 4.6 Architecture

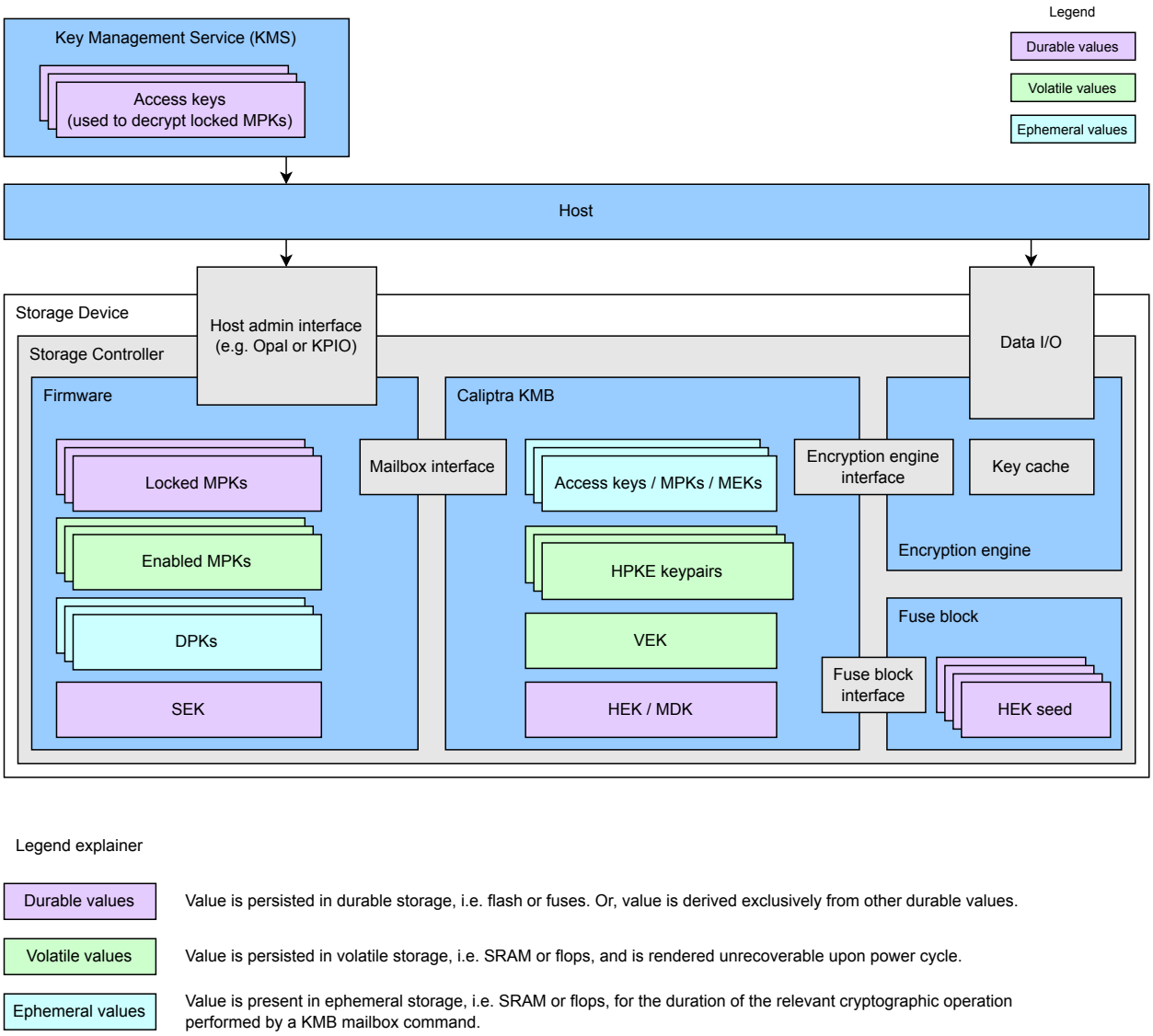



**Figure 1:** OCP L.O.C.K. high level blocks

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 or Enterprise 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 [9]. 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, Enterprise, or Key Per I/O support.

- A composite **epoch protection key (EPK)**, which is the output of a KDF 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 derived from a seed held in fuses and is only visible to KMB hardware. This provides assurance that an advanced adversary is unable to recover key material that had been in use by the drive prior to HEK seed 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 two keys: the MEK secret and the **MEK Deobfuscation Key (MDK)**, and is allowed to load that wrapped MEK into the encryption engine via KMB. This supports Opal or Enterprise 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, Enterprise, 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 4.7. 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.

### 4.6.1 Key hierarchy

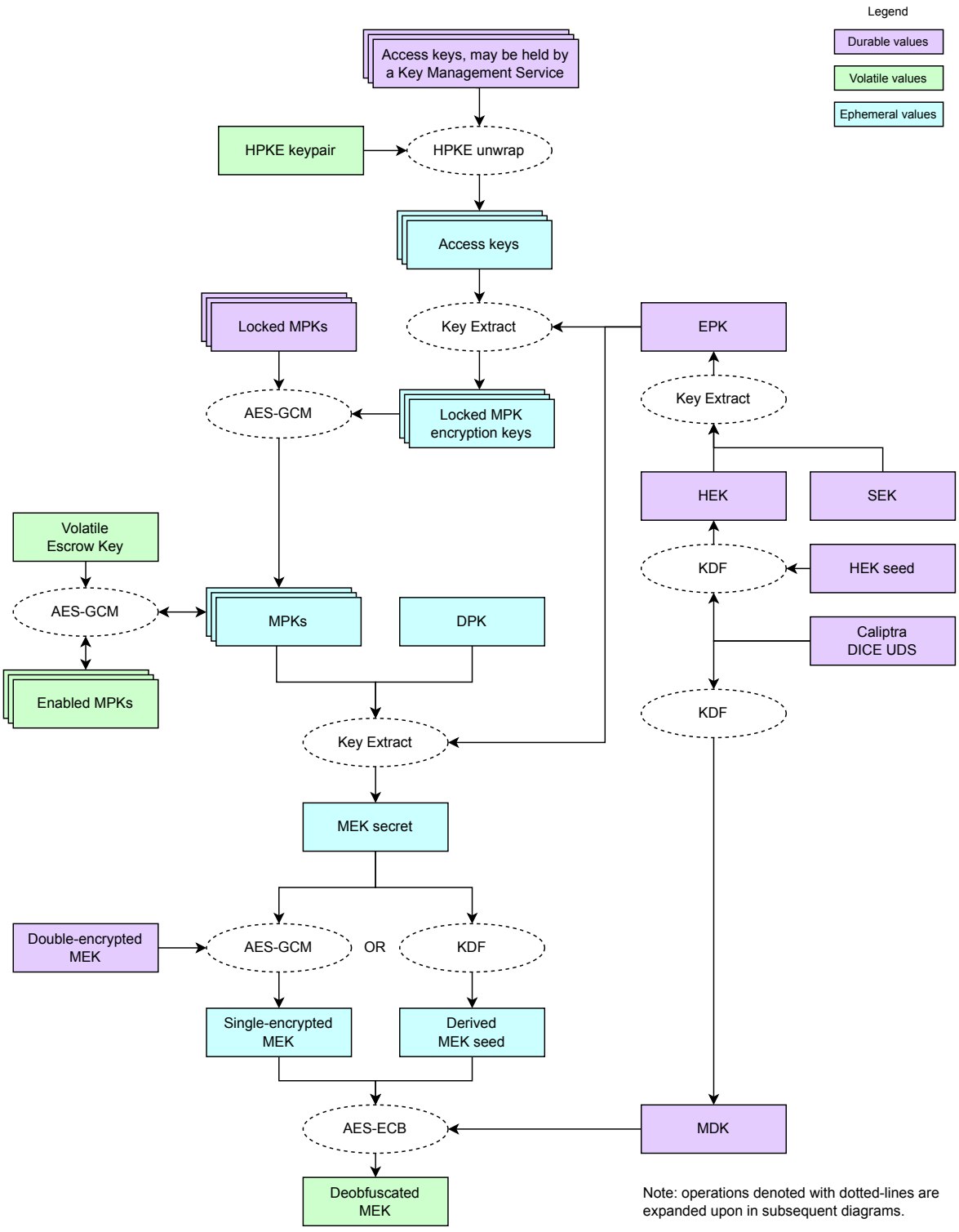



**Figure 2:** OCP L.O.C.K. key hierarchy

Table 1 enumerates the CSPs (Critical Security Parameters) used by KMB, along with where each CSP is held in the Key Vault (KV). KV slots 16-23 are reserved for OCP L.O.C.K.

**Table 1:** Critical Security Parameters

| CSP | Usage | Lifecycle | KV slot number |
|---|---|---|---|
| UDS | Used to derive the DICE IDevID CDI, which in turn is used to derive the MDK and HEK. See Figures 22 and 18. | Held in the Key Vault during cold reset and zeroized after use.<br>Rendered irrecoverable if zeroized from persistent storage. | 0 |
| MDK | Used to perform the inner encryption and decryption of each MEK. See Section 4.6.5.2. | Derived from the UDS during cold reset. Held in the Key Vault and lost on cold reset.<br>Rendered irrecoverable if the DICE UDS is zeroized from persistent storage. | 16 |
| HEK seed | Contributes to the derivation of the HEK. See Section 4.6.4.1. | Randomly generated and stored in fuses.<br>Drive firmware zeroizes the HEK seed from fuses on demand, according to the constraints given in Section 4.6.4.2. | N/A |
| HEK | Contributes to the derivation of the EPK. See Figure 12 for one of several flows where this is done. | Derived from the HEK seed and UDS during cold reset. Held in the Key Vault and lost on cold reset.<br>Rendered irrecoverable if the HEK seed or the DICE UDS are zeroized from persistent storage. | 22 |
| SEK | Contributes to the derivation of the EPK. See Figure 12 for one of several flows where this is done. | Managed outside of Caliptra. Randomly generated by drive firmware and stored in flash. When held by Caliptra, the SEK is held in volatile memory and is zeroized after each use.<br>Drive firmware zeroizes the SEK from flash on demand, according to the constraints given in Section 4.6.4.2. | N/A |
| EPK | Contributes to the derivation of each Locked MPK encryption key. See Figure 12 for one of several flows where this is done.<br>Contributes to the derivation of each MEK secret. See Figure 20. | Derived from the HEK and SEK. Held in the Key Vault and zeroized after each use.<br>Rendered irrecoverable if the SEK, HEK seed, or DICE UDS are zeroized from persistent storage. | 17 |

*(continued on next page)*

*(continued from previous page)*

| CSP | Usage | Lifecycle | KV slot number |
|---|---|---|---|
| Volatile Escrow Key (VEK) | Allows KMB firmware to escrow volatile secrets for future access by runtime KMB firmware.<br>Encrypts/decrypts Enabled MPKs. See Figures 13 and 20. | Randomly generated when first needed at runtime. Held in the Key Vault and lost on cold reset. | 18 |
| HPKE private keys | Provides transport encryption for injected access keys. See Section 4.6.2.1. | Randomly generated on startup. Held in volatile memory, rotated on demand, and lost on cold reset. | N/A |
| Access keys | Contributes to the derivation of a Locked MPK encryption key. See Figure 12 for one of several flows where this is done. | Managed outside of Caliptra. Asymmetrically encrypted using the HPKE public key. When held by Caliptra, each access key is held in volatile memory and is zeroized after each use. | N/A |
| Locked MPK encryption keys | Encrypts/decrypts a Locked MPK. See Figure 12 for one of several flows where this is done. | Derived from the EPK and an injected access key. Held in the Key Vault and zeroized after each use.<br>Rendered irrecoverable if the SEK, HEK seed, or DICE UDS are zeroized from persistent storage. | 19 |
| MPKs | Contributes to the derivation of an MEK secret. See Figure 20. | Randomly generated on demand. Encrypted with a Locked MPK encryption key or the VEK (depending on whether the MPK is locked or enabled). When decrypted, each MPK is held in volatile memory and is zeroized after each use.<br>Rendered irrecoverable if the SEK, HEK seed, or DICE UDS are zeroized from persistent storage. | N/A |
| MPK secrets | Contributes to the derivation of an MEK secret. See Figure 20. | Derived from a sequence of MPKs. When held by Caliptra, the MPK secret is held in the Key Vault and is zeroized after an MEK secret is derived. | 20 |
| DPKs | Contributes to the derivation of an MEK secret. See Figure 20. | Managed outside of Caliptra. When held by Caliptra, the DPK is held in volatile memory and is zeroized after each use. | N/A |

*(continued on next page)*

*(continued from previous page)*

| CSP | Usage | Lifecycle | KV slot number |
|---|---|---|---|
| MEK secrets | Either encrypts/decrypts an MEK or is used to derive an MEK seed. See Sections 4.6.5.2 and 4.6.5.3. | Derived from the EPK, DPK, and zero or more decrypted MPKs. Held in the Key Vault and zeroized after each use.<br>Rendered irrecoverable if the SEK, HEK seed, or DICE UDS are zeroized from persistent storage. | 21 |
| MEK seeds | Used to derive an MEK. See Section 4.6.5.3. | Derived from an MEK secret. Held by firmware and zeroized after each use.<br>Rendered irrecoverable if the SEK, HEK seed, or DICE UDS are zeroized from persistent storage. | N/A |
| MEKs | Encrypts/decrypts user data written to the storage device. See Section 4.6.5. | Either randomly generated on demand and encrypted at rest with the MEK secret and MDK, or derived from the MEK secret. Held in the Key Vault and zeroized after programming to the Encryption Engine.<br>Rendered irrecoverable if the SEK, HEK seed, or DICE UDS are zeroized from persistent storage. | 23[1] |

#### 4.6.1.1 Key Vault

Some elements of the OCP L.O.C.K. key hierarchy are held in the Caliptra Key Vault. Such keys are wielded directly by hardware and cannot be read by firmware.

Other elements of the key hierarchy are not thus protected, and are held in KMB's memory instead. Table 2 describes why each such CSP is held outside of the Key Vault.

**Table 2:** CSPs visible to KMB firmware

| Firmware-visible CSP | Reason for being visible to KMB firmware |
|---|---|
| SEK | Managed by drive firmware, passed in via mailbox command. |
| DPKs | Managed by drive firmware, passed in via mailbox command. |

*(continued on next page)*

[1] Randomly-generated MEKs are held in memory rather than the Key Vault when they are first generated. See Table 2 for more details.

*(continued from previous page)*

| Firmware-visible CSP | Reason for being visible to KMB firmware |
|---|---|
| HPKE private keys | To mitigate side-channel attacks, the ECDH and ML-KEM engines mandate that any operation whose source includes a key from the Key Vault must place the result back in the Key Vault. If the HPKE private keys were held in the Key Vault, then the ECDH and ML-KEM Decaps results would be held in the Key Vault as well. Section 4.6.2.1.1 illustrates that these results are used to compute an HMAC message, a SHA3 digest, and an AES-GCM IV. Caliptra's Key vault hardware does not presently support these operations. Therefore the ECDH and ML-KEM Decaps results cannot be held in the Key Vault; ergo, the HPKE private keys also cannot be held in the Key Vault. |
| Access keys | Each access key is encrypted using an HPKE private key. Since firmware manages the HPKE private keys, it is not meaningful to decrypt the access key into the Key Vault. |
| Decrypted MPKs | MPKs are encrypted at rest. The encryption key for a given MPK may be periodically rotated. To effectuate this, the MPK is decrypted using the old encryption key, and re-encrypted with the new encryption key. The AES engine does not presently support using keys in the Key Vault as messages to encrypt. Therefore, MPKs must be decrypted into memory when their encryption keys are rotated. |
| MEK seeds (only used for derived MEKs) | See Section 4.6.5.3 for rationale on why the MEK seed is readable by firmware. |
| Random MEKs (upon initial generation) | Caliptra does not presently support a data path between its DRBG and its AES engine. Therefore, when a random MEK is first generated, KMB firmware reads the random MEK from the DRBG, and then encrypts it with the MDK (inner layer) and MEK secret (outer layer). See Section 4.6.5.1.1 for additional details on the motivation for the inner layer of MDK encryption. |

All keys held by KMB firmware are protected by Caliptra, and cannot be viewed by drive firmware.

#### 4.6.1.2 Key Derivation

In this specification, several flows involve deriving keys via a key derivation function (KDF). This specification leverages the KDF in Counter Mode defined in NIST SP 800-108 [10] section 4.1, where the PRF is HMAC512. As each key produced by this KDF is no larger than 512 bits, a single invocation of the underlying PRF is needed. Therefore, each instance of `SP 800-108 KDF` in subsequent flows deconstructs to the following operation:

$K_{OUT}$ = HMAC512($K_{IN}$, 0x01 || Label [ || 0x00 || Context])

There is one exception to this in Section 4.6.5.3, where CMAC instead of HMAC is used as the PRF for a particular instance of KDF. See Section 4.6.5.3.1 for details.

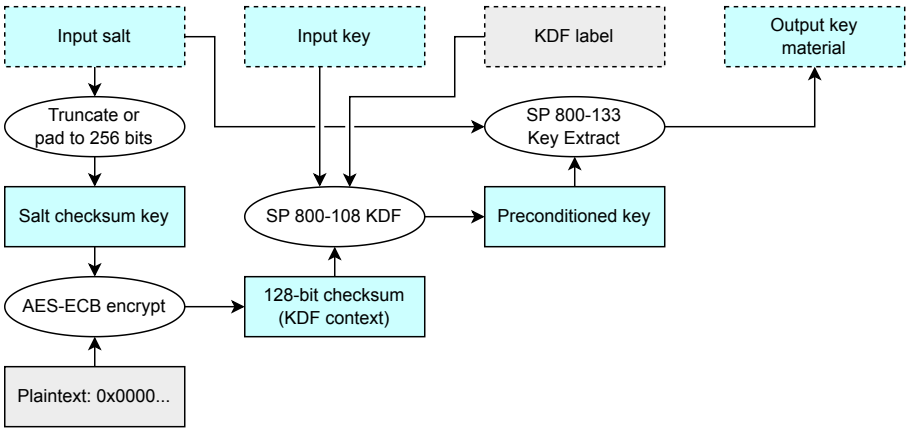
#### 4.6.1.3 Key Extraction

In this specification, several flows involve combining together multiple symmetric keys. NIST SP 800-133 [11] section 6.3 provides a number of FIPS-approved methods for combining keys together; this specification leverages the HMAC-based key-extraction process detailed there.

The keys being combined in this specification generally 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.

Section 6.3 of [11] stipulates that each component key "**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 EPK with different DPKs to produce different MEK secrets. Therefore, this specification defines the "Preconditioned Key Extract" operation, illustrated in Figure 3.




SP 800-133 Key Extract = T(HMAC-hash(salt, $K_1$ || ... || $K_n$ || $D_1$ || ... || $D_m$), kLen)

In this construction:

- T is the truncation operation, and kLen is 512
- HMAC-hash is HMAC512
- $K_1$ || ... || $K_n$ is the preconditioned key (n = 1)
- $D_1$ || ... || $D_m$ is the empty string (m = 0)

Therefore, SP 800-133 Key Extract reduces to HMAC512(salt, preconditioned key)

**Figure 3:** Preconditioned Key Extract

The input key is preconditioned using a unique identifier derived from the salt, before being combined with the salt using the SP 800-133 key-extraction process. Therefore each preconditioned key fed to the final HMAC step is unique to the given key mixing operation, and will not be re-used with a different salt to produce a different output key.

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

The design of this operation is informed by a number of constraints on Caliptra hardware. Keys in the Key Vault support the following symmetric cryptographic operations:

- Use as an AES key
- Use as an HMAC key or message (where the HMAC result is never revealed to firmware)

Notably, if a key in the Key Vault is used as an HMAC message, it constitutes the entirety of the HMAC message. No concatenation with other data is possible. Therefore a key in the Key Vault cannot be used as e.g. an SP 800-108 KDF's context string, which must be concatenated with other values such as the block counter.

The AES engine is used to compute the salt checksum because that is the only supported path for extracting an identifier from a key in Key Vault.

#### 4.6.1.4 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 [12] 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 Figures 4 and 5. These operations are similar in principle to XAES-256-GCM [13].

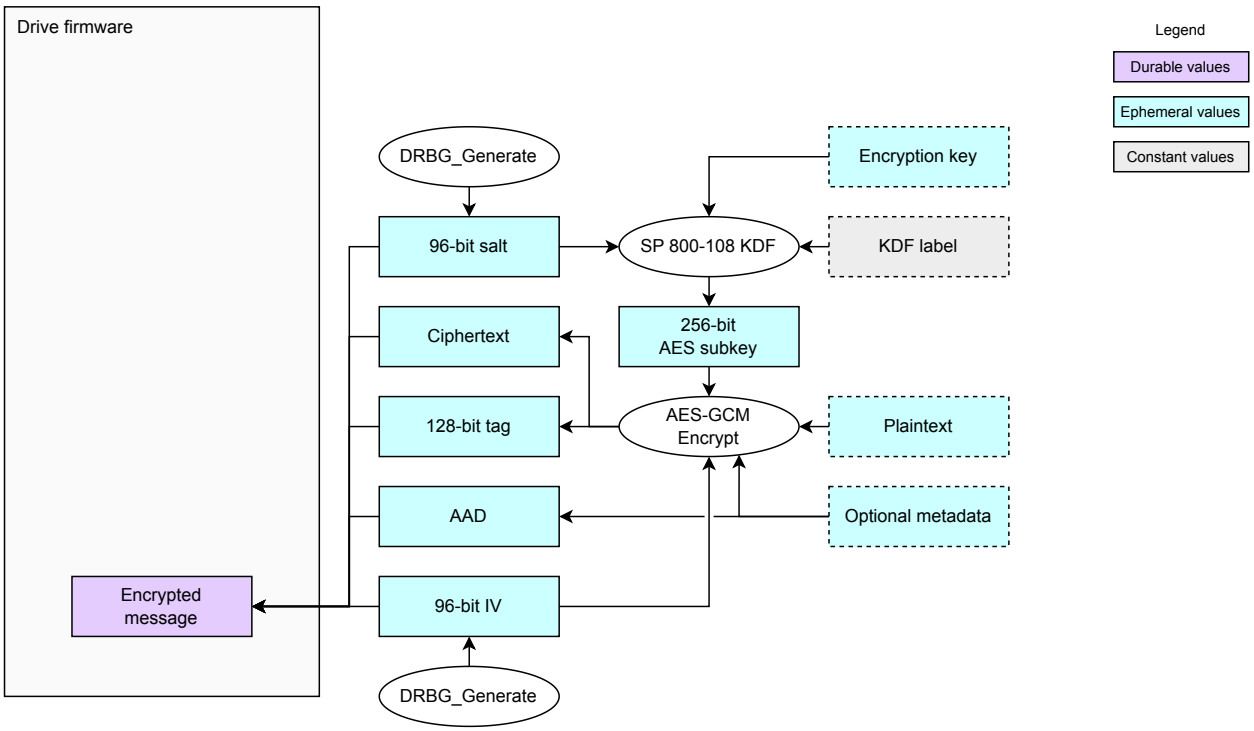



**Figure 4:** Preconditioned AES-Encrypt

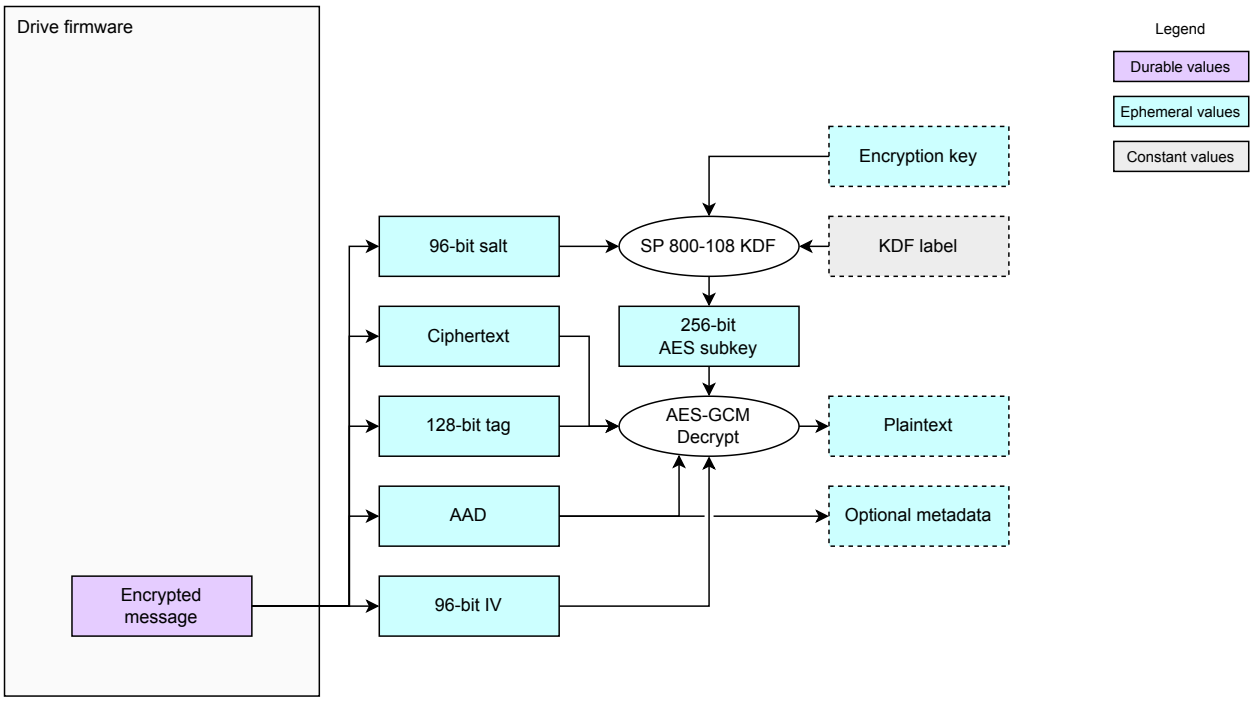



**Figure 5:** Preconditioned AES-Decrypt

Each encryption key is preconditioned using a randomly-generated 96-bit salt to compute a subkey, which is used with AES-GCM-Encrypt. This ensures that it is safe to use a given input encryption key in approximately $2^{80}$ preconditioned AES-Encrypt operations before an IV collision is expected to occur with probability greater than $2^{-32}$. The $2^{80}$ limit 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.

#### 4.6.1.5 VEK

The Volatile Escrow Key (VEK) is held in the Key Vault and is used to protect volatile keys held outside of the Key Vault, specifically Enabled MPKs.

The VEK is lazily generated upon first use and is lost when the drive shuts down. Lazy generation allows injected host entropy to contribute to the key's generation.

The VEK is randomly generated and held in a Key Vault slot. There does not exist a hardware data path directly between Caliptra's DRBG and Key Vault. Therefore, KMB firmware will read random data from the DRBG and send it to the Key Vault. To prevent KMB firmware from seeing the actual VEK, the randomness obtained by firmware is permuted using the HEK, as illustrated in Figure 6.

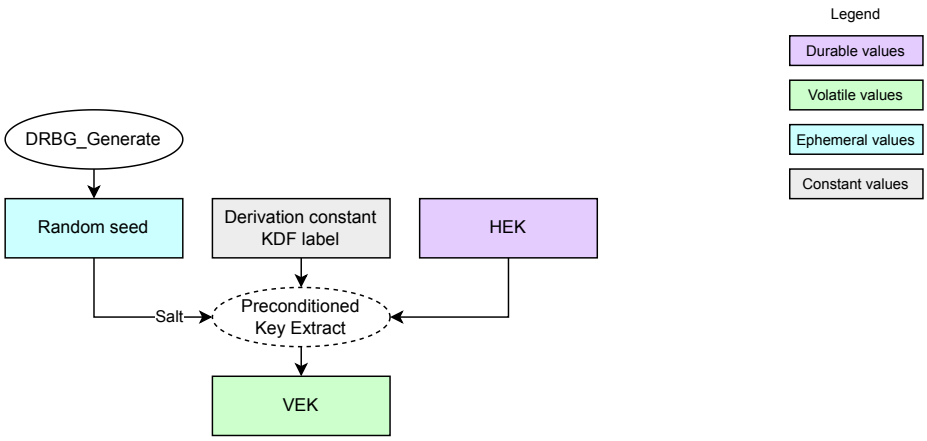



**Figure 6:** VEK Generation

### 4.6.2 MPKs

Multi-party Protection Keys (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 enabled. In both these states the MPK is encrypted to a secret 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.
- An Enabled MPK's encryption key is a common volatile secret held within KMB which is randomly generated and is lost when the drive shuts down. See Section 4.6.1.5 for details on how this key is generated. Enabled 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 "enabled" 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 supply the MPK's encrypted access key to drive firmware via a host-facing API. Upon receipt, the drive firmware passes that encrypted access key to KMB, along with the Locked MPK, to produce the Enabled MPK which is cached in drive memory. This is performed once for each MPK to which the MEK is bound, prior to drive firmware instructing KMB to program the MEK. See Section 4.8 for details on how host-facing APIs map to KMB mailbox commands.

#### 4.6.2.1 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 [14] identity. HPKE keypairs are randomly generated on KMB startup, and mapped to unique handles. Keypairs may be periodically rotated. Drive firmware is responsible for enumerating the available HPKE public keys and exposing them to the user via a host interface. See Section 4.8 for how the public keys may be presented to the host.

When a user wishes to generate or enable 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 enable the MPK.

Upon Caliptra cold reset or firmware-update reset, the HPKE keypairs are autonomously regenerated. See Section 4.6.8.

#### 4.6.2.1.1 HPKE algorithm support

An HPKE variant relies on three types of algorithms: KEM, KDF, and AEAD. Table 3 describes the HPKE algorithms that KMB supports.

**Table 3:** HPKE algorithm support

| Algorithm type | Algorithm | IANA code point | Reference document |
|---|---|---|---|
| KEM | ML-KEM-1024 | 0x0042 | PQ and PQ/T Hybrid Algorithms for HPKE [15] |
| | ML-KEM-1024 + P-384 | 0x0052 | |
| | P-384 | 0x0011 | RFC 9180 [9] |
| KDF | HKDF-SHA384 | 0x0002 | |
| AEAD | AES-256-GCM | 0x0002 | |

The definitions for the post-quantum KEMs are currently distributed across a number of documents. [16] contains generic constructions, [17] contains concrete parameterizations, and [15] contains HPKE IANA code-point assignments.[1]

Figures 7, 8, and 9 illustrate each KEM algorithm. The details of how the AES-GCM key and IV are derived are illustrated in Figure 10.

[1] As of this specification's publication, post-quantum HPKE KEMs are not fully ratified in IETF. However, the draft authors are confident that the IANA code points and associated definitions will not change before the drafts are finalized, with two exceptions for the ML-KEM-1024 + P-384 KEM parameterization specified in [17]: the KDF will be updated from HKDF-SHA-256 to SHA3-256, and the label may be changed. The IETF editors expect to make these changes in September 2025. Figure 8, where this KEM is illustrated, uses the SHA3-256 KDF and the label currently specified in [17].

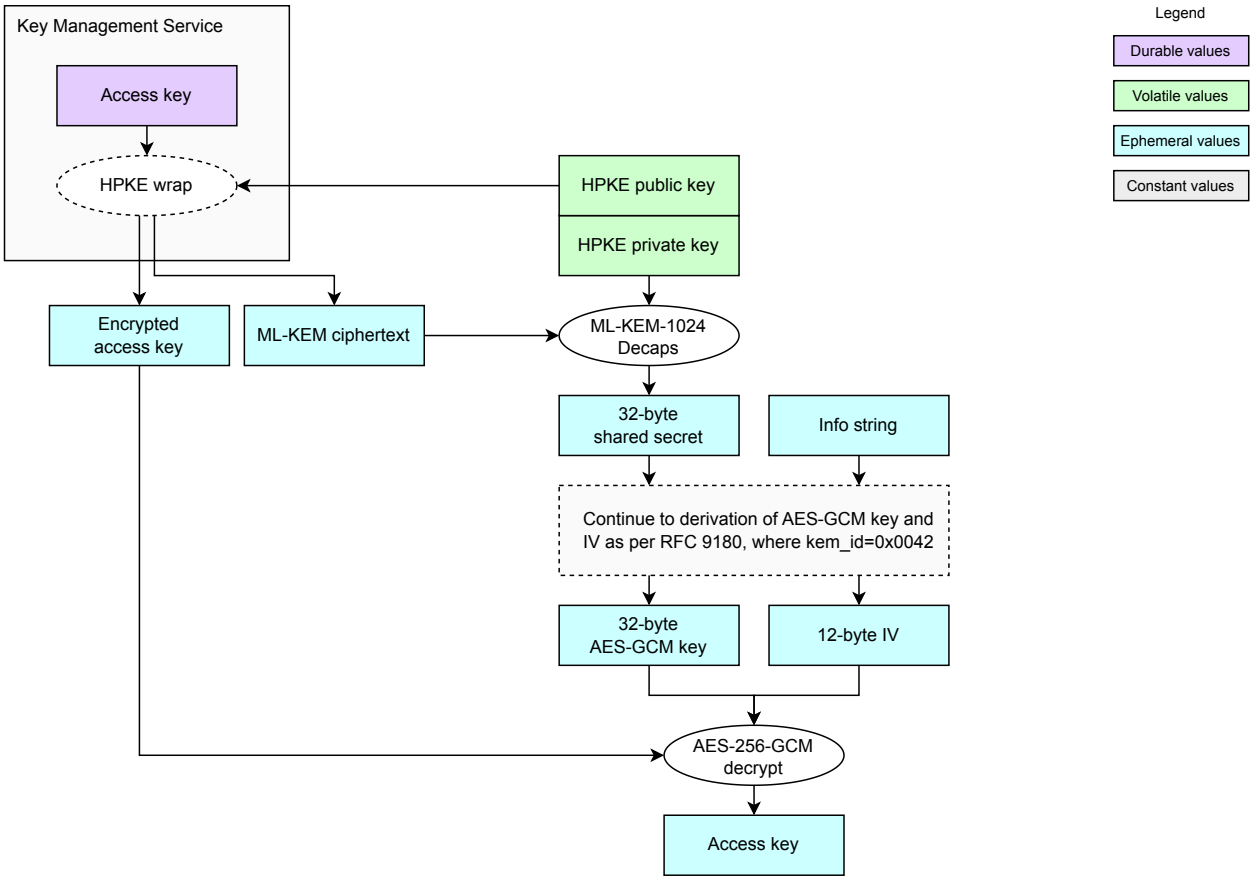



**Figure 7:** HPKE unwrap for access keys with ML-KEM-1024

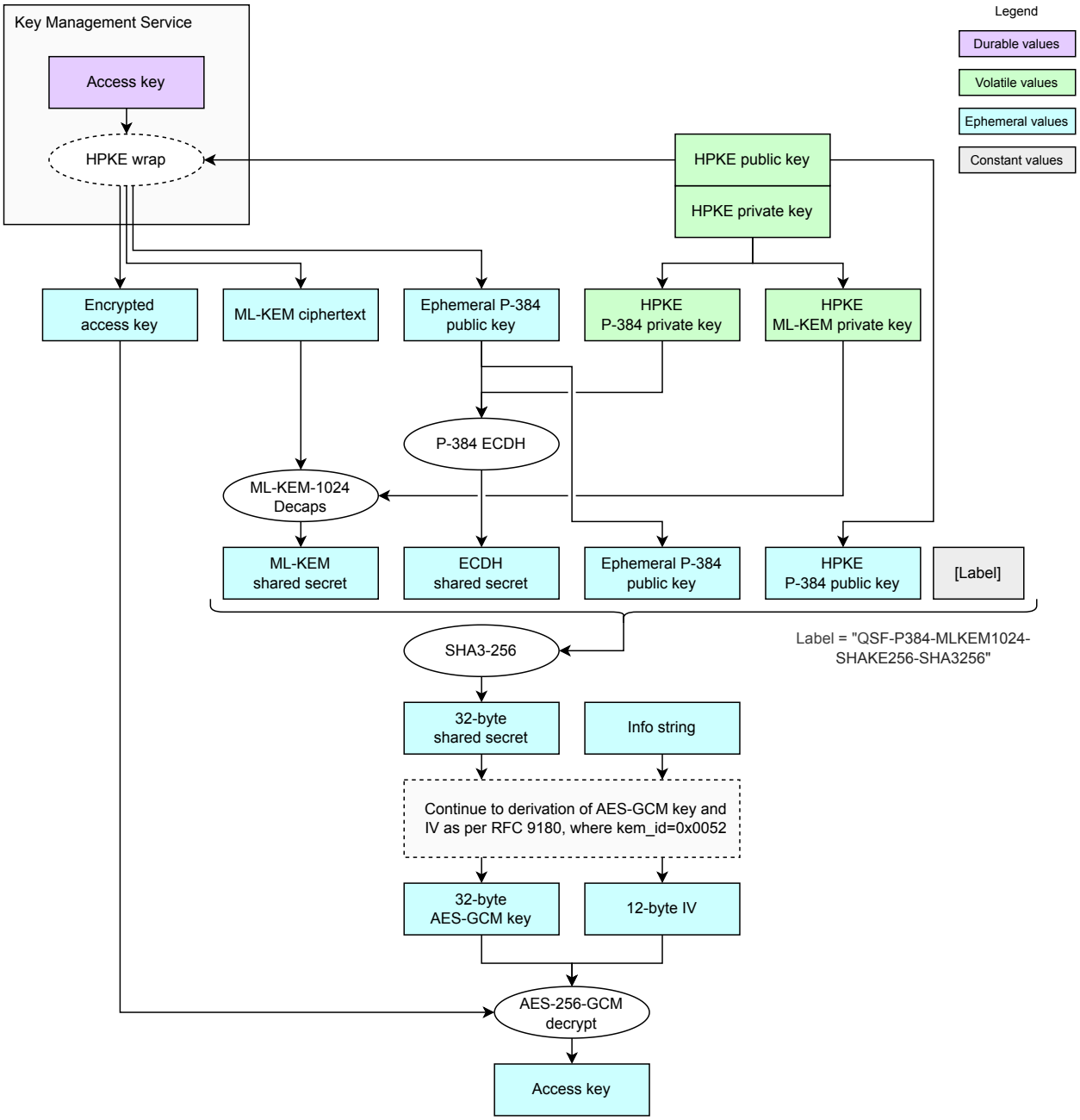



**Figure 8:** HPKE unwrap for access keys with hybrid ML-KEM-1024 + P-384 ECDH

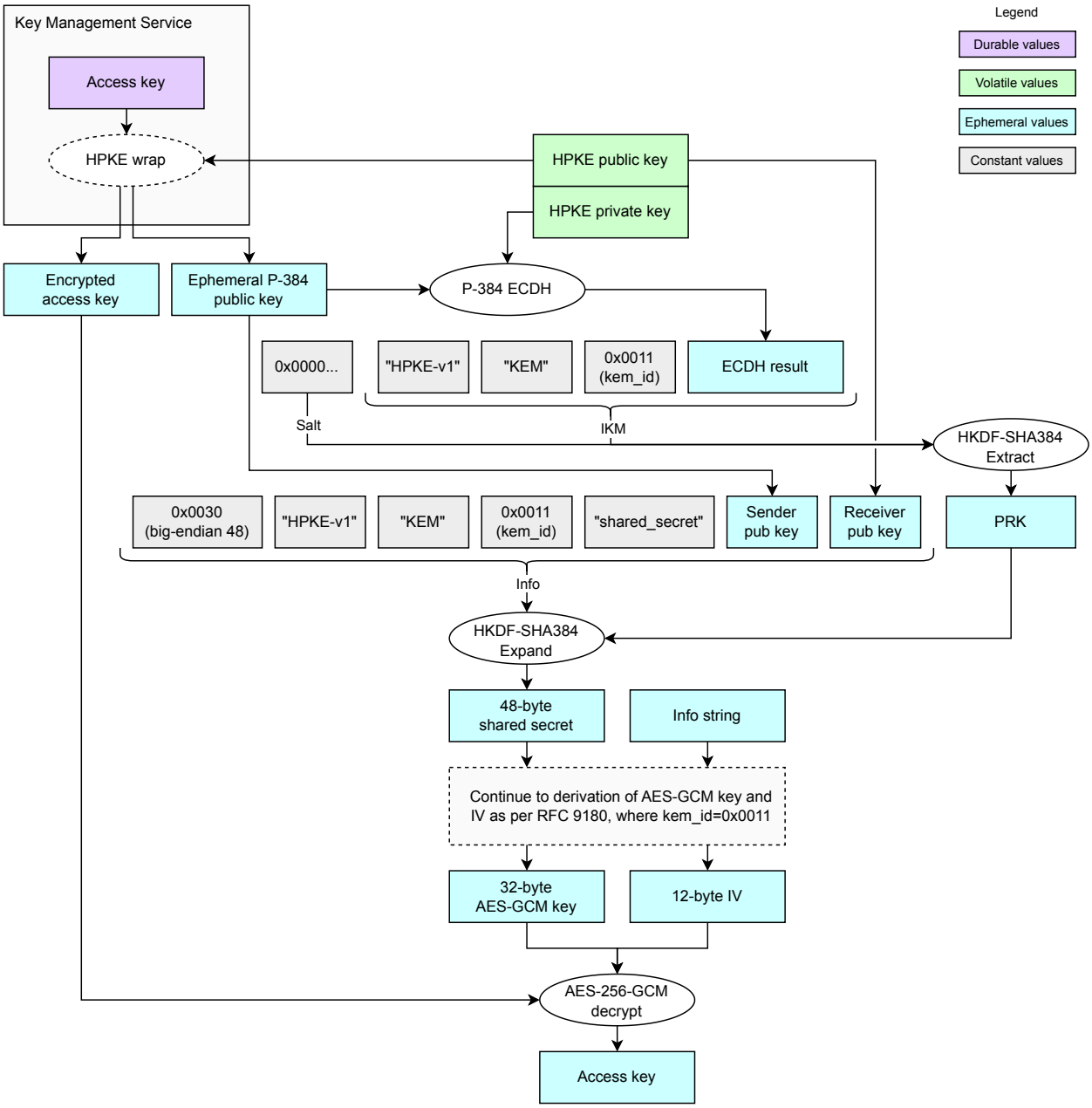



**Figure 9:** HPKE unwrap for access keys with P-384 ECDH

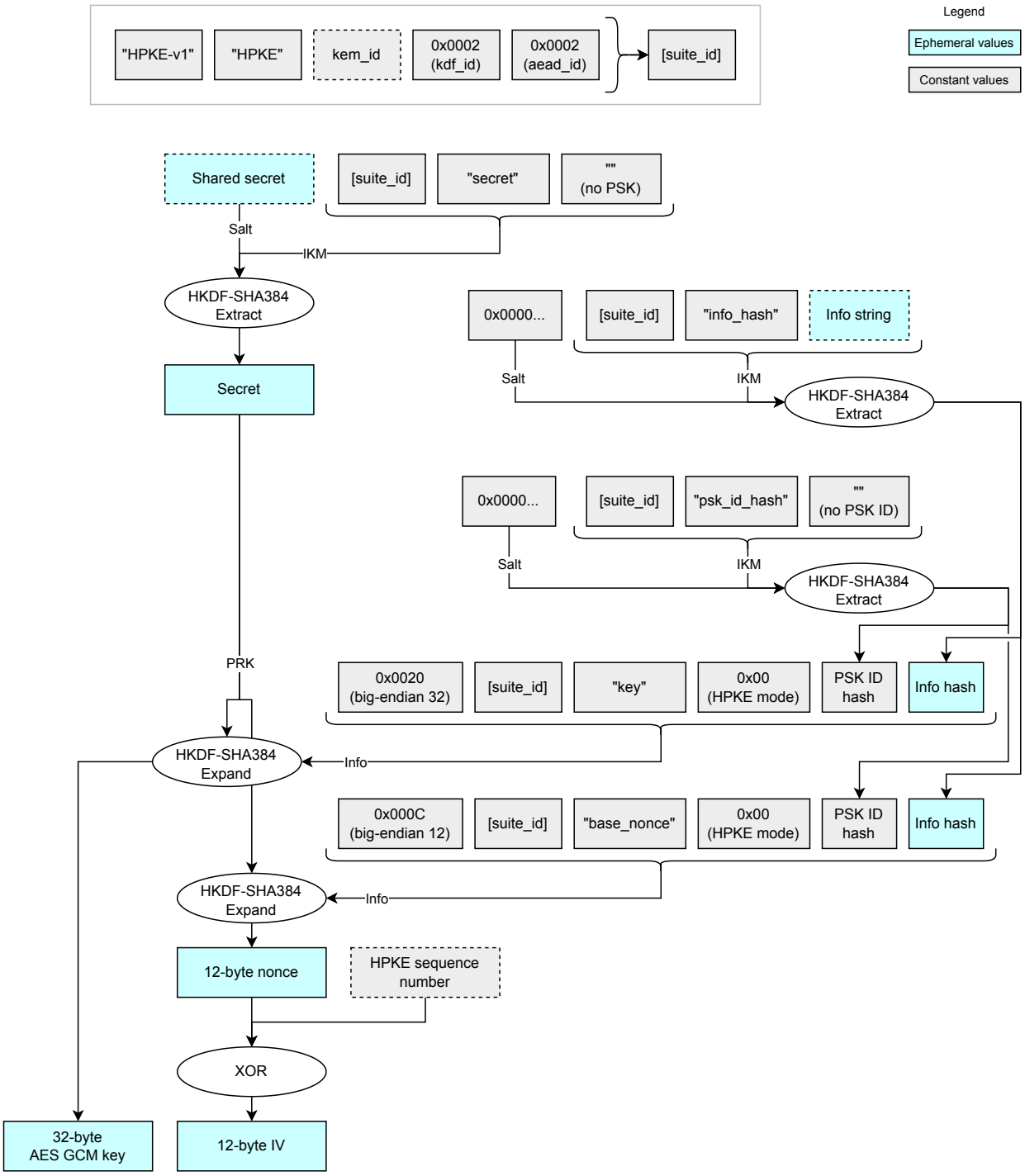



**Figure 10:** HPKE AES-GCM key and IV derivation

The HPKE 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 4.6.2.2.3, 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.

#### 4.6.2.1.2 HPKE public key endorsement

Upon request, KMB can issue an endorsement for a given HPKE public key, signed by Caliptra runtime firmware's DICE alias key. The endorsement and its fields are populated by KMB firmware. This is illustrated in Figure 11.

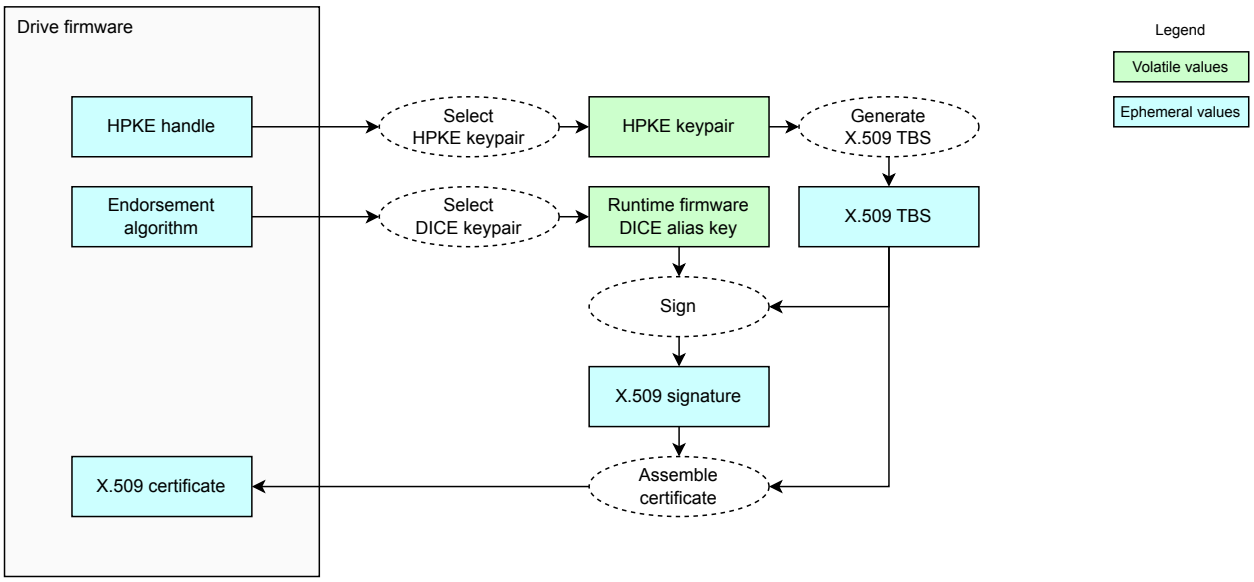



**Figure 11:** HPKE public key endorsement

The endorsement includes the HPKEIdentifiers extension as defined in [18], with the `kem_id`, `kdf_id`, and `aead_id` identifiers populated based on the HPKE keypair's algorithms. See Table 3 for the algorithm identifiers supported by KMB.

### 4.6.2.2 MPK lifecycle

MPKs can be generated, enabled, and have their access keys rotated and tested.

#### 4.6.2.2.1 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. See Section 4.8 for guidance on how drive firmware should populate 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.

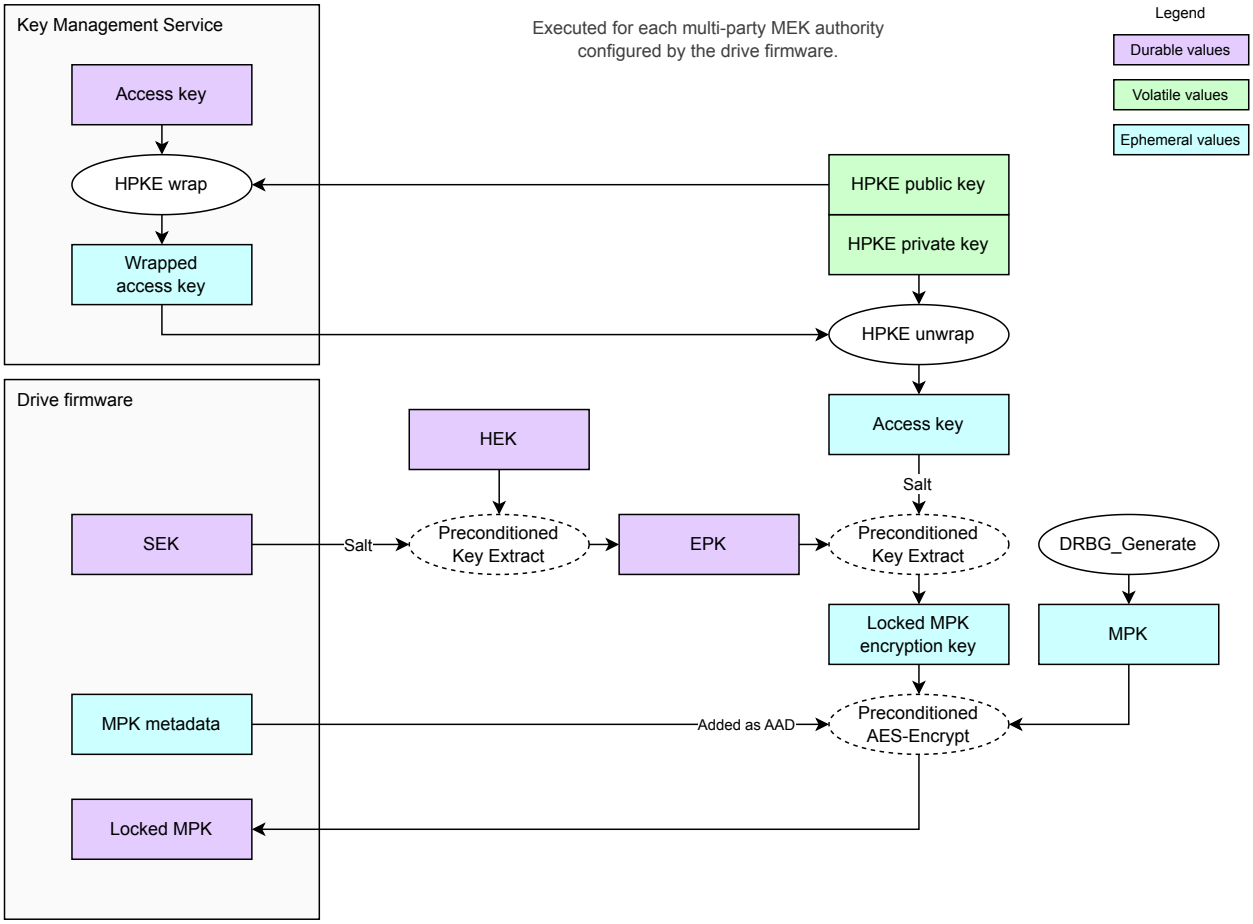



**Figure 12:** MPK generation

#### 4.6.2.2.2 MPK enabling

Encrypted MPKs stored at rest in persistent storage are considered “locked”, and must be enabled before they can be used to load an MEK. Enabled MPKs are encrypted to the VEK when handled by drive firmware. Enabled MPKs are invalidated when the VEK is rotated, which occurs on Caliptra cold reset. See Section 4.6.8.

To enable 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 VEK, preserving the MPK metadata.
5. Return the re-encrypted “enabled” MPK to the drive firmware.

Drive firmware may then stash the encrypted Enabled MPK in volatile storage, and later provide it to the KMB when loading an MEK, as described in Section 4.6.5.

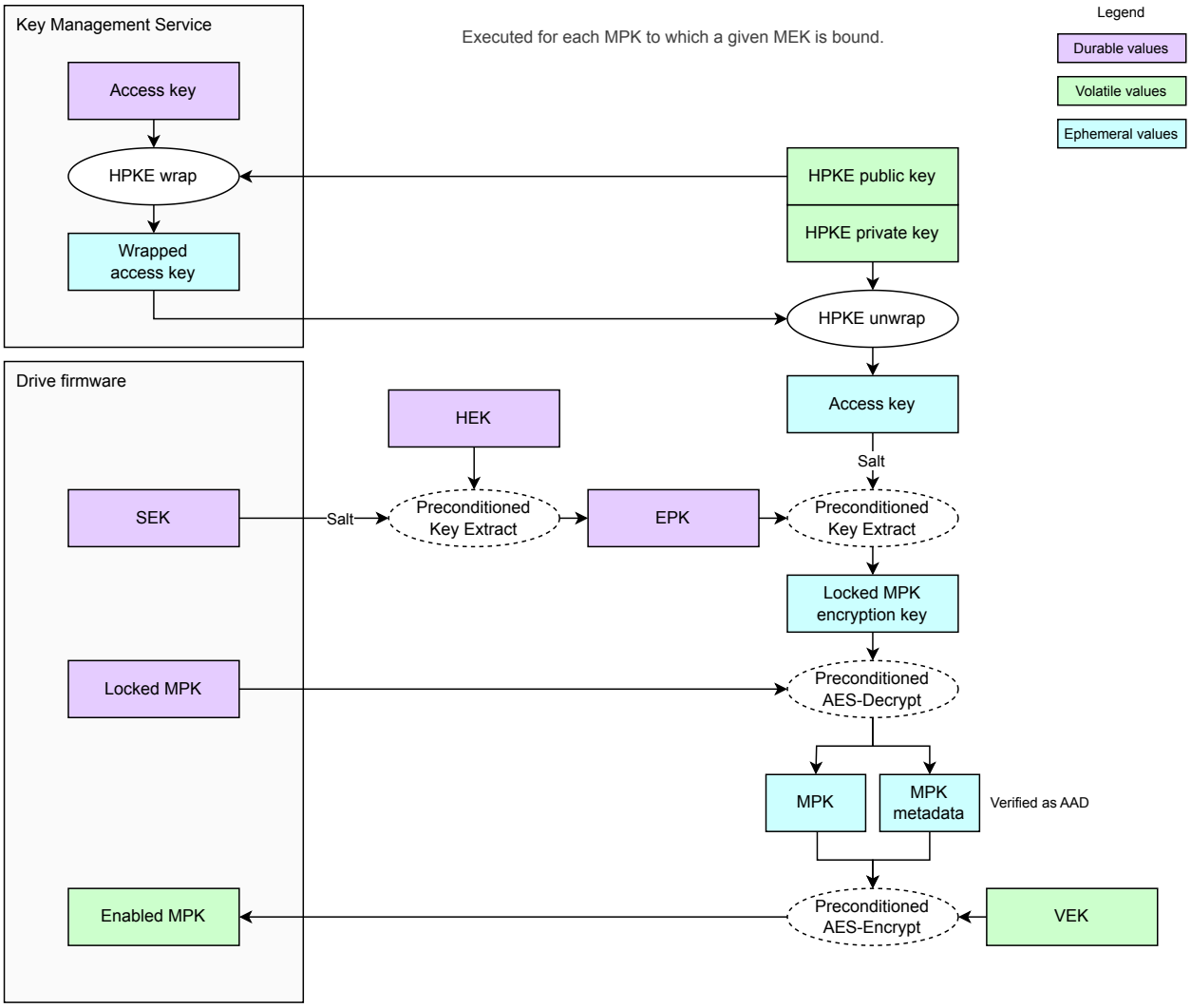



**Figure 13:** MPK enabling

#### 4.6.2.2.3 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.

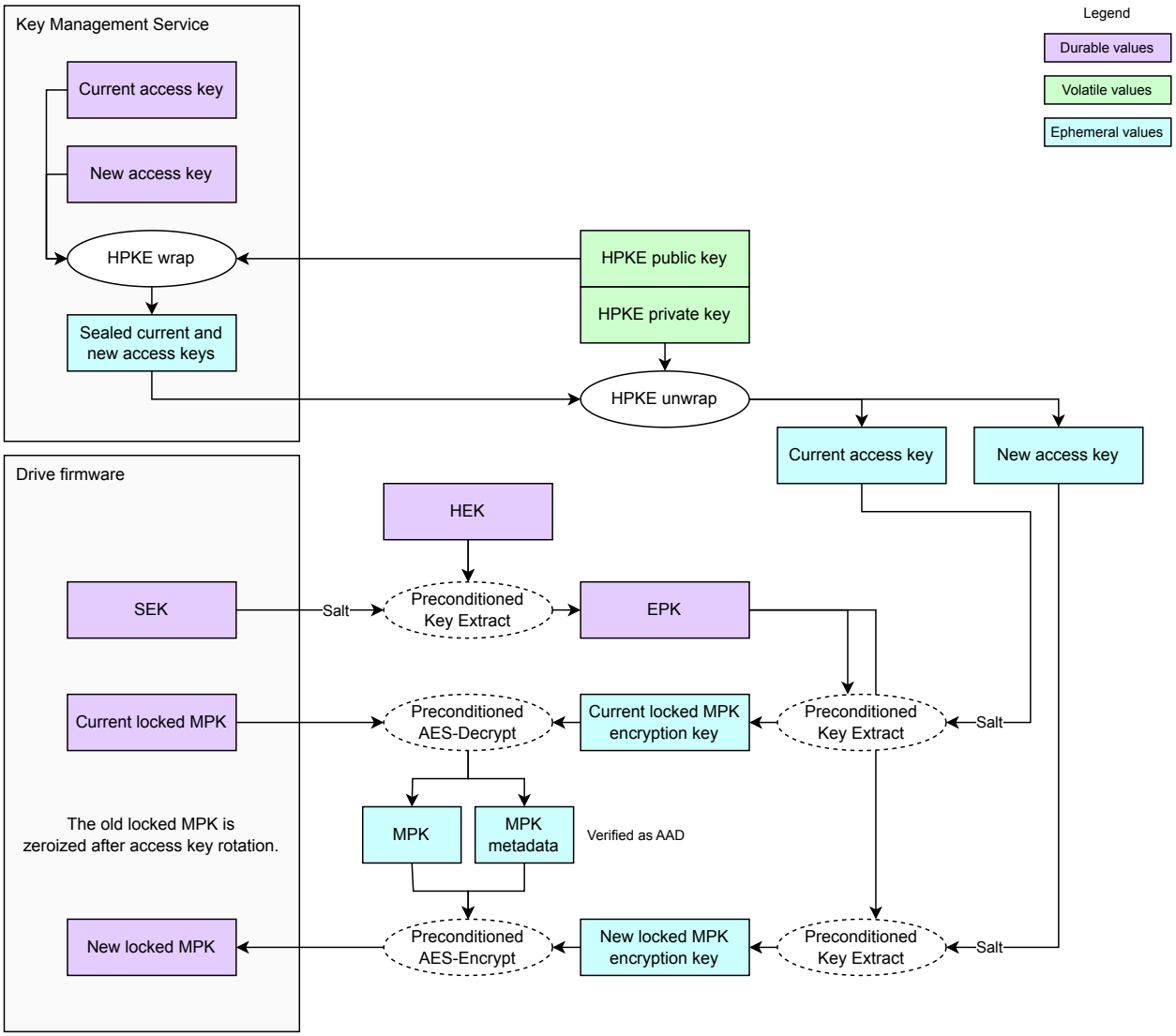



**Figure 14:** MPK access key rotation

#### 4.6.2.2.4 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.

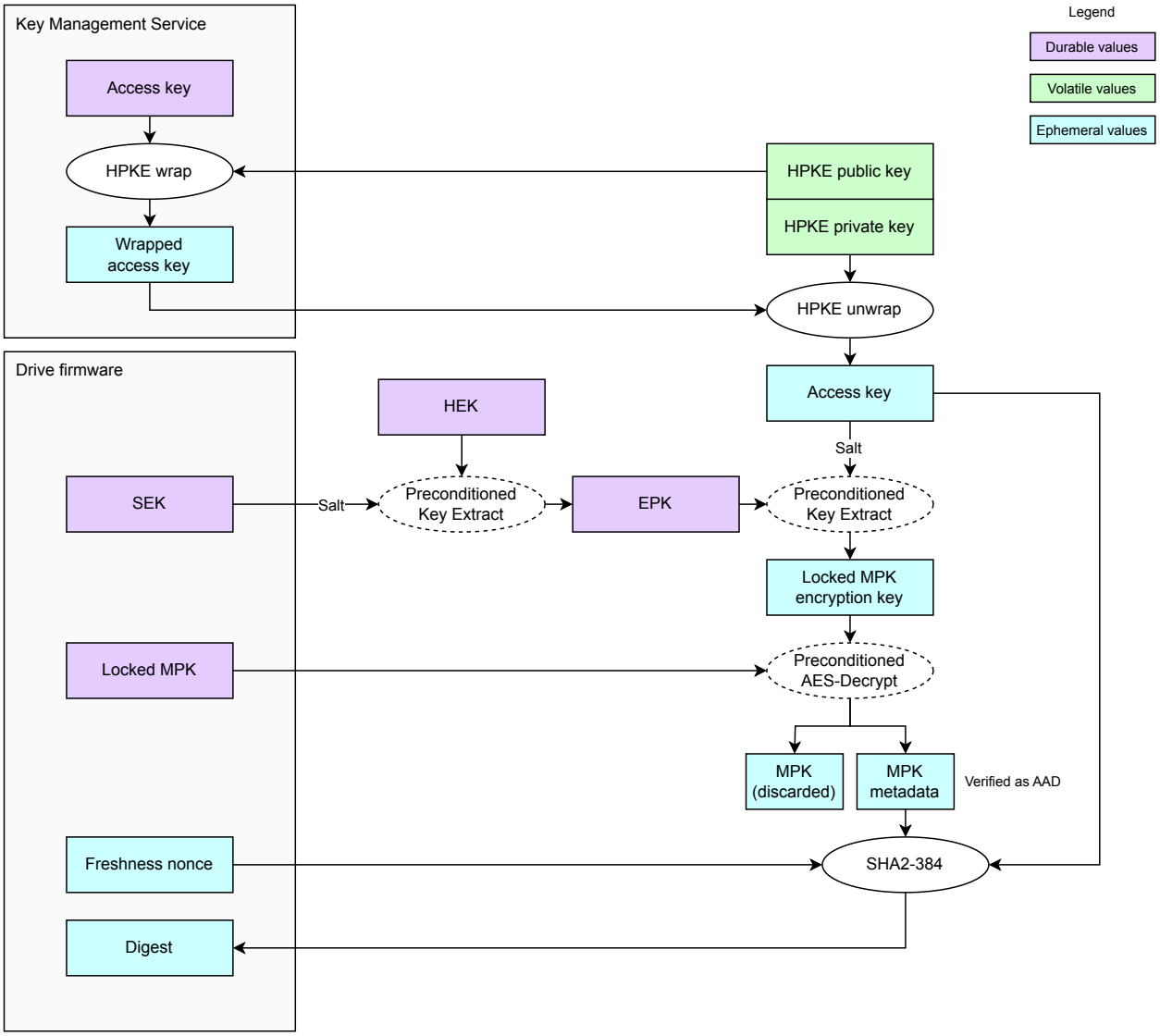



**Figure 15:** MPK access key testing

### 4.6.3 DPKs

Drive firmware provides a Data Protection Key (DPK) 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 4.6.5.

#### 4.6.3.1 Example flow when generating or loading an MEK

In this flow, the DPK may be encrypted by a user's C_PIN. Drive firmware logic which decrypts MEKs can be repurposed to produce the DPK. This flow may be used to support TCG Opal or Enterprise.

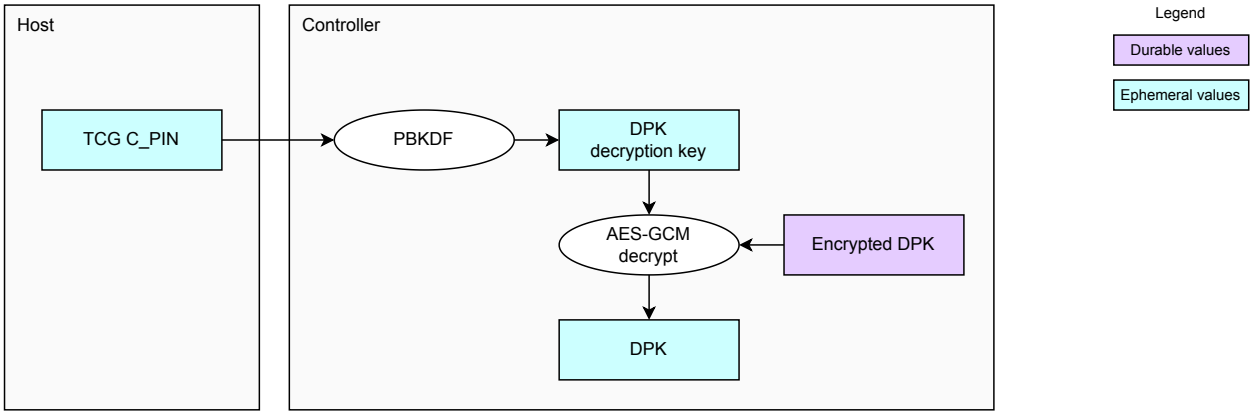



**Figure 16:** Example drive flow to decrypt a DPK based on a host-provided C_PIN

#### 4.6.3.2 Example flow when deriving an MEK

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

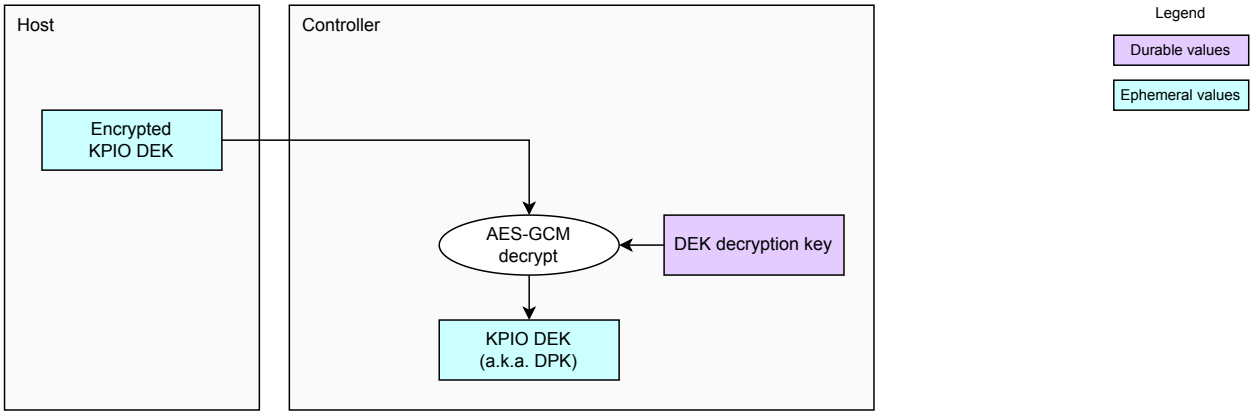



**Figure 17:** Example drive flow to accept an injected DPK

### 4.6.4 HEKs and SEKs

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

- The **HEK** is derived from seeds 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, with edge cases detailed in Section 4.6.4.1.
- The **SEK** is managed by drive firmware, and may be stored in flash. If the SEK is not randomized, 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.

The drive must only provide HEK seeds or SEKs to KMB that are cryptographically-strong random values, because otherwise it is not possible to cryptographically erase data through zeroization of those values. Note: as depicted in Figure 18, MCU ROM may provide an all-zeroes HEK seed to KMB, provided that it correctly indicates to KMB that the HEK seed is not randomized.

#### 4.6.4.1 HEK lifecycle

Caliptra with L.O.C.K. features a series of n 256-bit HEK slots present in a fuse bank, dubbed $HEK_0$..$HEK_{n-1}$, where $4 \leq n \leq 16$. The vendor is responsible for determining the number of HEK slots available in fuses. These HEK slots are programmed and read by MCU, via the Caliptra fuse controller. MCU is responsible for transitioning each HEK slot individually from blank → randomized → zeroized. $HEK_x$ shall only be randomized once $HEK_{x-1}$ has been zeroized. MCU shall zeroize a HEK slot by blowing every bit.

Each HEK slot contains a HEK seed. MCU ROM reads the currently-randomized HEK slot from fuses during cold reset, and passes its contents to Caliptra Core via a fuse register.

There are two edge cases where the HEK is available even if there is no randomized seed available in the HEK fuse bank. In both cases, the HEK is derived from an all-zeroes HEK seed.

- To enable pre-production testing, if Caliptra is in the Unprovisioned or Manufacturing lifecycle state, the HEK is available.
- To enable lengthening the useful lifespan of the storage device, once all HEK slots have been zeroized, the device can be permanently transitioned into a mode where the HEK is available. A field-programmable "permanent-HEK" logical fuse bit is allocated to determine whether this mode is active.

Table 4 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 HEK fuse bank.

**Table 4:** HEK lifecycle states

| Caliptra lifecycle state | HEK fuse state, where x is the current HEK slot | HEK seed value | Reported HEK state |
|---|---|---|---|
| Unprovisioned or Manufacturing | [Any] | 0x0000… | HEK_AVAIL_UNERASABLE |
| Production | HEK slot 0 is unprogrammed. | N/A | HEK_UNAVAIL_EMPTY |
| Production | HEK slot x is randomized. | HEK slot x | HEK_AVAIL_PROGRAMMED |
| Production | HEK slot x is zeroized. | N/A | HEK_UNAVAIL_ZEROIZED |
| Production | HEK slot x is corrupted. | N/A | HEK_UNAVAIL_CORRUPTED |
| Production | Every HEK slot is zeroized, and the permanent-HEK fuse bit has been set. | 0x0000… | HEK_AVAIL_UNERASABLE |

In addition to the HEK seed, the HEK is derived from Caliptra's DICE UDS. The mechanism used to compute the HEK is described in Figure 18. This flow is performed across MCU ROM and Caliptra ROM upon KMB startup.

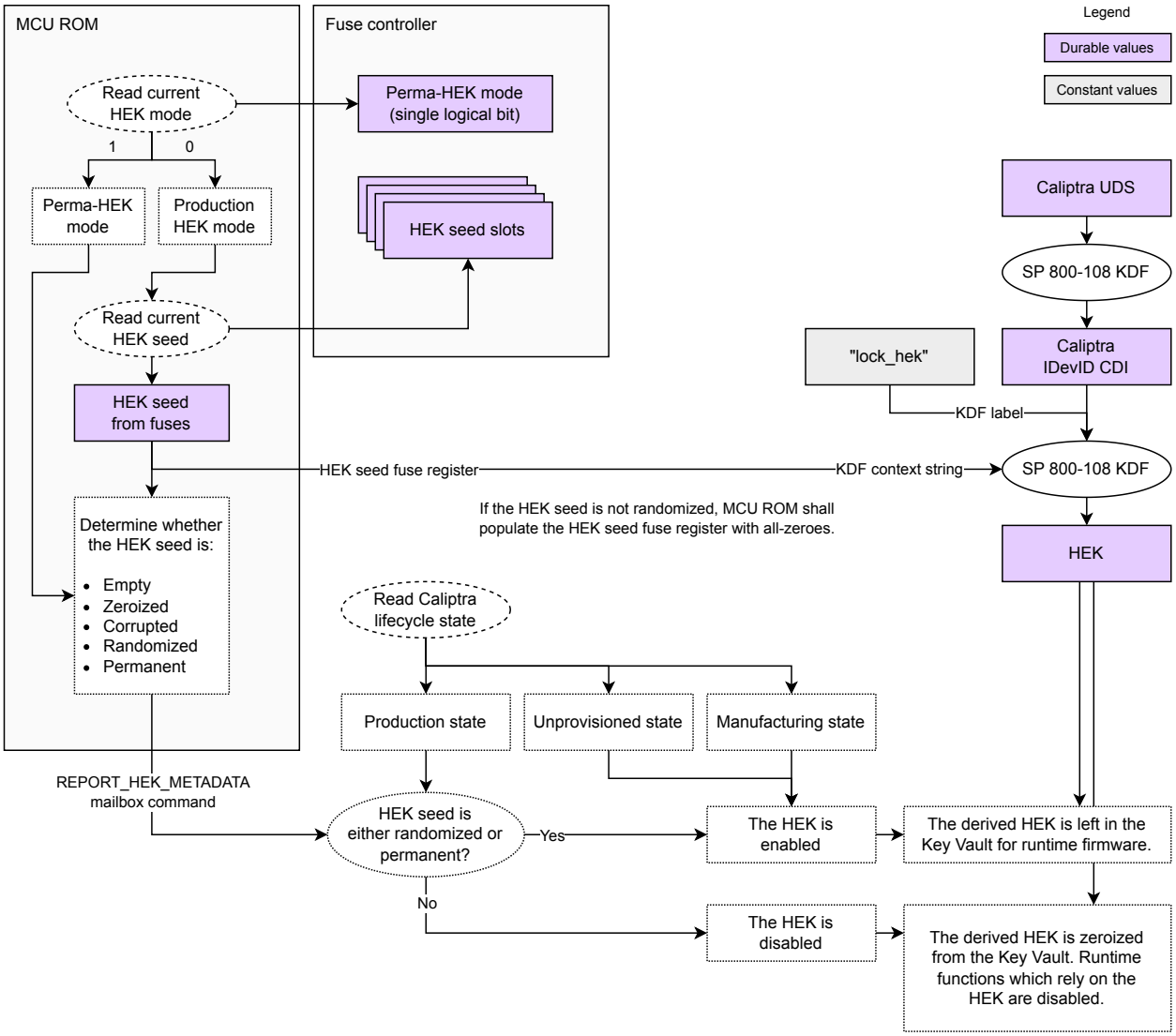



**Figure 18:** HEK derivation

MCU ROM is responsible for invoking the REPORT_HEK_METADATA command exactly once, after cold reset and before Caliptra FMC/runtime firmware is first loaded. If REPORT_HEK_METADATA is not invoked, KMB will zeroize the HEK and runtime functions which rely on it will be disabled.

By deriving the HEK in ROM based on secrets that are not available to runtime firmware, KMB ensures that compromised runtime firmware cannot derive the HEK using replayed HEK seed values.

Note: the HEK is derived before REPORT_HEK_METADATA is invoked due to IDevID having been zeroized from the Key Vault by the time Caliptra ROM is ready to handle mailbox commands.

#### 4.6.4.2 HEK and SEK lifecycle

Figure 19 illustrates the case where a drive ships with four HEK slots.

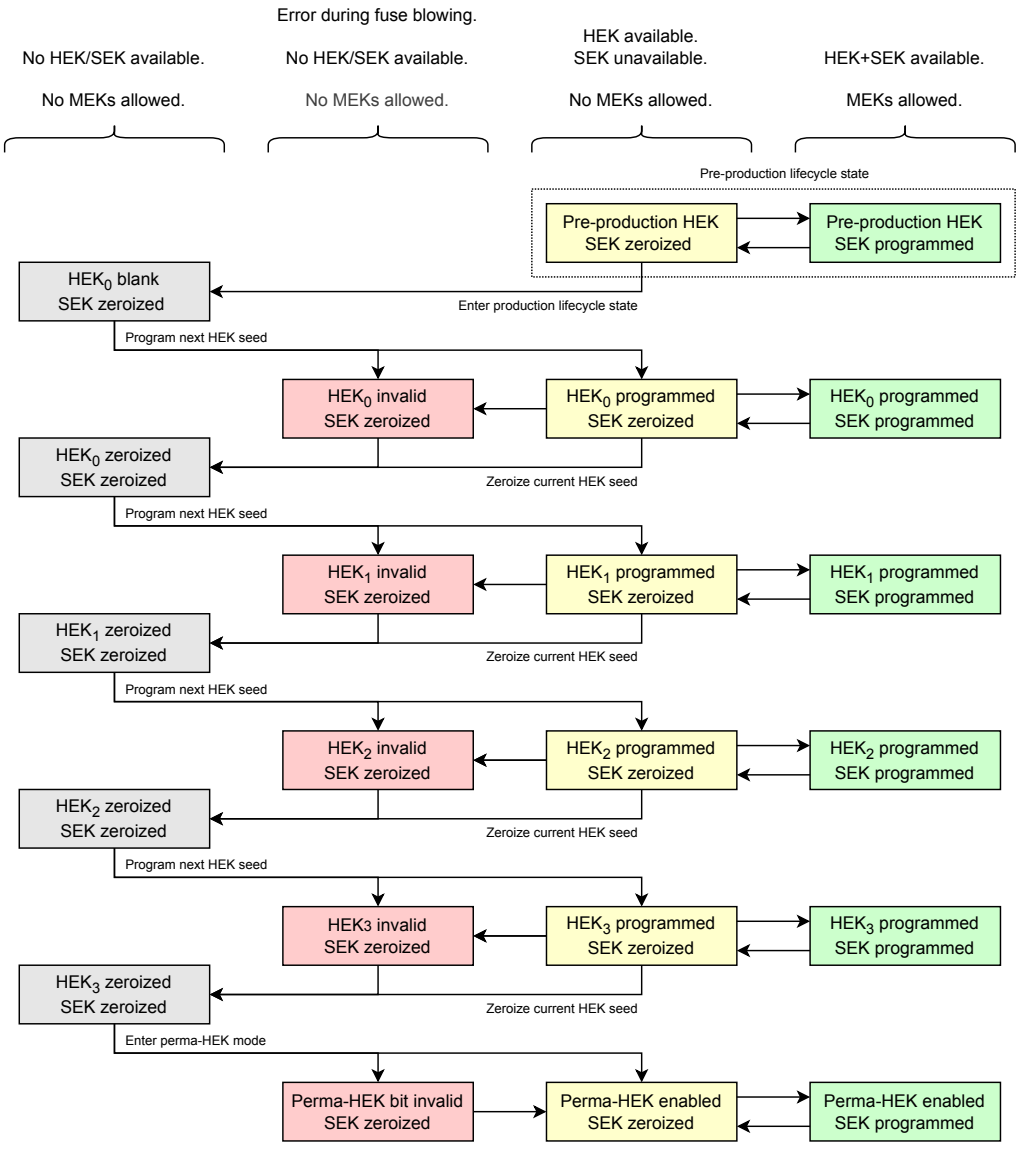



**Figure 19:** HEK and SEK state machine

Drive firmware is responsible for performing each state transition, including programming and zeroizing HEK seeds and entering perma-HEK mode.

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

Drive firmware is responsible for enforcing that HEK and SEK programming / zeroization follows the state machine illustrated in Figure 19. Specifically:

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

- The HEK seed shall only be zeroized once the SEK is zeroized.
- The HEK seed shall only be zeroized after it has been randomized or corrupted.
- The next HEK seed shall only be programmed once the prior HEK seed has been zeroized.
- Perma-HEK mode shall only be enabled once all HEK seeds have been zeroized.

Drive firmware is responsible for zeroizing the encryption engine's key cache when the SEK transitions from randomized to zeroized. HEK transitions involve power cycles and therefore the key cache is implicitly zeroized across such transitions.

### 4.6.5 MEKs

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

Figure 20 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.

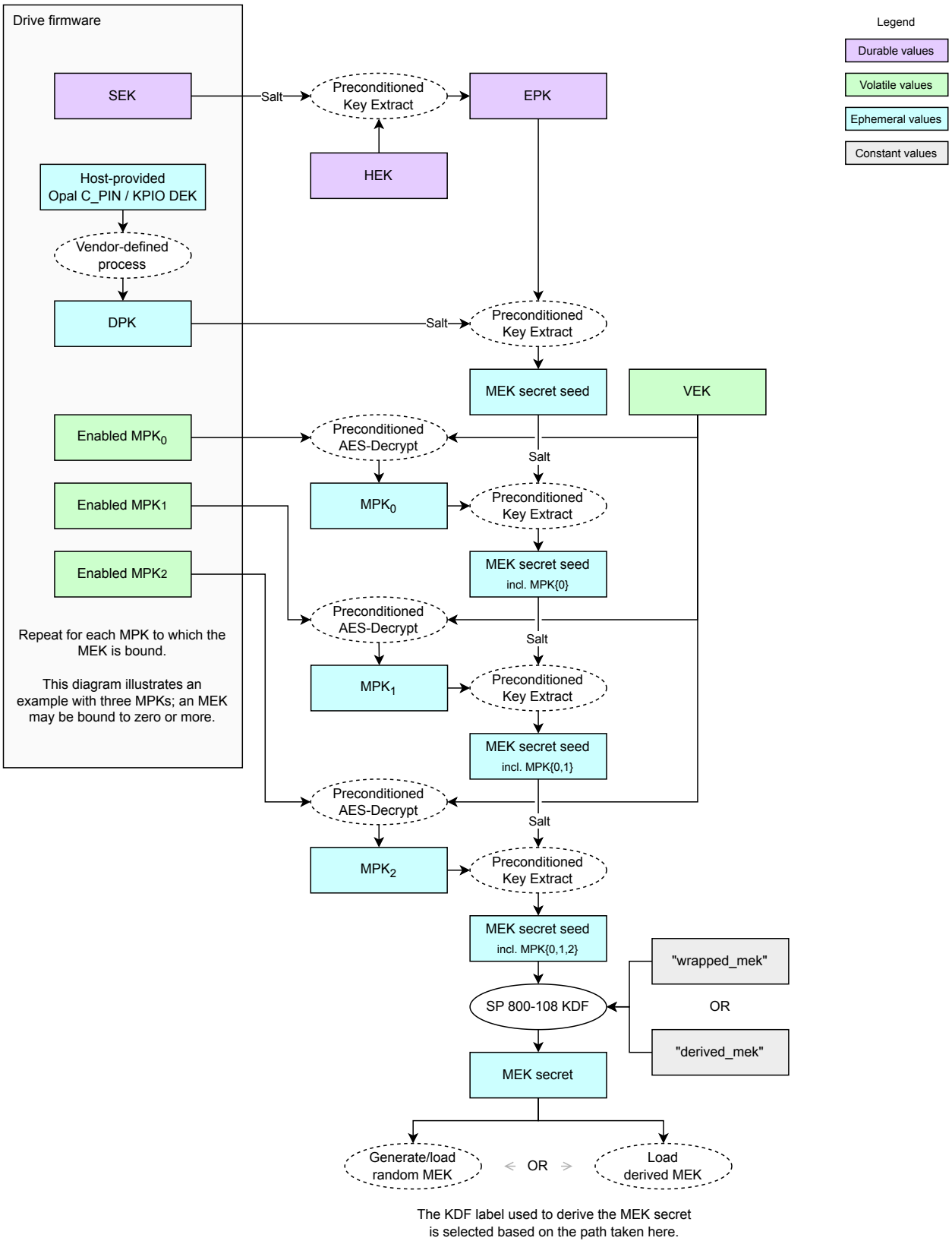



**Figure 20:** MEK secret derivation

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

See Section 4.6.5.2 for how the MEK secret is used to generate and load random MEKs. See Section 4.6.5.3 for how the MEK secret is used to derive deterministic MEKs.

#### 4.6.5.1 Support for non-MPA storage API flows

KMB allows an MEK to be bound to zero MPKs, which supports TCG flows that do not interact with the TCG MEK-MPA specification. The process for establishing such an MEK is illustrated in Figure 21.

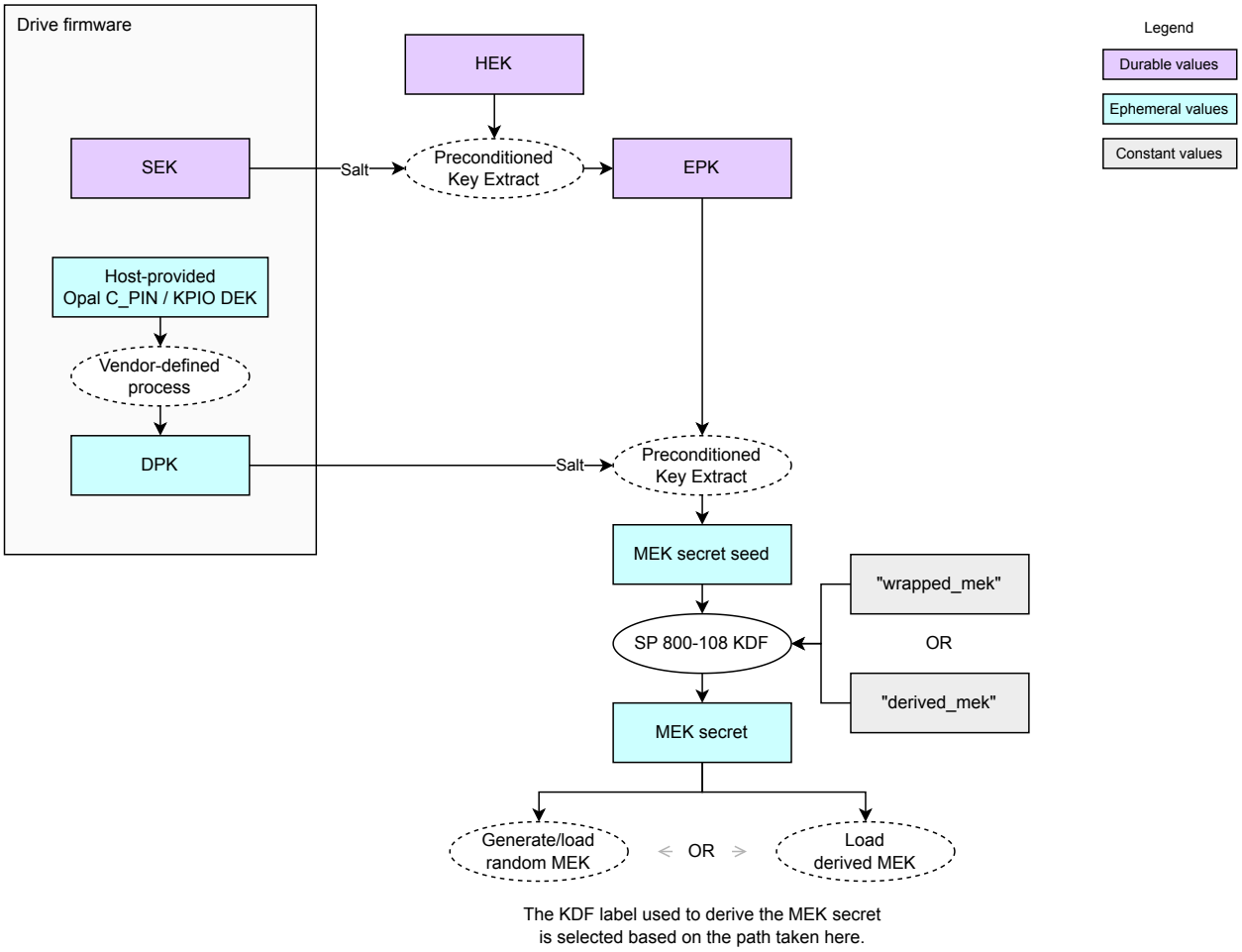



**Figure 21:** Establishing an MEK bound to zero MPKs

##### 4.6.5.1.1 Protecting MEKs with the MDK

Randomly-generated MEKs are encrypted at rest with AES-GCM, as illustrated in Section 4.6.5.2. During initial MEK encryption, Caliptra's AES engine accepts the MEK plaintext from KMB firmware and returns the ciphertext back to KMB firmware. Later, during MEK decryption, Caliptra's AES engine accepts the MEK ciphertext from KMB firmware and writes the decrypted plaintext to the Key Vault. From Caliptra hardware's perspective, AES-GCM

encryption and decryption are the same operation. Therefore, if the MEK were protected with only a single layer of AES-GCM encryption, compromised KMB firmware could instruct Caliptra's AES engine to treat decryption like encryption and return the decrypted plaintext to KMB firmware, trivially disclosing the MEK.

To mitigate this, Caliptra hardware enforces that before an MEK can be loaded into the Key Vault, it must first pass through an AES-ECB decryption operation keyed using the MEK Deobfuscation Key (MDK). The MDK is derived in ROM upon cold reset, as described in Section 4.6.8 and illustrated in Figure 22.

Unlike with AES-GCM, Caliptra's AES engine can differentiate between encryption and decryption in AES-ECB mode. Hardware enforces that for any AES-ECB decryption operation using the Key Vault slot that holds the MDK, the result is always routed to the Key Vault. The MDK encryption layer therefore ensures that the plaintext MEK is never visible to KMB firmware once the MEK has been decrypted.

By deriving the MDK in ROM based on secrets that are not available to runtime firmware, KMB ensures that compromised runtime firmware cannot re-derive the MDK into a separate Key Vault slot, which could have looser protections for whether firmware can see the results of an AES-ECB decryption operation.

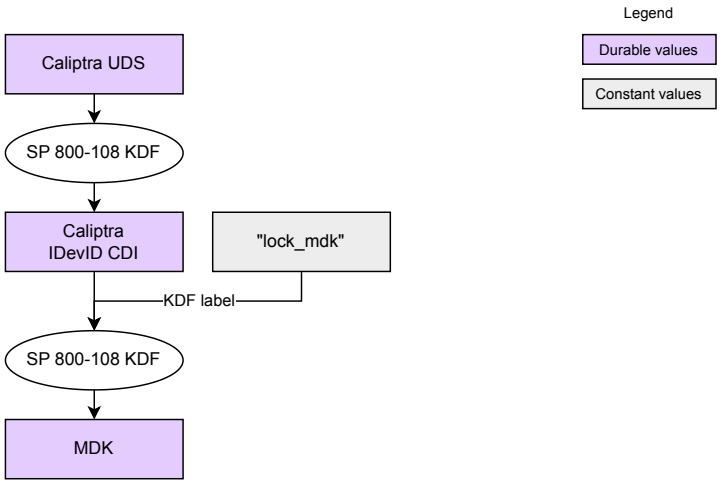



**Figure 22:** Deriving the MDK

#### 4.6.5.2 Generating and loading a random MEK

Figures 23 and 24 elide the process of deriving the MEK secret, as those details are captured in Figure 20. When an MEK is generated or loaded, the inputs given in Figure 20 are provided by drive firmware so that KMB may derive the MEK secret.

Randomly-generated MEKs are given two layers of at-rest encryption: an inner AES-ECB layer using the MDK, and an outer AES-GCM layer using the MEK secret. The wrapped MEK's integrity tag is associated with the outer layer. The inner AES-ECB layer does not have its own MAC. Therefore the extra layer of encryption does not modify the size of the wrapped MEK. See Section 4.7.2.22.2 for details on the format and layout of the wrapped MEK.

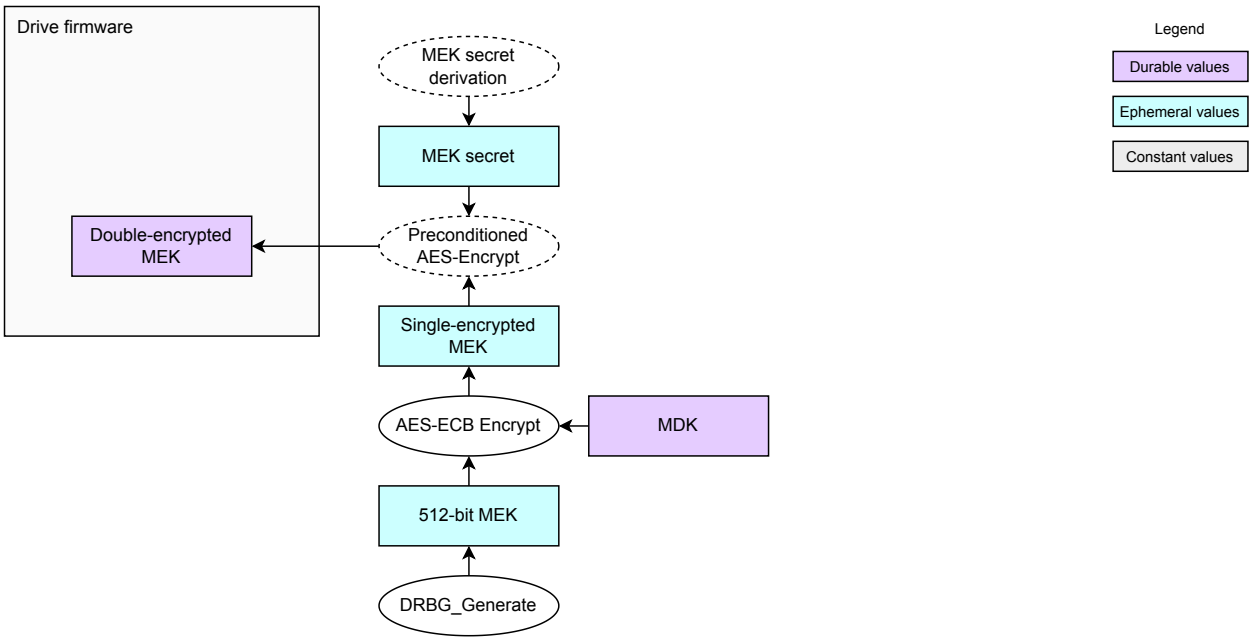



**Figure 23:** Generating an MEK

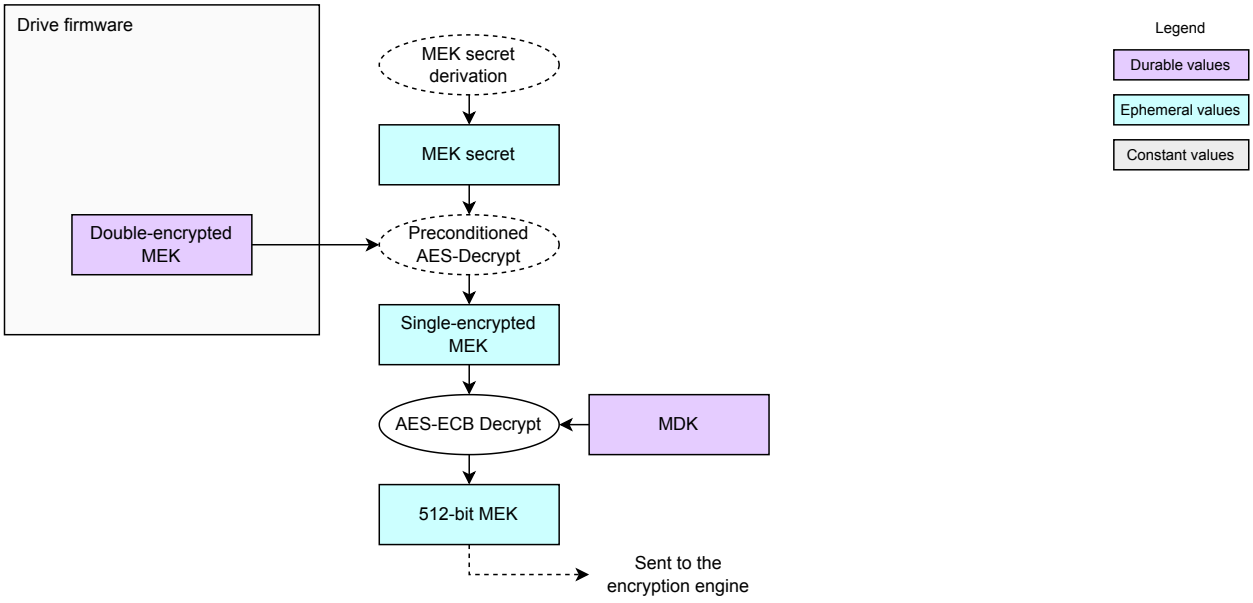



**Figure 24:** Loading an MEK

#### 4.6.5.3 Deriving an MEK

Figure 25 elides the process of deriving the MEK secret, as those details are captured in Figure 20. When an MEK is derived, the inputs given in Figure 20 are provided by drive firmware so that KMB may derive the MEK secret.

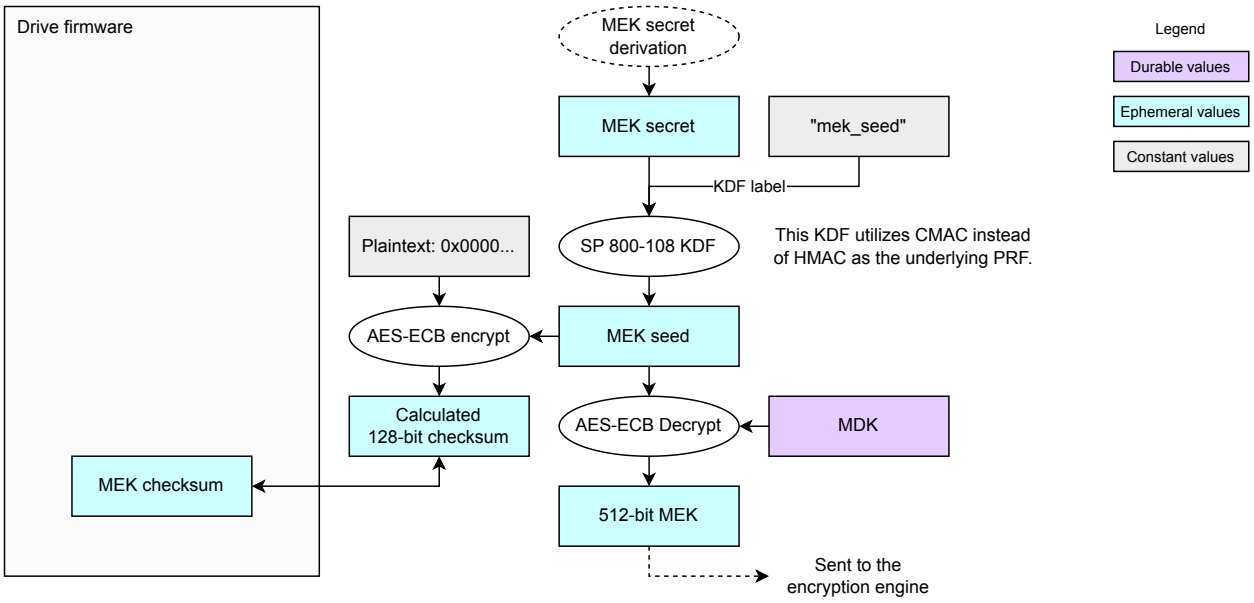



**Figure 25:** Deriving an MEK

#### 4.6.5.3.1 Deobfuscating MEK seeds with the MDK

The MEK seed is calculated via CMAC instead of HMAC, and then “decrypted” via the MDK, for the following reasons:

- As described in Section 4.6.5.1.1, the only way to populate the Key Vault slot that holds the MEK is via the AES engine operating in ECB-decrypt mode (where the key is taken from the Key Vault slot that holds the MDK).
- The AES engine does not allow messages to be read from the Key Vault. Therefore KMB must derive a key from the MEK secret that is readable by firmware, so firmware can pass it as an AES message to compute the MEK.
- Since the MEK secret is in the Key Vault, the result of any HMAC operations would be required to go back to the Key Vault, and not be available to firmware.
- The CMAC operation, by contrast, utilizes the AES engine in a manner that allows the results to be readable by firmware. Therefore CMAC is used to compute the MEK seed from the MEK secret.
- The resulting MEK seed is then passed as an AES message to the AES engine, which uses it in a decryption operation to produce the derived MEK.

#### 4.6.5.3.2 Derived MEK checksums

When deriving an MEK, drive firmware provides an MEK checksum to KMB. See Figure 25 for details on how the checksum is derived. If the provided checksum is non-zero, it is compared with the calculated checksum. The operation fails (and the derived MEK is not sent to the encryption engine) if the provided checksum does not match the calculated checksum. This check is skipped if the provided checksum is all-zeroes.

Upon success, the calculated checksum is returned to drive firmware. Drive firmware may choose to store the checksum persistently and use it for future derivations, or may discard the

returned checksum. The checksum allows drive firmware to ensure that an MEK derived in the future matches a given MEK derived previously. This check would be a precaution against mistakes or bit flips that would result in an incorrect MEK being programmed to the encryption engine, which could lead to data loss. This specification does not define how drive firmware should react to a checksum mismatch.

The checksum is not derived from the MEK directly, because Caliptra hardware restricts what operations can be done with the Key Vault slot that holds the MEK.

An example flow where the MEK checksum would be useful is as follows:

1. Drive firmware is instructed to establish a new MEK.
2. Drive firmware instructs KMB to produce the new MEK via DERIVE_MEK, as part of the sequence specified in Section 4.6.5.4.
3. Upon success, drive firmware is given the derived-MEK's checksum, which it stores in non-volatile memory.
4. On a subsequent boot, drive firmware reads the SEK into volatile memory, and is later asked to re-establish the MEK it derived in step 2.
5. The copy of the SEK held in volatile memory has suffered corruption since it was last read from persistent storage, and a bit has flipped.
6. Drive firmware invokes DERIVE_MEK. As part of this command, drive firmware provides the derived-MEK checksum it saved in non-volatile storage.
7. KMB detects the checksum mismatch and refuses to program the derived MEK. Drive firmware surfaces this failure to the host.

In this example, if the user had proceeded to write data to the disk using the MEK derived in step 6, that data would not be recoverable on subsequent boots unless the SEK experienced the exact same bit flip pattern as in step 5. The derived-MEK checksum allows the drive to detect this scenario and alert the user. Note that there are other possible mechanisms for mitigating this risk. The use of derived-MEK checksums is optional.

Note: a checksum is not produced for wrapped MEKs. For wrapped MEKs, any change in the inputs which produce the MEK secret would result in an error during AES-GCM decryption of the provided MEK.

#### 4.6.5.4 MEK command sequence

KMB exposes the following mailbox commands for establishing an MEK:

- INITIALIZE_MEK_SECRET
- MIX_MPK
- GENERATE_MEK
- LOAD_MEK
- DERIVE_MEK

Figure 26 illustrates how these commands are used in sequence.

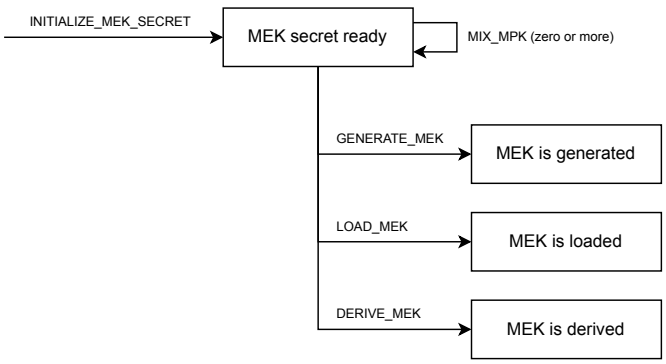



**Figure 26:** MEK command sequence

Integrators may elect to utilize either MEK generation or MEK derivation when establishing an MEK.

#### 4.6.5.5 AES-XTS considerations

The MEK sent to the encryption engine may be used as an AES-XTS key. FIPS 140-3 IG [19] 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 [11] 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 [10] section 4 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.

### 4.6.6 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 a TRNG and which may be updated with entropy from the host.

Caliptra hardware [20] supports either an integrated or external DRBG. Caliptra Subsystem requires leveraging the integrated DRBG. This construction is illustrated in Figure 27.

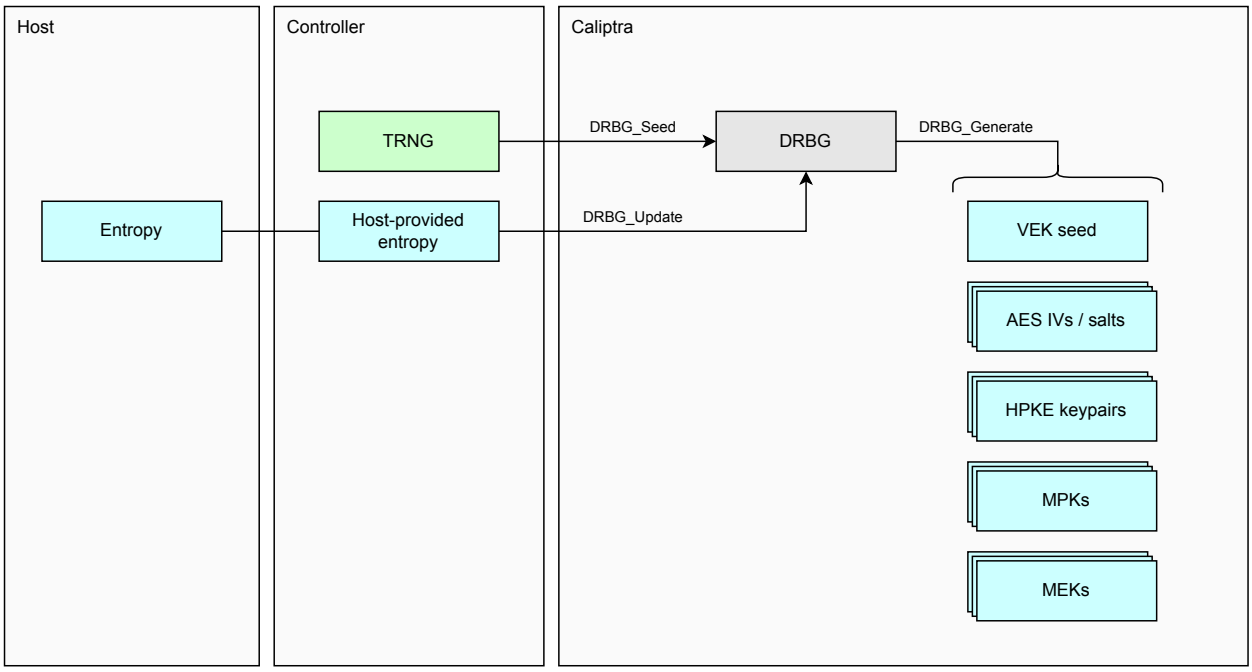



**Figure 27:** Integrated DRBG

Host-provided entropy can be mixed into the DRBG state via the CM_RANDOM_STIR Caliptra mailbox command.

### 4.6.7 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.

### 4.6.8 Reset behavior

This section defines expected behavior for KMB across each Caliptra reset type, as defined in [21].

**Table 5:** Behavior on Caliptra reset types

| Caliptra reset type | Behavior |
|---|---|

*(continued on next page)*

*(continued from previous page)*

| Caliptra reset type | Behavior |
|---|---|
| Cold boot / cold reset | Caliptra ROM derives the HEK and MDK, then sets the LOCK_IN_PROGRESS bit in the SS_OCP_LOCK_CTRL register.<br><br>Next, Caliptra runtime firmware performs the following steps:<br><br>• 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.<br><br>• Initializes random HPKE keypairs for each supported algorithm, and assigns handles to them, reported via ENUMERATE_HPKE_HANDLES.<br><br>The VEK is not initialized upon cold reset, as it is generated lazily upon first use. See Section 4.6.1.5 for details. |
| Warm reset | The HEK, MDK, VEK, and HPKE keypairs are undisturbed. |
| Firmware update reset | The HEK, MDK, and VEK are undisturbed.<br><br>HPKE keypairs are discarded prior to firmware update and are regenerated after update. |

Caliptra cold boot / cold reset shall only be triggered by an NVMe subsystem reset [22] caused by main power being applied to the subsystem.

A given integration may trigger a Caliptra warm reset in response to other reset events.

## 4.7 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.

***Note: as of this specification's publication, KMB firmware has not yet been implemented. As such, certain details of the interface specification are subject to change. This notice will be removed once firmware has been implemented.***

### 4.7.1 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.

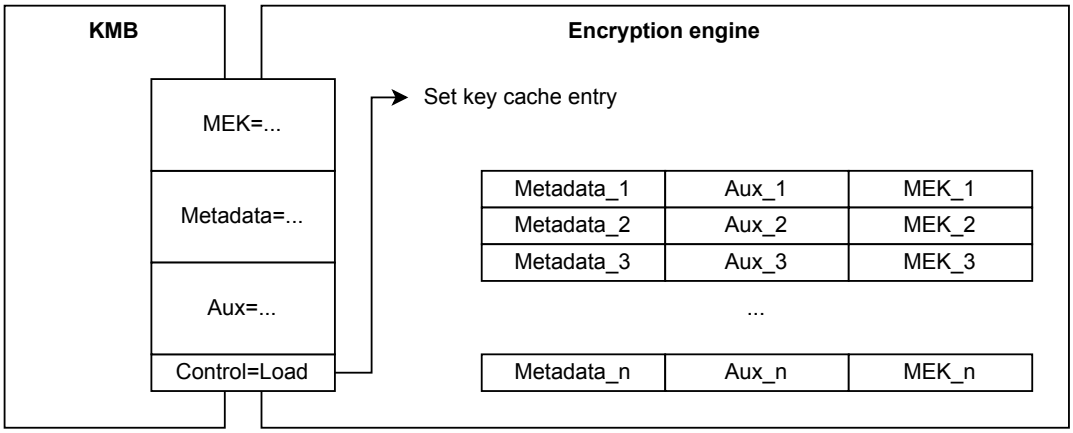
#### 4.7.1.1 Overview

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.

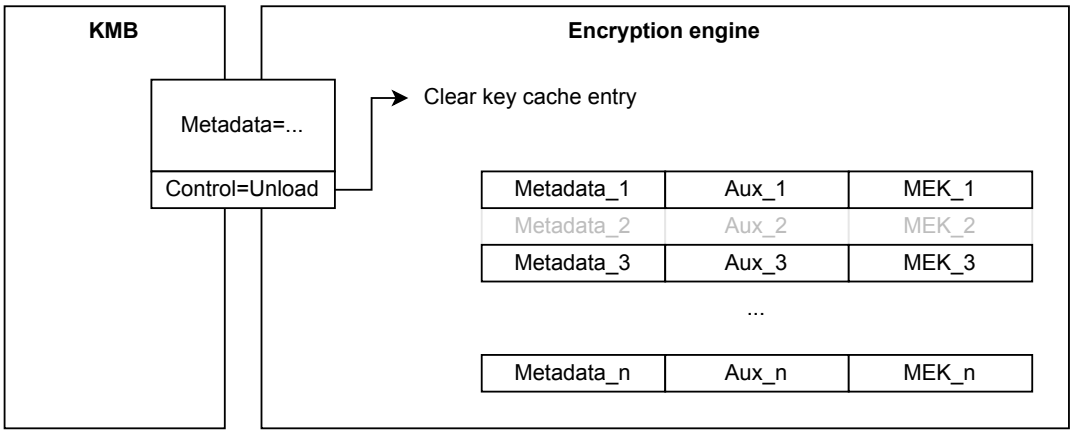
To ensure that MEKs are only ever visible to KMB and the encryption engine, KMB is the only entity which can load and unload (metadata, aux, MEK) tuples. Drive firmware arbitrates all operations in the KMB to encryption engine interface, and is therefore responsible for managing which MEK is loaded in the key cache. Drive firmware has full control on metadata and optional aux. Figure 28 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.

**Populate MEK into encryption engine**




**Remove MEK from encryption engine**




**Sanitize encryption engine**

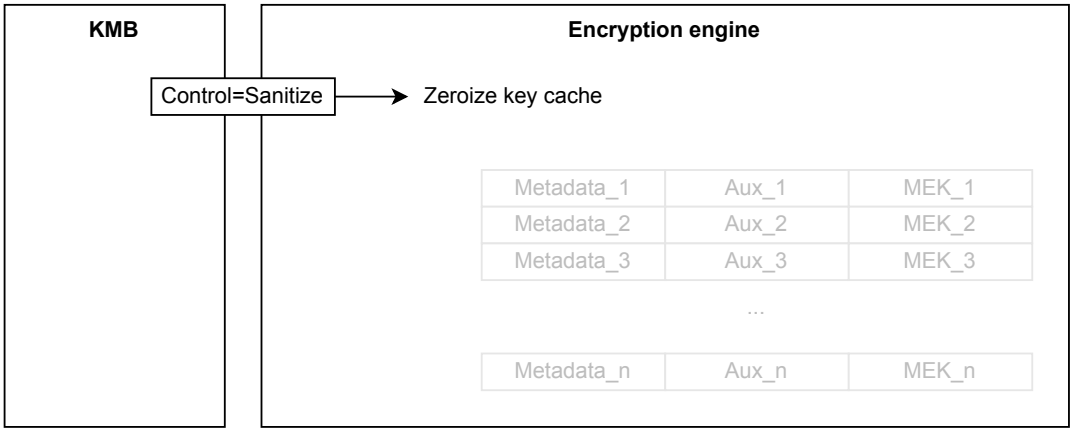



**Figure 28:** KMB to encryption engine SFR interface

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

#### 4.7.1.2 Special Function Registers

KMB uses Special Function Registers (SFRs) to communicate with the encryption engine as shown in Table 6 and each of the following subsections which describe the registers.

The OCP_LOCK_MEK_ADDRESS register represents the combined registers SS_KEY_RELEASE_BASE_ADDR_L and SS_KEY_RELEASE_BASE_ADDR_H and shall contain the fixed base address of the MEK register.

The SS_KEY_RELEASE_SIZE register shall contain the size of the MEK, in bytes. This shall be set to 40h by the MCU ROM.

MCU ROM is responsible for configuring SS_KEY_RELEASE_BASE_ADDR_L, SS_KEY_RELEASE_BASE_ADDR_H and SS_KEY_RELEASE_SIZE, prior to setting CPTRA_FUSE_WR_DONE to prevent further modifications.

**Table 6:** KMB to encryption engine SFRs

| Register | Address | Byte Size | Description |
|---|---|---|---|
| Media Encryption Key (MEK) | OCP_LOCK_MEK_ADDRESS | 40h | Register to provide MEK. |
| Metadata (METD) | OCP_LOCK_MEK_ADDRESS + 40h | 14h | Register to provide metadata |
| Auxiliary Data (AUX) | OCP_LOCK_MEK_ADDRESS + 60h | 20h | Register to provide auxiliary values |
| Control | OCP_LOCK_MEK_ADDRESS + 80h | 4h | Register to handle commands |

OCP_LOCK_MEK_ADDRESS contains the base address for the SFRs shown in Table 6. The vendor is responsible for ensuring that KMB can access these SFRs through these addresses.

For alignment, offsets OCP_LOCK_MEK_ADDRESS + OCP_LOCK_MEK_LENGTH + 14h..20h (inclusive) are intentionally unallocated.

In the register definitions that follow, the “Type” column indicates whether KMB may read from or write to the given register.

##### 4.7.1.2.1 Media Encryption Key register

**Table 7:** OCP_LOCK_MEK_ADDRESS: 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 29.

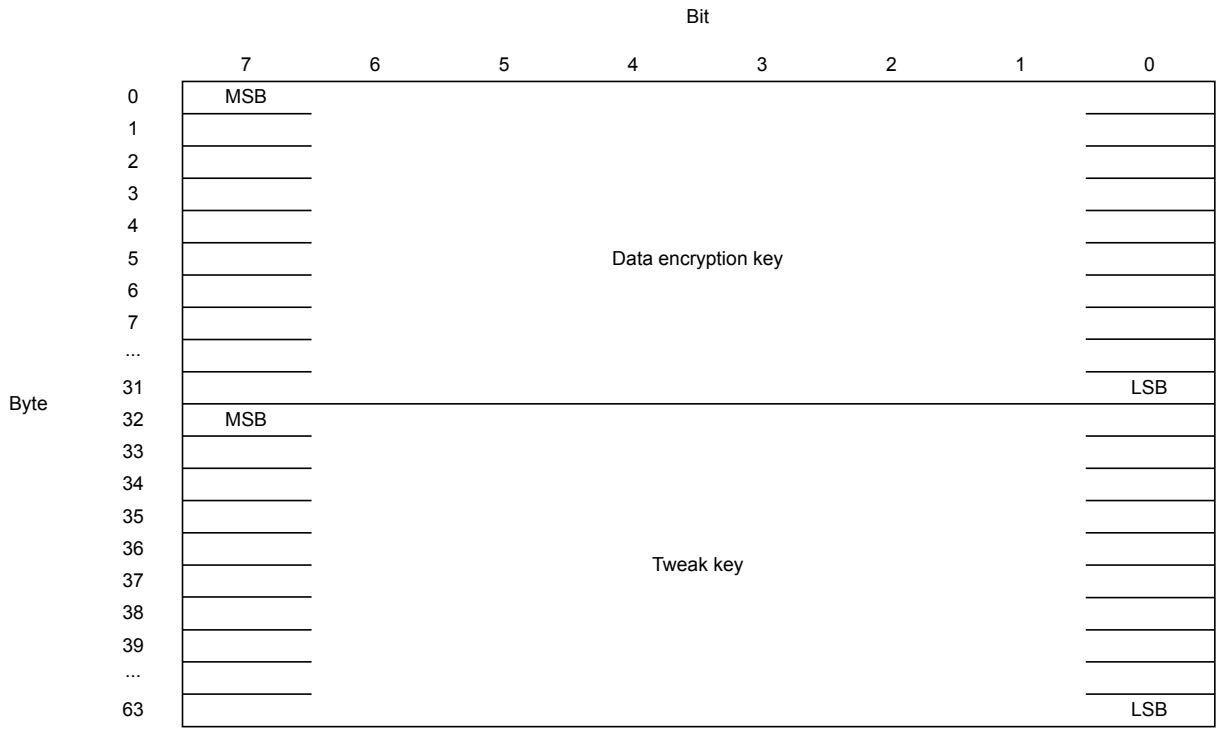



**Figure 29:** MEK format example for AES-XTS-256

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. The DMA engine will copy the key value stored in Key Vault slot 23 to the destination address, specified by OCP_LOCK_MEK_ADDRESS.

#### 4.7.1.2.2 Metadata register

Table 8 defines the Metadata register used to pass additional data related to the MEK.

**Table 8:** Offset OCP_LOCK_MEK_ADDRESS + 40h: 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 non-secret metadata, in order for the MEK to be used for any key-related operations including data I/O. The **METD** field is used to convey this metadata.

When loading or deriving an MEK, the KMB takes an **METD** value as input from drive firmware and writes it to the **METD** register without any modification. This allows the vendor full control of the MEK indexing algorithm.

Two examples of how a storage device might leverage the **METD** field are provided: 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 an (LBA range, Namespace ID) pair. Then, the (LBA range, Namespace ID) pair can be leveraged into **METD** as illustrated in Figure 30.

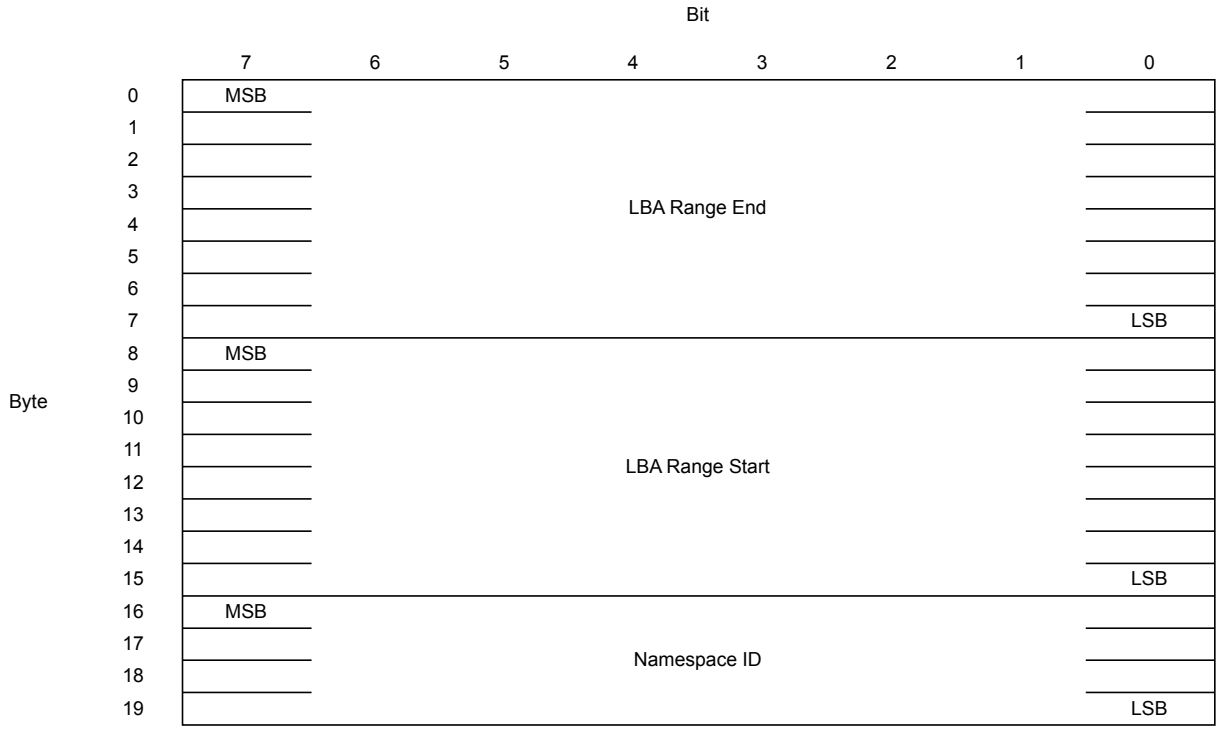



**Figure 30:** LBA range based metadata format

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

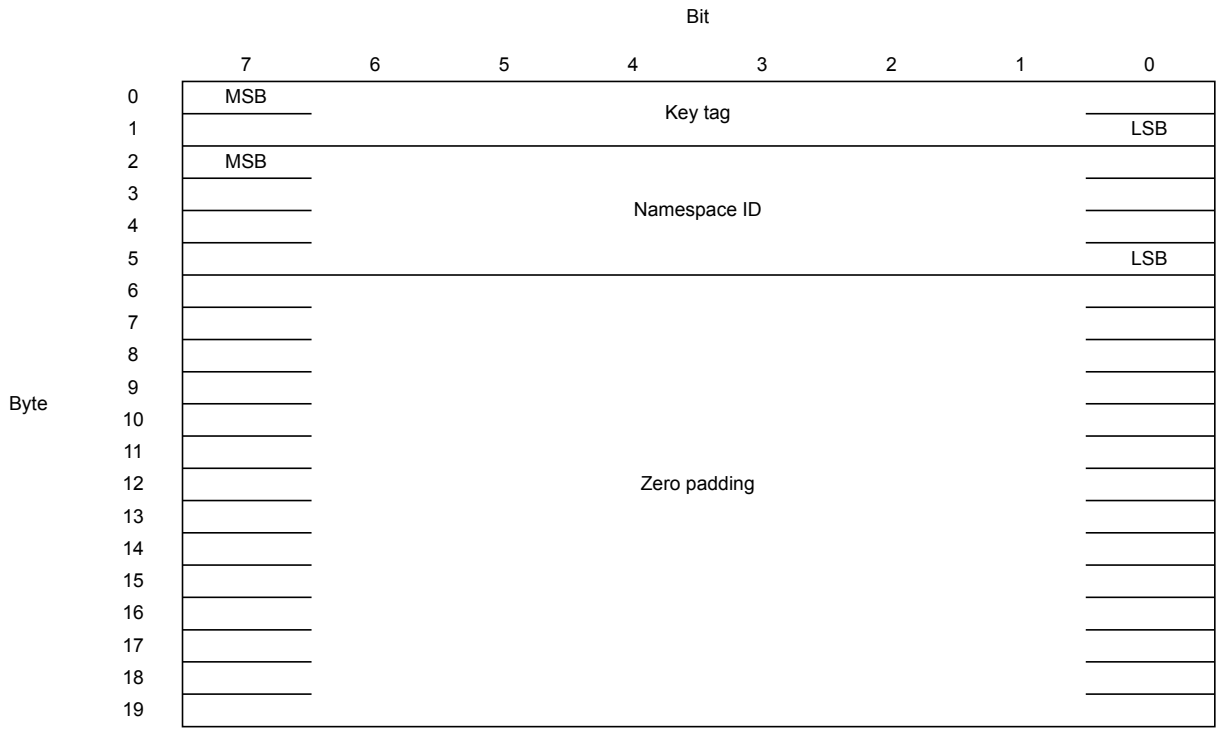



**Figure 31:** Key tag based metadata format

The above examples are not the only possible values of **METD**. Vendors may design and use their own **METD** if it is more suitable for their system.

#### 4.7.1.2.3 Auxiliary Data register

Table 9 defines the Auxiliary Data register used to pass additional vendor-specific data related to the MEK.

**Table 9:** OCP_LOCK_MEK_ADDRESS + 60h: 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 32 can be used.

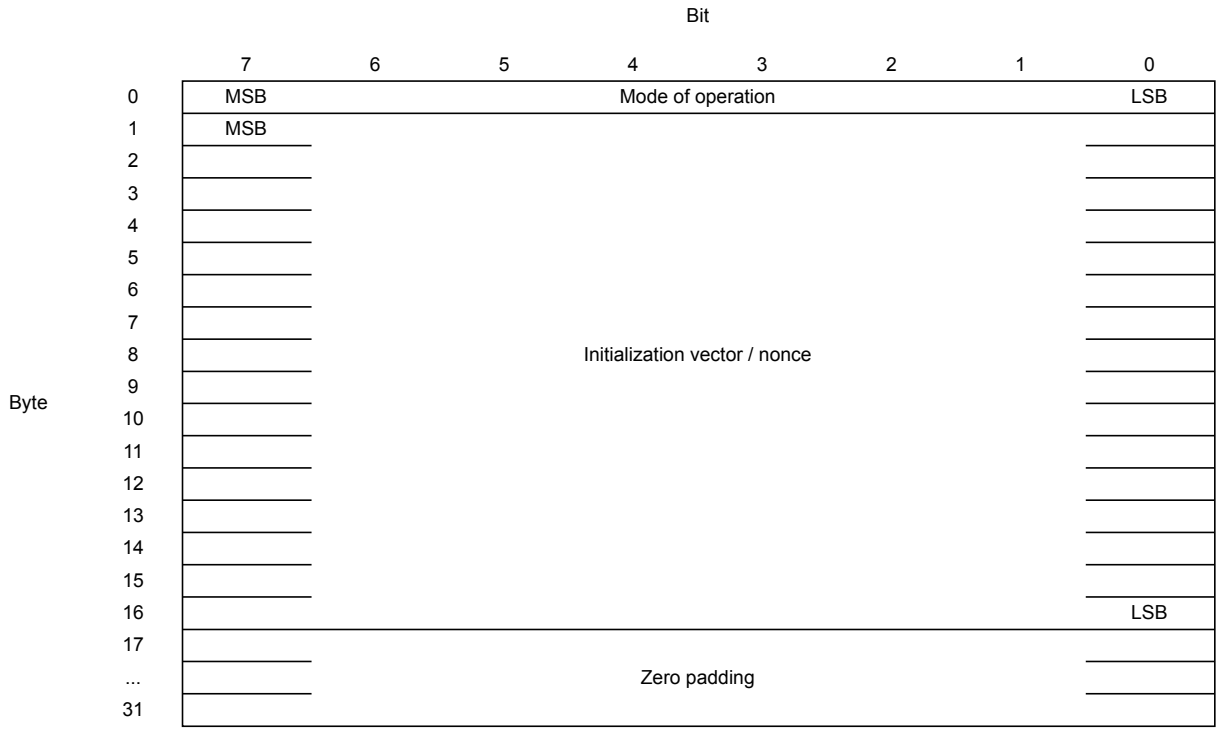



**Figure 32:** Auxiliary data format example

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.

#### 4.7.1.2.4 Control register

Table 10 defines the Control register used to sequence the execution of a command and obtain the status of that command.

**Table 10:** Offset OCP_LOCK_MEK_ADDRESS + 80h: CTRL – Control

| Bits | Type | Reset | Description |
|---|---|---|---|

*(continued on next page)*

*(continued from previous page)*

| Bits | Type | Reset | Description |
|---|---|---|---|
| 31 | RO | 0h | **Ready (RDY):** After any reset, the default value of this bit shall be 0b. Once the encryption engine is initialized and becomes ready to handle commands, the encryption engine shall set this bit from 0b to 1b. When a fatal error has occurred in the encryption engine, the encryption engine may set this bit from 1b to 0b.<br><br>KMB checks that this bit is set to 1b before issuing any commands to the encryption engine. |
| 30:20 | RO | 0h | Reserved |
| 19:16 | RO | 0h | **Error (ERR):** When the encryption engine sets the DONE bit from 0b 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 11.<br><br>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 12. |
| 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 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:<br><br>• set the DONE bit to 0b,<br>• set the EXE bit to 0b,<br>• set the ERR field to 0000b, and<br>• set the CMD bit to 0000b. |
| 0 | RW | 0b | **Execute (EXE):** After any reset, the default value of this field shall be 0b. When this bit is set from 0b to 1b by KMB, the encryption engine shall execute the command specified in the CMD field. Once the execution is complete, the encryption engine shall simultaneously set this bit from 1b to 0b, set the DONE bit from 0b to 1b, and populate the ERR field. While the EXE bit is 1b, the encryption engine is busy. |

**Table 11:** CTRL error codes

| Value | Description |
|---|---|
| 0h | Command successful |
| 1h | Invalid command code |
| 2h to 3h | Reserved |
| 4h to Fh | Vendor Specific |

**Table 12:** 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.

#### 4.7.1.3 Control register state machine

Figure 33 illustrates the control register's state machine as it transitions from power-on to executing a command.

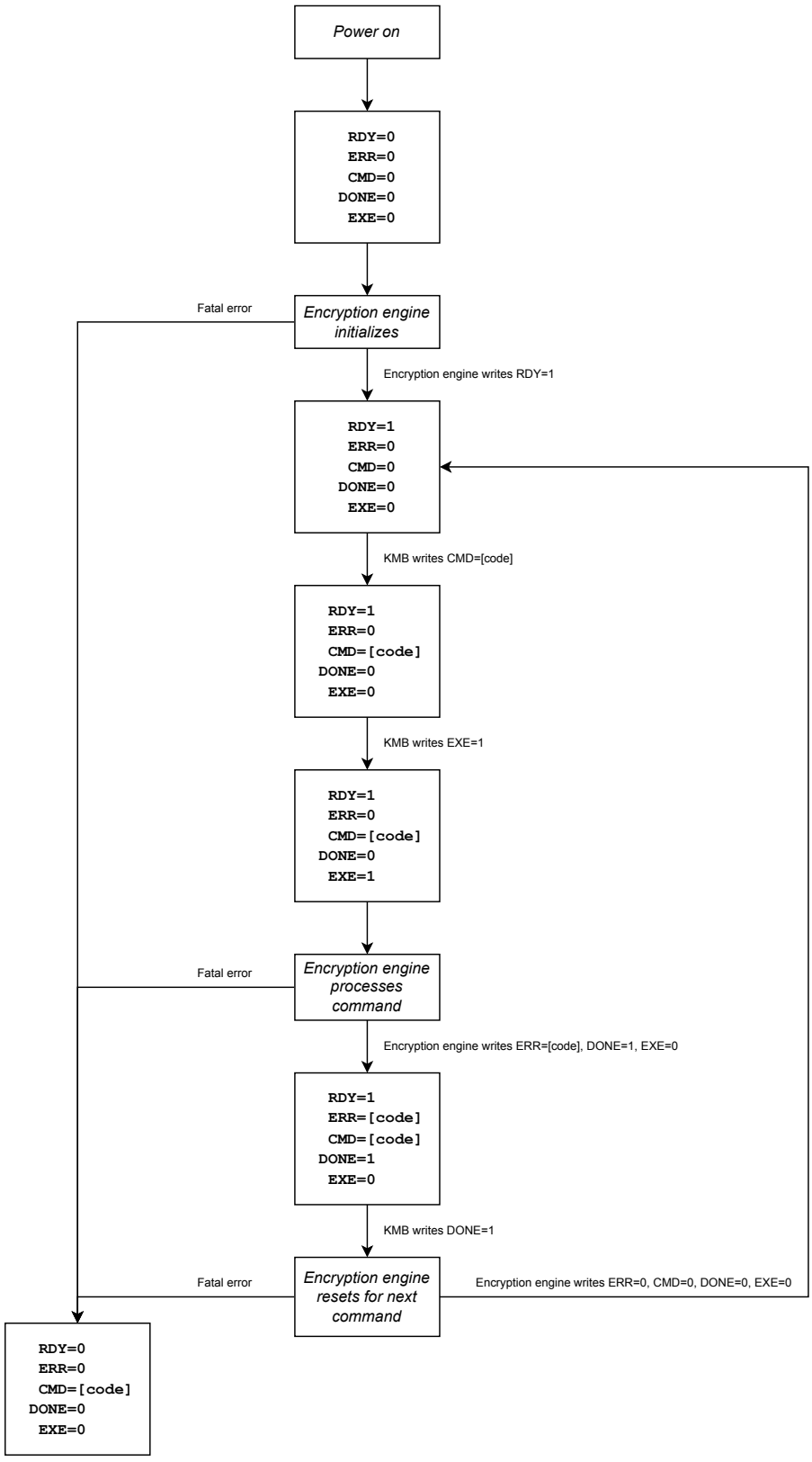



**Figure 33:** Control register state machine

#### 4.7.1.4 KMB command sequence

Figure 34 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.

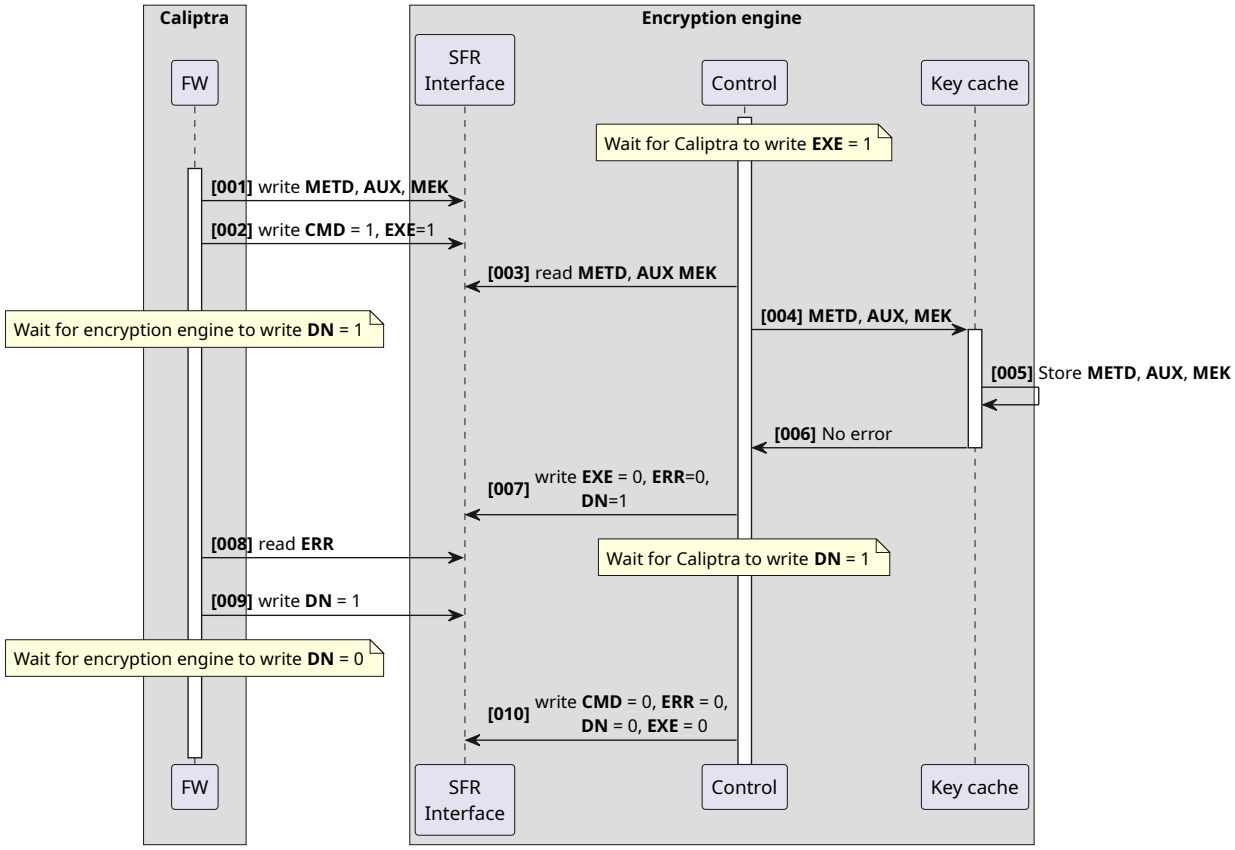



**Figure 34:** Command execution example for loading an MEK

### 4.7.2 Mailbox interface

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

#### 4.7.2.1 FIPS status indicator

Each mailbox command returns a `fips_status` field. This provides an indicator of whether KMB is operating in FIPS mode. Table 13 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 13:** Values for the FIPS status field

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

#### 4.7.2.2 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.

#### 4.7.2.3 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.

#### 4.7.2.4 REPORT_HEK_METADATA

This command is exposed by Caliptra ROM and allows MCU ROM to report metadata about the HEK. This command must be called exactly once after each cold reset and before loading Caliptra's FMC and runtime firmware.

By exposing this command in ROM, Caliptra de-privileges runtime compromise of drive firmware that might cause it to mis-report the HEK seed state, which could lead to incorrect device state attestations.

Command Code: 0x5248_4D54 ("RHMT")

**Table 14:** REPORT_HEK_METADATA input arguments

| Name | Type | Description |
|---|---|---|
| chksum | u32 | Checksum over other input arguments, computed by the caller. |
| reserved | u32 | Reserved. |
| total_slots | u16 | Total number of HEK seed slots. |
| active_slot | u16 | Currently-active HEK seed slot. Zero-indexed. |
| seed_state | u16 | HEK seed state. See Table 15. |
| padding | u16 | Reserved. |

**Table 15:** HEK seed state values

| Value | Description |
|---|---|
| 0h | HEK_SEED_UNAVAIL_EMPTY |
| 1h | HEK_SEED_UNAVAIL_ZEROIZED |
| 2h | HEK_SEED_UNAVAIL_CORRUPTED |
| 3h | HEK_SEED_AVAIL_PROGRAMMED |
| 4h | HEK_SEED_AVAIL_UNERASABLE |
| 5h to FFFFh | Reserved |

MCU ROM shall adhere to Table 16 when populating the HEK metadata.

**Table 16:** Conditions for setting seed_state and active_slot

| Condition | Value of seed_state | Value of active_slot |
|---|---|---|
| All HEK seed slots are blank. | HEK_SEED_UNAVAIL_EMPTY | 0 |
| One or more HEK seed slots are zeroized, and either the next seed slot is blank, or there are no remaining seed slots. | HEK_SEED_UNAVAIL_ZEROIZED | The last slot that was zeroized. |
| A HEK seed slot has been corrupted by an interrupted write. | HEK_SEED_UNAVAIL_CORRUPTED | The slot that is corrupted. |
| A HEK seed slot has been programmed with randomness. | HEK_SEED_AVAIL_PROGRAMMED | The slot that is programmed with randomness. |
| All HEK seed slots have been zeroized and the perma-HEK mode bit has been programmed. | HEK_SEED_AVAIL_UNERASABLE | The last slot that was zeroized. |

**Table 17:** REPORT_HEK_METADATA 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. |

#### 4.7.2.5 GET_STATUS

Allows drive firmware to determine if the encryption engine is ready to process commands as well as vendor-defined drive encryption engine status data.

Command Code: 0x4753_5441 (“GSTA”)

**Table 18:** GET_STATUS input arguments

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

**Table 19:** 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. |

#### 4.7.2.6 GET_ALGORITHMS

Allows drive firmware to determine the types of algorithms supported by KMB for endorsement, KEM, MPK, and access key generation.

Command Code: 0x4741_4C47 (“GALG”)

**Table 20:** GET_ALGORITHMS input arguments

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

**Table 21:** 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:<br>• Byte 0 bit 0: ecdsa_secp384r1_sha384 [23]<br>• Byte 0 bit 1: ml-dsa-87 [24] |
| hpke_algorithms | u32 | Identifies the supported HPKE algorithms: {kem/aead/kdf}_id<br>• Byte 0 bit 0: 0x0011, 0x0002, 0x0002 [9]<br>• Byte 0 bit 1: 0x0042, 0x0002, 0x0002 [15]<br>• Byte 0 bit 2: 0x0052, 0x0002, 0x0002 [15]<br>See Table 3 for definitions of each HPKE algorithm identifier. |

*(continued on next page)*

*(continued from previous page)*

| Name | Type | Description |
|---|---|---|
| access_key_sizes | u32 | Indicates the length of plaintext access keys:<br>• 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.

#### 4.7.2.7 CLEAR_KEY_CACHE

This command unloads all MEKs in the encryption engine.

Command Code: 0x434C_4B43 (“CLKC”)

**Table 22:** 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 23:** CLEAR_KEY_CACHE 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. |

#### 4.7.2.8 ENUMERATE_HPKE_HANDLES

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

Command Code: 0x4548_444C (“EHDL”)

**Table 24:** ENUMERATE_HPKE_HANDLES input arguments

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

**Table 25:** ENUMERATE_HPKE_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. |
| hpke_handle_count | u32 | Number of HPKE handles (N). |
| hpke_handles | HpkeHandle[N] | List of (HPKE handle value, HPKE algorithm) tuples. |

**Table 26:** HpkeHandle contents

| Name | Type | Description |
|---|---|---|
| handle | u32 | Handle for HPKE keypair held in KMB memory. |
| hpke_algorithm | u32 | HPKE algorithm. Shall be a bit value indicated as supported in Table 21. |

#### 4.7.2.9 ENDORSE_HPKE_PUB_KEY

This command generates a signed certificate for the specified HPKE public key using the specified endorsement algorithm.

Command Code: 0x4548_504B (“EHPK”)

**Table 27:** ENDORSE_HPKE_PUB_KEY input arguments

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

**Table 28:** ENDORSE_HPKE_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). |

*(continued on next page)*

*(continued from previous page)*

| Name | Type | Description |
|---|---|---|
| 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. |

#### 4.7.2.10 ROTATE_HPKE_KEY

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

Command Code: 0x5248_504B (“RHPK”)

**Table 29:** ROTATE_HPKE_KEY input arguments

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

**Table 30:** ROTATE_HPKE_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. |
| hpke_handle | u32 | Handle for new HPKE keypair held in KMB memory. |

#### 4.7.2.11 GENERATE_MPK

This command unwraps the specified access key, generates a random MPK, then encrypts the MPK with a Locked MPK encryption key. The Locked MPK 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.

Command Code: 0x474D_504B (“GMPK”)

**Table 31:** GENERATE_MPK input arguments

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

*(continued on next page)*

*(continued from previous page)*

| Name | Type | Description |
|---|---|---|
| sek | u8[32] | Soft Epoch Key. |
| metadata_len | u32 | Length of the metadata argument. |
| metadata | u8[metadata_len] | Metadata for the MPK. |
| sealed_access_key | SealedAccessKey | HPKE-sealed access key. |

**Table 32:** 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. |

#### 4.7.2.12 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 its current Locked MPK encryption key, then re-encrypted with its new Locked MPK encryption key. The new Locked MPK is returned.

The drive stores the returned Locked MPK and zeroizes the old Locked MPK.

Command Code: 0x5245_5750 (“REWP”)

**Table 33:** 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. |
| current_locked_mpk | LockedMpk | Current MPK to be rewrapped. |
| sealed_access_key | SealedAccessKey | HPKE-sealed current access key. |
| new_ak_ciphertext | u8[access_key_len+Nt] | New access key ciphertext and authentication tag. `access_key_len` is provided by the SealedAccessKey. `Nt` is the HPKE value associated with the `aead_id` identifier from the SealedAccessKey’s `hpke_algorithm`. See Section 4.7.2.22.1 for details. |

**Table 34:** 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. |

#### 4.7.2.13 ENABLE_MPK

This command decrypts `sealed_access_key`, then uses it to derive a Locked MPK encryption key, which is used to decrypt `locked_mpk`. The MPK is then re-encrypted with the VEK. The Enabled MPK is returned.

Command Code: 0x524D_504B (“RMPK”)

**Table 35:** ENABLE_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. |
| sealed_access_key | SealedAccessKey | HPKE-sealed access key. |
| locked_mpk | LockedMpk | MPK encrypted to the HEK and access key. |

**Table 36:** ENABLE_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. |
| enabled_mpk | EnabledMpk | MPK encrypted to the VEK. |

#### 4.7.2.14 INITIALIZE_MEK_SECRET

This command shall initialize the MEK secret seed as described in Figure 20.

Command Code: 0X494D_0B53 (“IMKS”)

**Table 37:** INITIALIZE_MEK_SECRET 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 38:** INITIALIZE_MEK_SECRET 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. |

#### 4.7.2.15 MIX_MPK

This command decrypts the specified MPK with the VEK, 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.

The MEK secret seed must already be initialized by INITIALIZE_MEK_SECRET or this command shall return an error.

Command Code: 0x4D4D_504B (“MMPK”)

**Table 39:** MIX_MPK input arguments

| Name | Type | Description |
|---|---|---|
| chksum | u32 | Checksum over other input arguments, computed by the caller. |
| reserved | u32 | Reserved. |
| enabled_mpk | EnabledMpk | MPK encrypted to the VEK. |

**Table 40:** 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. |

#### 4.7.2.16 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.

Command Code: 0x5441_434B ("TACK")

**Table 41:** TEST_ACCESS_KEY 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. |
| 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 42:** 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. |

#### 4.7.2.17 GENERATE_MEK

This command generates a random 512-bit MEK and encrypts it using the MEK secret.

INITIALIZE_MEK_SECRET must be invoked prior to this command or an error shall be returned.

Command Code: 0x474D_454B ("GMEK")

**Table 43:** GENERATE_MEK input arguments

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

**Table 44:** 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 secret. |

#### 4.7.2.18 LOAD_MEK

This command decrypts the given encrypted 512-bit MEK using the MEK secret.

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.

INITIALIZE_MEK_SECRET must be invoked prior to this command or an error shall be returned.

Command Code: 0x4C4D_454B (“LMEK”)

**Table 45:** LOAD_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 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 secret. |
| cmd_timeout | u32 | Timeout in ms for command to encryption engine to complete. |

**Table 46:** 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. |

#### 4.7.2.19 DERIVE_MEK

This command derives an MEK using the MEK secret.

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.

INITIALIZE_MEK_SECRET must be invoked prior to this command or an error shall be returned.

Command Code: 0x444D_454B (“DMEK”)

**Table 47:** DERIVE_MEK input arguments

| Name | Type | Description |
|---|---|---|
| chksum | u32 | Checksum over other input arguments, computed by the caller. |
| reserved | u32 | Reserved. |
| mek_checksum | u8[16] | Checksum used to validate the derived MEK. May be all-zeroes, in which case no check is performed. |
| 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 48:** 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. |
| reserved | u32 | Reserved. |
| mek_checksum | u8[16] | Checksum calculated for the derived MEK. |

#### 4.7.2.20 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.

Command Code: 0x554D_454B (“UMEK”)

**Table 49:** 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 50:** 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. |

#### 4.7.2.21 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 51:** 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 53. |
| padding | u16 | Reserved. |
| nonce | u8[16] | Freshness nonce to be included in the signed IETF EAT. |

**Table 52:** 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. |
| hek_erasures_remaining | u16 | Remaining number of times the HEK may be erased.<br>See Section 4.7.2.21.1 for how this field is calculated. |
| hek_state | u16 | State of the currently-active HEK. See Table 54. |
| sek_state | u16 | SEK state from the input argument. |
| eat_len | u16 | Total length of the IETF EAT. |
| nonce | u8[16] | Nonce from the input argument. |
| eat | u8[eat_len] | CBOR-encoded and signed IETF EAT. See Section B for the format. |

**Table 53:** SEK state values

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

The value reported from Table 54 depends on Caliptra's current lifecycle state as well as the `seed_state` value from REPORT_HEK_METADATA.

**Table 54:** HEK state values

| Value | State | Description |
|---|---|---|
| 0h | HEK_UNAVAIL_EMPTY | The drive has transitioned to the Production life cycle state and a HEK has not yet been provisioned.<br>Reported if `seed_state` is HEK_SEED_UNAVAIL_EMPTY. |
| 1h | HEK_UNAVAIL_ZEROIZED | The current HEK has been zeroized.<br>Reported if `seed_state` is HEK_SEED_UNAVAIL_ZEROIZED. |
| 2h | HEK_UNAVAIL_CORRUPTED | The current HEK slot is corrupted.<br>Reported if `seed_state` is HEK_SEED_UNAVAIL_CORRUPTED. |
| 3h | HEK_AVAIL_PROGRAMMED | A randomized HEK is provisioned in the current slot.<br>Reported if `seed_state` is HEK_SEED_AVAIL_PROGRAMMED. |
| 4h | HEK_AVAIL_UNERASABLE | The HEK is not used to provide sanitization capabilities. In this state, data written to the storage device cannot be cryptographically erased via HEK zeroization.<br>Reported if Caliptra is not in the production lifecycle state or if `seed_state` is HEK_SEED_AVAIL_UNERASABLE.<br>KMB internally derives a HEK in this state, but the HEK is derived from non-erasable secrets. |
| 5h to FFFFh | Reserved | Reserved. |

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

See Table 4 for more information about the relationship between HEKs and Caliptra lifecycle states.

HEK_AVAIL_UNERASABLE → 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_UNERASABLE is a one-way transition that the device owner can trigger once all HEK slots have been zeroized.

See Figure 19 for more details about the transitions between HEK states.

#### 4.7.2.21.1 Calculation of hek_erasures_remaining

The `hek_erasures_remaining` field in REPORT_EPOCH_KEY_STATE is based on the following values from REPORT_HEK_METADATA:

- total_slots
- active_slot
- seed_state

The calculation is as follows:

$$\text{seed\_is\_erased} = \begin{cases} true & \text{if seed\_state} = \text{HEK\_SEED\_UNAVAIL\_ZEROIZED} \\ true & \text{if seed\_state} = \text{HEK\_SEED\_AVAIL\_UNERASABLE} \\ false & \text{otherwise} \end{cases}$$

$$\text{hek\_erasures\_remaining} = \text{total\_slots} - \text{active\_slot} - \begin{cases} 1 & \text{if seed\_is\_erased} \\ 0 & \text{otherwise} \end{cases}$$

### 4.7.2.22 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.

#### 4.7.2.22.1 SealedAccessKey type

This type holds an HPKE-wrapped access key.

**Table 55:** SealedAccessKey contents

| Name | Type | Description |
|---|---|---|
| hpke_handle | u32 | Handle for HPKE keypair held in KMB memory. |
| hpke_algorithm | u32 | HPKE algorithm. Must be a bit value indicated as supported in Table 21. |
| access_key_len | u32 | Access key length in bytes. Must be the scalar value associated with a bit value indicated as supported in Table 21. |
| info_len | u32 | Length of the info field. |
| info | u8[info_len] | Info to use with HPKE unwrap. |
| kem_ciphertext | u8[Nenc] | HPKE encapsulated key. |
| ak_ciphertext | 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 [9], `Nenc` and `Nt` would be 97 and 16, respectively.

#### 4.7.2.22.2 WrappedKey type

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

**Table 56:** WrappedKey contents

| Name | Type | Description |
|---|---|---|
| key_type | u16 | Type of the wrapped key.<br>• 0h: Reserved<br>• 1h: LOCKED_MPK (ciphertext held at rest)<br>• 2h: ENABLED_MPK (ciphertext held in RAM)<br>• 3h: WRAPPED_MEK<br>• 4h to FFFFh: Reserved |
| reserved | u16 | Reserved. |
| salt | u8[12] | Random salt for the given wrapped key. |
| metadata_len | u32 | Length of the metadata field. |
| key_len | u32 | Length of the encrypted key. |
| iv | u8[12] | Initialization vector for AES operation. |
| metadata | u8[metadata_len] | Metadata associated with the wrapped key. |
| ciphertext | u8[key_len+16] | Key ciphertext and authentication tag. |

The AAD for the encrypted message is constructed as `key_type` || `salt` || `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 `key_len`:

**Table 57:** WrappedKey variants

| Name | key_type | key_len |
|---|---|---|
| LockedMpk | LOCKED_MPK | 32 |
| EnabledMpk | ENABLED_MPK | 32 |
| WrappedMek | WRAPPED_MEK | 64 |

#### 4.7.2.23 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.

KMB mailbox errors should generally result in an error being surfaced to the host.

**Table 58:** 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_ERR + u8 | 0x4C45_52xx (“LERx”) | The low byte indicates the error code and ready state.<br>• Bit 0: RDY field of the encryption engine’s CTRL register.<br>• Bits [3:1]: reserved<br>• Bits [7:4]: ERR field of the encryption engine’s CTRL register. See Table 11 for an enumeration of code values. |
| 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_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_MEK_CHKSUM_FAIL | 0x4C4D_4346 (“LMCF”) | Error during MEK derivation due to checksum mismatch. |
| LOCK_HEK_NOT_AVAILABLE | 0x4C48_4E41 (“LHNA”) | The operation requires the HEK, which is unavailable. |
| LOCK_MEK_NOT_INITIALIZED | 0x4C4D_4E49 (“LMNI”) | MIX_MPK, GENERATE_MEK, LOAD_MEK, or DERIVE_MEK were called without first invoking INITIALIZE_MEK_SECRET. |

## 4.8 Host APIs

In addition to supporting TCG Opal, Enterprise, and Key Per I/O, OCP L.O.C.K. provides underlying support for the Media Encryption Key Multiparty Authorization (MEK-MPA) Opal feature-set [18]. These bindings are described in Table 59.

**Table 59:** Mapping between KMB mailbox commands and host-facing storage APIs

| KMB mailbox command | Related host-facing API | Description |
|---|---|---|
| ENUMERATE_HPKE_HANDLES | TCG MEK-MPA | The HPKE public key certificates reported in the Certificate table are enumerated internally by this command. |
| ENDORSE_HPKE_PUB_KEY | TCG MEK-MPA | The HPKE public key certificates reported in the Certificate table are produced by this command. |
| ROTATE_HPKE_KEY | TCG MEK-MPA | When GenKey is invoked on an HPKE Certificate table entry, the key is rotated via this command. |
| GENERATE_MPK | TCG MEK-MPA | When an AccessCondition is initialized, via InitializeAccessCondition, its key material is produced by this command.<br><br>The MPK's metadata must be set to the AccessCondition's UID. The SealedAccessKey's `info` must be set to the concatenation of the hex value of the InitializeAccessCondition method UID and the hex value of the target AccessCondition object UID. |
| REWRAP_MPK | TCG MEK-MPA | When an AccessCondition's access key is changed via ChangeAccessKey, the rotation is accomplished via this command.<br><br>The SealedAccessKey's `info` must be set to the concatenation of the hex value of the ChangeAccessKey method UID and the hex value of the target AccessCondition object UID. |
| ENABLE_MPK | TCG MEK-MPA | When an AccessCondition is enabled via EnableAccess, that action is performed by this command.<br><br>The SealedAccessKey's `info` must be set to the concatenation of the hex value of the EnableAccess method UID and the hex value of the target AccessCondition object UID. |

*(continued on next page)*

*(continued from previous page)*

| KMB mailbox command | Related host-facing API | Description |
|---|---|---|
| TEST_ACCESS_KEY | TCG MEK-MPA | When an AccessCondition's access key is tested via TestAccessKey, that action is performed by this command.<br><br>The SealedAccessKey's `info` must be set to the concatenation of the hex value of the TestAccessKey method UID and the hex value of the target AccessCondition object UID. |
| MIX_MPK | TCG MEK-MPA | When a K_AES_* entry is created or unlocked, and that entry is bound to an AccessCondition, the K_AES_* entry is bound to the AccessCondition via this command. |
| GENERATE_MEK | TCG Opal or Enterprise | When a K_AES_* entry is created, that action may be performed by this command. |
| LOAD_MEK | TCG Opal or Enterprise | When a K_AES_* entry is unlocked, that action may be performed by this command. |
| DERIVE_MEK | TCG Opal or Enterprise | When a K_AES_* entry is unlocked, that action may be performed by this command. |
| DERIVE_MEK | TCG Key Per I/O | When a key tag with an associated DEK is injected, the MEK is derived via this command. |

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

## 4.9 Terminology

**Table 60:** Acronyms and abbreviations used throughout this document

| Abbreviation | Description |
|---|---|
| AES | Advanced Encryption Standard |
| CSP | Critical Security Parameter |
| DPK | Data Protection Key |
| DICE | Device Identifier Composition Engine |
| DRBG | Deterministic Random Bit Generator |

*(continued on next page)*

*(continued from previous page)*

| Abbreviation | Description |
|---|---|
| ECDH | Elliptic-curve Diffie–Hellman |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| EPK | Epoch Protection Key |
| HEK | Hard Epoch Key |
| 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 |
| MDK | MEK Deobfuscation Key |
| MEK | Media Encryption Key |
| ML-DSA | Module-Lattice-Based Digital Signature Algorithm |
| ML-KEM | Module-Lattice-Based Key-Encapsulation Mechanism |
| MPK | Multi-party Protection Key |
| NIST | National Institute of Standards and Technology |
| OCP | Open Compute Project |
| RTL | Register Transfer Level |
| SED | Self-encrypting drive |
| SEK | Soft Epoch Key |
| SSD | Solid-state drive |
| TCG | Trusted Computing Group |
| TRNG | True Random Number Generator |
| UART | Universal asynchronous receiver-transmitter |
| VEK | Volatile Escrow Key |
| XTS | XEX-based tweaked-codebook mode with ciphertext stealing |

## 4.10 Compliance

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

**Table 61:** Compliance requirements

| Item | Requirement | Mandatory |
|---|---|---|
| 1 | The device shall integrate Caliptra. | Yes |

*(continued on next page)*

*(continued from previous page)*

| Item | Requirement | Mandatory |
|---|---|---|
| 2 | OCP L.O.C.K. shall be enabled by setting the input strap `ss_ocp_lock_en` in Caliptra Core or `cptra_ss_strap_ocp_lock_en_i` in Caliptra Subsystem. | Yes |
| 3 | When a storage device is in a production lifecycle state, MEKs shall only be programmable to the encryption engine via Caliptra. See Section 4.6. | 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 4.6. | Yes |
| 5 | When the device is shipped it must be in the production lifecycle state and have a randomized HEK. See Section 4.6.4.2. | Yes |
| 6 | MCU ROM shall populate the HEK seed fuse register and invoke REPORT_HEK_METADATA upon cold reset. See Sections 4.6.4.1 and 4.7.2.4. | Yes |
| 7 | HEK and SEK programming/zeroization shall adhere to the sequence described in Section 4.6.4.2. | Yes |
| 8 | MEKs shall not be programmed while the SEK or HEK are zeroized. See Section 4.6.4. | Yes |
| 9 | The drive shall only provide HEK seeds or SEKs that are cryptographically-strong random values. See Section 4.6.4. | Yes |
| 10 | Drive firmware shall implement authorization controls to gate lifecycle events that have the potential to trigger data loss. See Section 4.6.7. | Yes |
| 11 | The encryption engine shall ensure that AES-XTS Key_1 and Key_2 are not equal. See Section 4.6.5.5. | Yes |
| 12 | KMB shall have access to the SFRs defined in Table 6 through the address defined by OCP_LOCK_MEK_ADDRESS. See Section 4.7.1.2. | Yes |
| 13 | MCU ROM shall configure registers SS_KEY_RELEASE_BASE_ADDR_L, SS_KEY_RELEASE_BASE_ADDR_H and SS_KEY_RELEASE_SIZE, then setting CPTRA_FUSE_WR_DONE to prevent further modifications. Additionally, OCP_LOCK_MEK_LENGTH must be set to 40h. See Section 4.7.1.2. | Yes |
| 14 | Upon KMB cold reset or firmware update reset, drive firmware shall enumerate HPKE keypairs supported by KMB and advertise them to the host, if drive firmware supports a host-facing API for doing so. See Section 4.6.2.1. | Yes |
| 15 | Integrations shall invoke Caliptra resets according to the directives given in Section 4.6.8. | Yes |

## 4.11 Repository location

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

# Appendix A: Preconditioned AES-Encrypt calculations

This appendix expands on Section 4.6.1.4 and provides additional details behind the claim that approximately $2^{80}$ preconditioned AES-Encrypt operations for a given encryption key 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. With a 96-bit salt, a given encryption key can be expanded to up to $2^{96}$ possible subkeys. This can be put in terms of the “Balls into bins” problem [25], where we are looking for the number of balls ($m$) required for the maximum load across $2^{96}$ 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 $m \approx 2^{128}$. Therefore a given encryption key may be used in at most $2^{128}$ preconditioned AES-Encrypt operations before any subkey derived from that key 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 [26] 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 subkey. If each of the $2^{96}$ derived subkeys might be used up to $2^{32}$ times, the expected number of derived subkeys that experience a collision is approximately $2^{96} \cdot 2^{-33} = 2^{63}$. Clearly a limit of $2^{128}$ encryption operations for a given encryption key, while meeting the letter of the FIPS requirements, is too large for safety.

We can calculate the safe margin by simply considering the 96-bit salt concatenated to the 96-bit IV. What we are really after is the maximum number of preconditioned AES-Encrypt invocations with a given encryption key such that the likelihood of experiencing a collision of the 192-bit (salt || IV) pair is at most $2^{-32}$. Leveraging the Birthday paradox equation where $p(n, d) \approx n^2/(2d)$, $n \approx \sqrt{2d \cdot p} = \sqrt{2^{192+1} \cdot 2^{-32}} = 2^{80.5}$.

Therefore a given encryption key may safely be used in at most $2^{80.5}$ preconditioned AES-Encrypt operations before an IV collision is expected to occur with probability greater than $2^{-32}$ across all of the AES-GCM subkeys derived from the encryption key.

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

This format is currently under development within TCG as part of the Hard Cryptographic Purge feature set[1].

[1] https://trustedcomputinggroup.org/trusted-computing-group-storage-work-group-call-for-participation/

## References


[1] "TCG Storage Architecture Core Specification." Trusted Computing Group. Available: https://trustedcomputinggroup.org/resource/tcg-storage-architecture-core-specification/

[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: Enterprise." Trusted Computing Group. Available: https://trustedcomputinggroup.org/resource/storage-work-group-storage-security-subsystem-class-enterprise-specification/

[4] "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/

[5] 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

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

[7] "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

[8] "Caliptra 2.0 Subsystem Specification." CHIPS Alliance. Available: https://github.com/chipsalliance/caliptra-ss/blob/9d10858/docs/CaliptraSSHardwareSpecification.md

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

[10] "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

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

[12] "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

[13] "The XAES-256-GCM extended-nonce AEAD." C2SP. Accessed: Jul. 2025. [Online]. Available: https://c2sp.org/XAES-256-GCM

[14] "TCG DICE." Trusted Computing Group. Available: https://trustedcomputinggroup.org/work-groups/dice-architectures/

[15] R. Barnes, "Post-Quantum and Post-Quantum/Traditional Hybrid Algorithms for HPKE." Jun. 2025. Available: https://www.ietf.org/archive/id/draft-ietf-hpke-pq-01.html

[16] D. Connolly, R. Barnes, and P. Grubbs, "Hybrid PQ/T Key Encapsulation Mechanisms." Jul. 2025. Available: https://www.ietf.org/archive/id/draft-irtf-cfrg-hybrid-kems-05.html

[17] D. Connolly and R. Barnes, "Concrete Hybrid PQ/T Key Encapsulation Mechanisms." Jul. 2025. Available: https://www.ietf.org/archive/id/draft-irtf-cfrg-concrete-hybrid-kems-00.html

[18] "TCG Storage Feature Set: Media Encryption Key Multiparty Authorization." Trusted Computing Group. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCG_Storage_MEK_Multiparty_Authorization_Feature_V1_00_RC1_10July2025.pdf

[19] “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

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

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

[22] “NVM Express Base Specification, Revision 2.2.” NVM Express, Mar. 2025. Available: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.2-2025.03.11-Ratified.pdf

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

[24] J. Massimo, P. Kampanakis, S. Turner, and B. E. Westerbaan, “Internet X.509 Public Key Infrastructure: Algorithm Identifiers for ML-DSA.” Jun. 2025. Available: https://www.ietf.org/archive/id/draft-ietf-lamps-dilithium-certificates-12.html

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

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