Qualcomm

Qualcomm Technologies, Inc.

Linux Android Userdata Encryption

80-PN330-9 Rev. A

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision History

Revision	Date	Description
A	March 2019	Initial release

Contents

Acronym

FDE	Full-Disk Encryption
FBE	File-Based Encryption
ME	Metadata Encryption
DE	Device Encrypted
CE	Credential Encrypted
PFK	Per File Key
PFE	Per File Encryption
ICE	Inline Crypto Engine
SHA	Secure Hash Algorithm
HMAC	Hashed Message Authentication Code
KDF	Key Derivation Function
WK	Wrapped Key
EK	Ephemeral Key
FNEK	File Name Encryption Key
FDWK	File Data Wrapped encryption Key

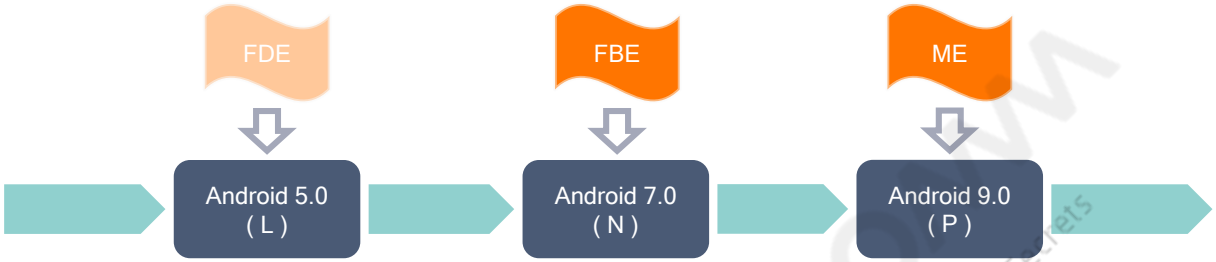
Section 1

General

Encryption

- Encryption is the process of encoding all user data on an Android device using symmetric encryption keys.
- Once a device is encrypted, all user-created data is automatically encrypted before being committed to the disk and all reads automatically decrypt data before returning it to the calling process.
- Encryption ensures that even if an unauthorized party tries to access the data, they won't be able to read it.

Encryption Methods



FDE – Full Disk Encryption

- A single key to protect the whole of a device's userdata partition.
- The user must provide their credentials before any part of the disk is accessible.

FBE – File Based Encryption

- Allow different files to be encrypted with different keys that can be unlocked independently.
- Support Direct Boot, thus enabling quick access to important device features like accessibility services and alarms.

ME – Metadata Encryption

- Based on FBE, a single key is used to encrypt whatever content is not encrypted by FBE.
- Not mandatory by Google.

Supporting Table

	EXT4 + FDE	EXT4 + FBE	EXT4 + ME	F2FS + FDE	F2FS + FBE	F2FS + ME
SM8150	Yes	Yes	Yes (default)	NA	NA	NA
SM6150/7150	Yes (UFS only)	NA	NA	NA	Yes (default)	Yes
SM6125	NA	NA	NA	NA	Yes	Yes (default)
SDM845	Yes	Yes	NA	NA	Yes	NA
SDM670/710	Yes	Yes	NA	NA	Yes	Yes
SDM660/630	Yes	Yes	NA	NA	NA	NA

Section 2

Full-Disk Encryption (FDE)

Section 2

Full-Disk Encryption (FDE)

Full Disk Encryption

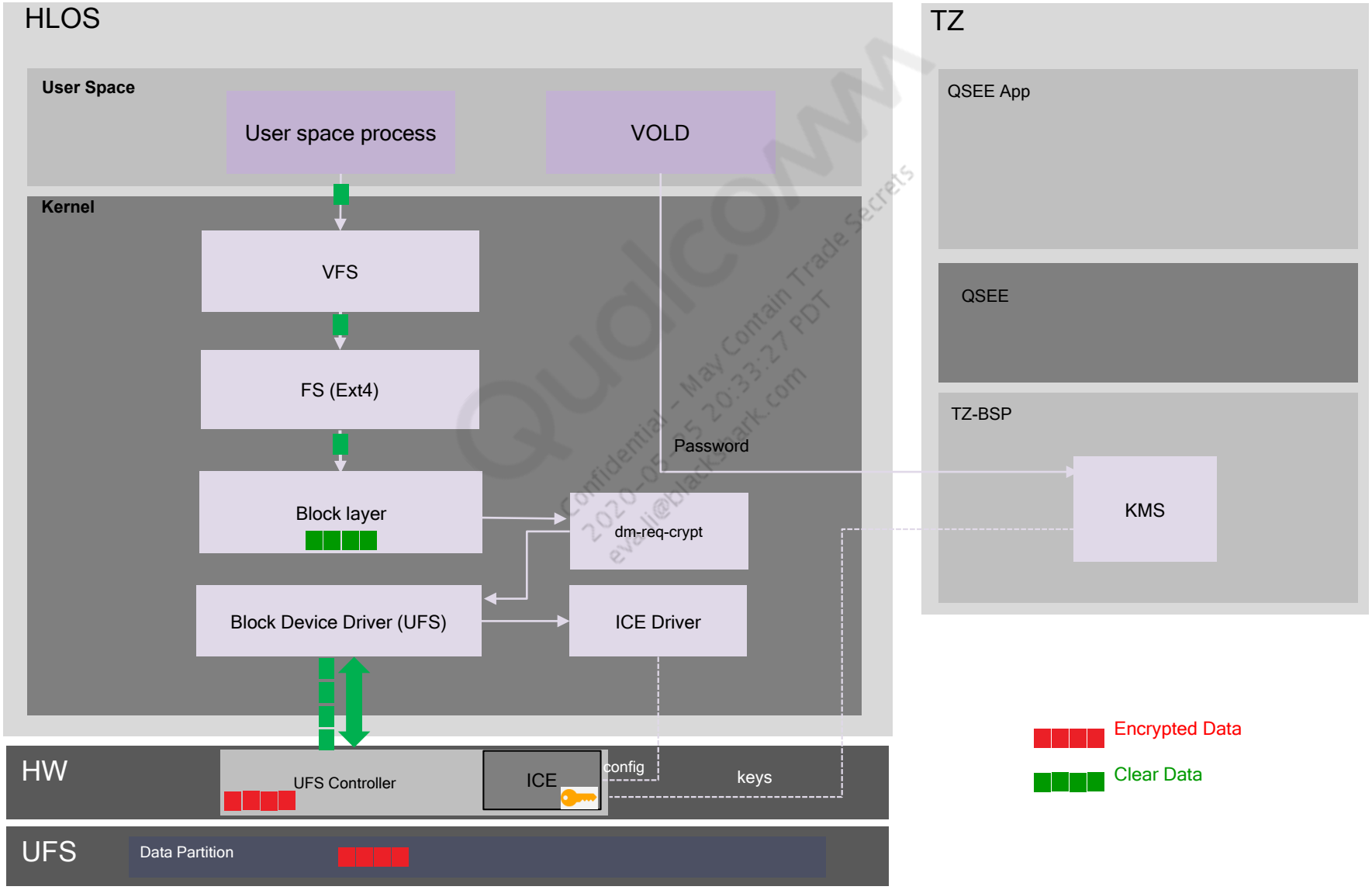
- Data partition encryption with single encryption key.

Basic Flows

- Encrypt new device (default password)
- Inplace encryption
- Decrypt device with default password
- Change password
- Decrypt device with non-default password

FDE is deprecated

Architecture – FDE



Encrypt New Device (1/2)

- Commercial devices get encrypted on first usage with default password
  - “default_password”

- Controlled by `fstab.qcom` file
  - init.target.rc : `mount_all /vendor/etc/fstab.qcom`
  - Partitions to be encrypted marked with `forceencrypt` or `encryptable` mgr flags
  - `forceencrypt` means that encryption is forced on first usage
  - `encryptable` means manual encryption

- If hardware crypto supported, cipher is AES-XTS.

- The last 16 KB of the userdata partition is reserved to store crypto metadata.

Encrypt New Device (2/2)

1. **Detect unencrypted filesystem with `/forceencrypt` flag**

   `/data` is not encrypted but needs to be because `/forceencrypt` mandates it. Unmount `/data`.

2. **Start encrypting `/data`**

   `vold.decrypt = "trigger_encryption"` triggers `init.rc`, which will cause `vold` to encrypt `/data` with no password. (None is set because this should be a new device.)

3. **Mount tmpfs**

   `vold` mounts a tmpfs `/data` (using the tmpfs options from `ro.crypto.tmpfs_options`) and sets the property `vold.encrypt_progress` to 0. `vold` prepepares the tmpfs `/data` for booting an encrypted system and sets the property `vold.decrypt` to: `trigger_restart_min_framework`

4. **Bring up framework to show progress**

   Because the device has virtually no data to encrypt, the progress bar will often not actually appear because encryption happens so quickly. See Encrypt an existing device for more details about the progress UI.

5. **When `/data` is encrypted, take down the framework**

   `vold` sets `vold.decrypt` to `trigger_default_encryption` which starts the `defaultcrypto` service. (This starts the flow below for mounting a default encrypted userdata.) `trigger_default_encryption` checks the encryption type to see if `/data` is encrypted with or without a password. Because Android 5.0 devices are encrypted on first boot, there should be no password set; therefore we decrypt and mount `/data`.

6. **Mount `/data`**

   `init` then mounts `/data` on a tmpfs RAMDisk using parameters it picks up from `ro.crypto.tmpfs_options`, which is set in `init.rc`.

7. **Start framework**

   Set `vold` to `trigger_restart_framework`, which continues the usual boot process.

Decrypt – default password (1/2)

When device boots up, encrypted partitions need to be decrypted before the usual flow proceeds.

- If partition is marked with either `forceencrypt` or `encryptable` flags and can't be mounted, it means that it is already encrypted
- We setup decryption with "default_password" and try to mount data temporarily, if we succeed, then we proceed with the usual flow

Decrypt – default password (2/2)

Starting an encrypted device with default encryption

This is what happens when you boot up an encrypted device with no password. Because Android 5.0 devices are encrypted on first boot, there should be no set password and therefore this is the *default encryption* state.

1. **Detect encrypted `/data` with no password**

   Detect that the Android device is encrypted because `/data` cannot be mounted and one of the flags `encryptable` or `forceencrypt` is set.

   `vold` sets `vold.decrypt` to `trigger_default_encryption`, which starts the `defaultcrypto` service. `trigger_default_encryption` checks the encryption type to see if `/data` is encrypted with or without a password.

2. **Decrypt /data**

   Creates the `dm-crypt` device over the block device so the device is ready for use.

3. **Mount /data**

   `vold` then mounts the decrypted real `/data` partition and then prepares the new partition. It sets the property `vold.post_fs_data_done` to 0 and then sets `vold.decrypt` to `trigger_post_fs_data`. This causes `init.rc` to run its `post-fs-data` commands. They will create any necessary directories or links and then set `vold.post_fs_data_done` to 1.

   Once `vold` sees the 1 in that property, it sets the property `vold.decrypt` to: `trigger_restart_framework`. This causes `init.rc` to start services in class `main` again and also start services in class `late_start` for the first time since boot.

4. **Start framework**

   Now the framework boots all its services using the decrypted `/data`, and the system is ready for use.

Additional Operations

- Inplace encryption
  - In “theory” starting Android 5, all devices should have partitions force encrypted on the first boot, however each OEM may choose not to mark partitions as `forceencrypt`.
  - Also, if a device is upgraded from previous Android versions, partitions are not encrypted automatically.
  - In place encryption occurs when user explicitly chooses to encrypt device via the **Settings** screen.
  - The flow is very similar to the default encryption, except that the password is not “default_password” and the encryption is started after the boot up is completed.
- Change password
  - Happens when the user sets ups the FDE password for the first time (it is changed from “default_password” or when he decides to change it.
  - Master key is decrypted with the previous password and then re-encrypted again with new password.
- If Qcom hardware keymaster is applied, the master key is managed by TZ and stored in SSD partition. It never leaves TZ, and is loaded to ICE directly.

Section 3

File-Based Encryption (FBE)

Section 3

File-Based Encryption (FBE)

File-Based Encryption

- Supported by Android 7.0 and later.
- Helps the new features.
  - Making applications Direct Boot aware
    - Direct Boot allows encrypted devices to boot straight to the lock screen. Basic operations, like alarm/accessibility/receive calls, are available even before user inputs credentials.
  - Supporting multiple users
    - Each user in a multi-user environment gets a separate encryption key.
    - Every user gets two keys: a DE and a CE key. User 0 must log into the device first as that is a special user.

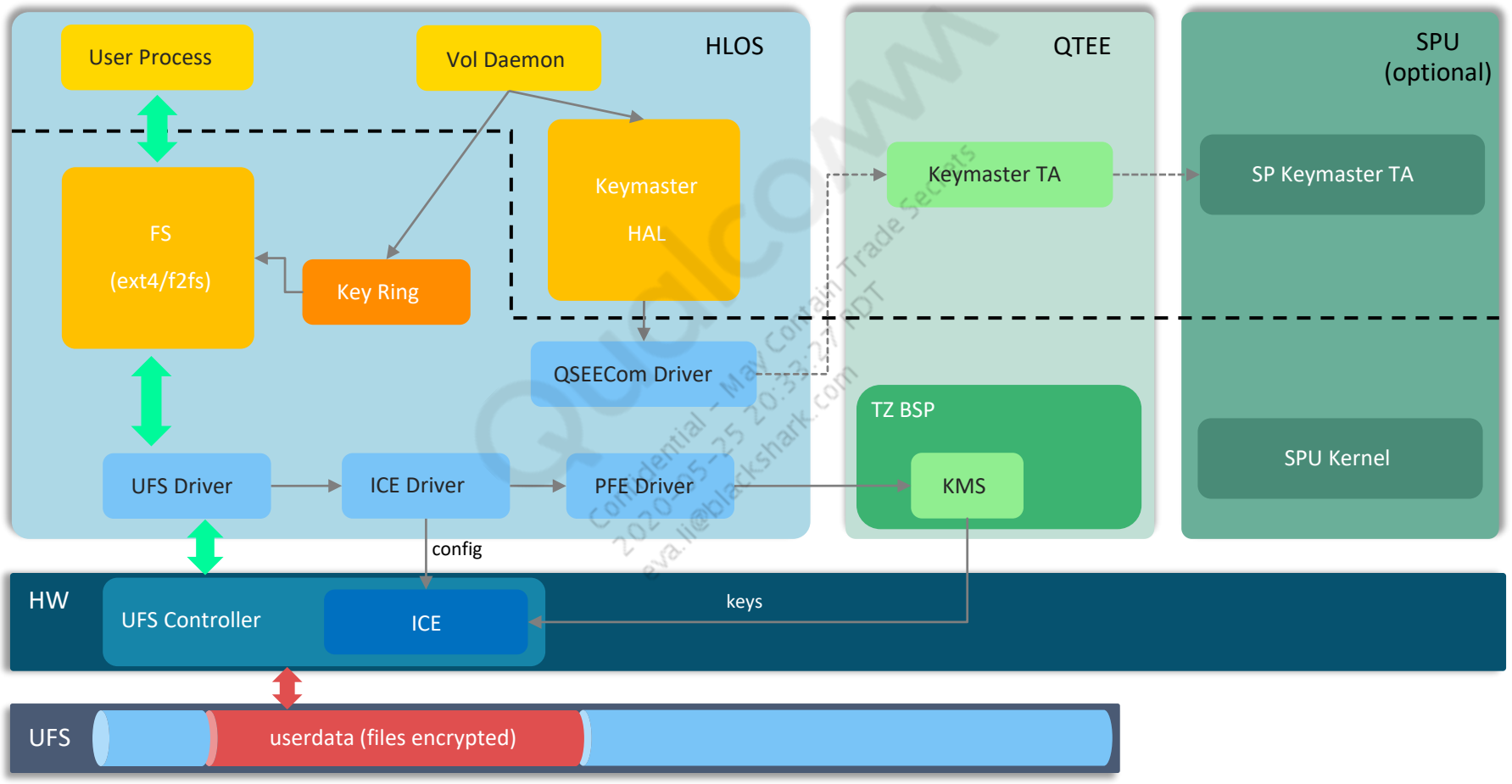
Different PFK in QC releases

- msm-4.9 : Allows different files to be encrypted with different keys.
- msm-4.14 : A file is encrypted by the class key of its encryption policy.

Requirements

- The recommended solution is to use a kernel based on 4.4 or later.
- Ensure Keymaster is started before `/data` is mounted
- userdata partition should be formatted as ext4 or f2fs filesystem.
- FBE is enabled by adding below flag to the fstab line in the final column for the userdata partition.
  - "`fileencryption=contents_encryption_mode[:filenames_encryption_mode]`"
  - *`contents_encryption_mode`* can be only *aes-256-xts*
  - *`filenames_encryption_mode`* has two possible values: *aes-256-cts* and *aes-256-heh*
  - For Hardware FBE, it should be "`fileencryption=ice`"

Architecture – FBE



Encryption Policy

```
#define FS_KEY_DESCRIPTOR_SIZE      8

/* Encryption algorithms */
#define FS_ENCRYPTION_MODE_INVALID              0
#define FS_ENCRYPTION_MODE_AES_256_XTS                      1
#define FS_ENCRYPTION_MODE_AES_256_GCM                      2
#define FS_ENCRYPTION_MODE_AES_256_CBC                      3
#define FS_ENCRYPTION_MODE_AES_256_CTS                      4
#define FS_ENCRYPTION_MODE_AES_128_CBC                      5
#define FS_ENCRYPTION_MODE_AES_128_CTS                      6
#define FS_ENCRYPTION_MODE_SPECK128_256_XTS     7
#define FS_ENCRYPTION_MODE_SPECK128_256_CTS     8
#define FS_ENCRYPTION_MODE_PRIVATE              127

struct fscrypt_policy {
    __u8 version;
    __u8 contents_encryption_mode;
    __u8 filenames_encryption_mode;
    __u8 flags;
    __u8 master_key_descriptor[FS_KEY_DESCRIPTOR_SIZE];
};
```


- Encryption policies are applied at the directory level.
- All subfolders inherit the encryption policy from the top encrypted directory.
- The root directory "`/data`" doesn't include any encryption policy.

DE & CE

For a FBE-enabled device, each user of the device has two storage locations available to applications:

- Device Encrypted (DE) storage, which is a storage location available both during Direct Boot mode and after the user has unlocked the device.
- Credential Encrypted (CE) storage, which is the default storage location and only available after the user has unlocked the device.

Upon the 1st boot after FBE is enabled, 3 types of key will be created and applied on different directories.

- System DE key
- User DE key for user 0
- User CE key for user 0

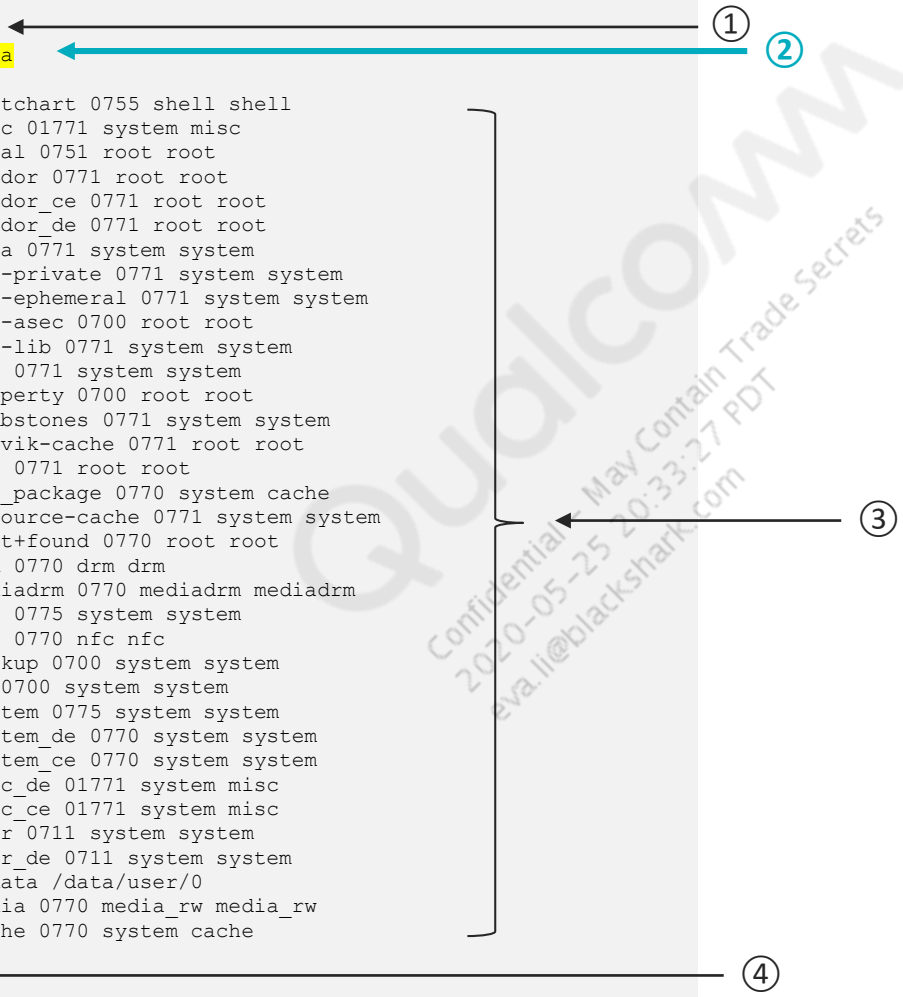
If an additional user X is created, new User DE&CE keys for user X will be generated, and applied to those directories only for user X.

Example – Top Level Directory

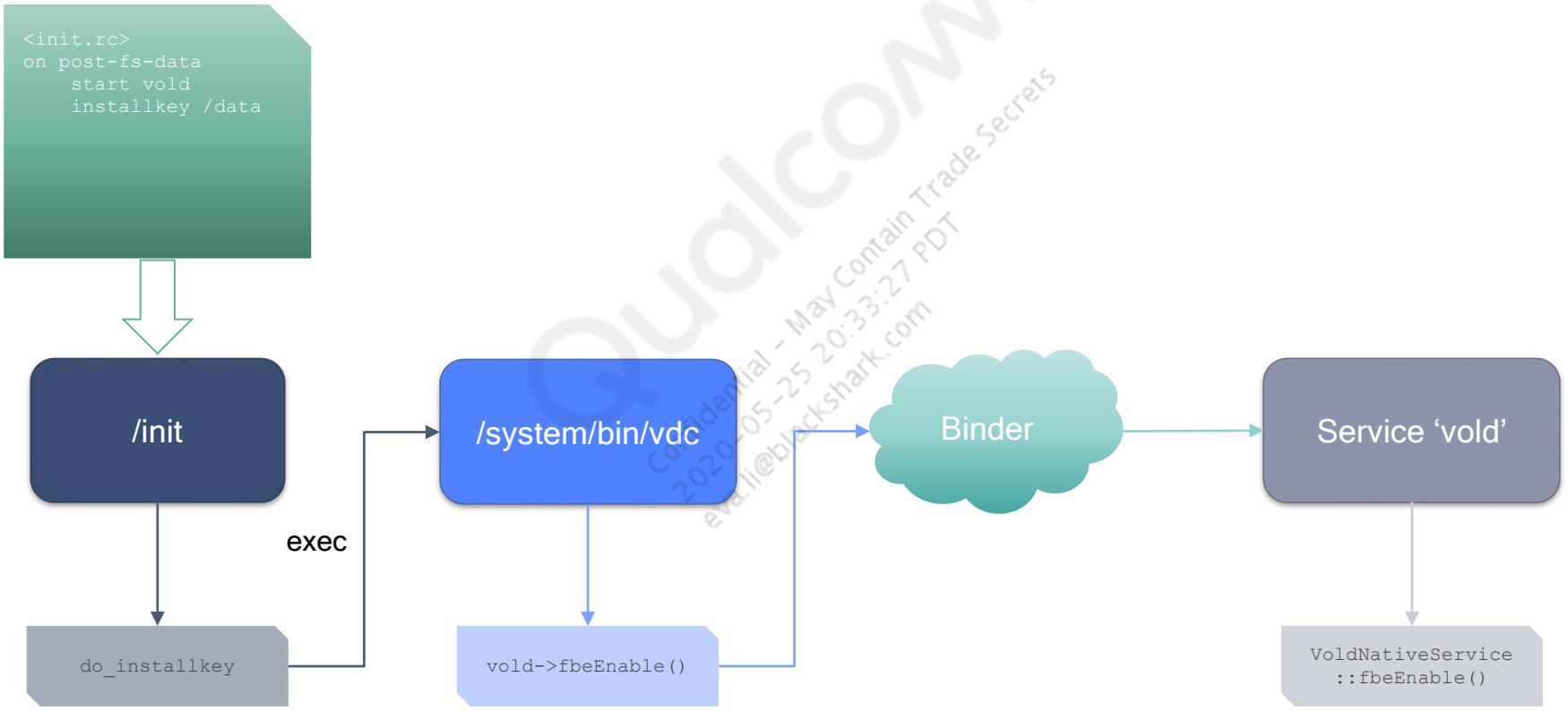
Unencrypted	System DE Policy	User.0 DE Policy	User.0 CE Policy
/data/lost+found /data/unencrypted /data/media /data/system_ce /data/system_de /data/misc_ce /data/misc_de /data/user /data/user_de /data/vendor_ce /data/vendor_de	/data/misc /data/local /data/vendor /data/property /data/tombstones /data/dalvik-cache /data/resource-cache /data/backup /data/system /data/cache /data/adb /data/anr /data/app /data/app-asec /data/app-ephemeral /data/app-lib /data/app-private /data/bootchart /data/dpm /data/drm /data/mediadrm /data/nfc /data/ota /data/ota_package /data/ss	/data/system_de/0 /data/misc_de/0 /data/vendor_de/0 /data/user_de/0	/data/system_ce/0 /data/misc_ce/0 /data/vendor_ce/0 /data/media/0 /data/data

Boot Flow 1 – init.rc

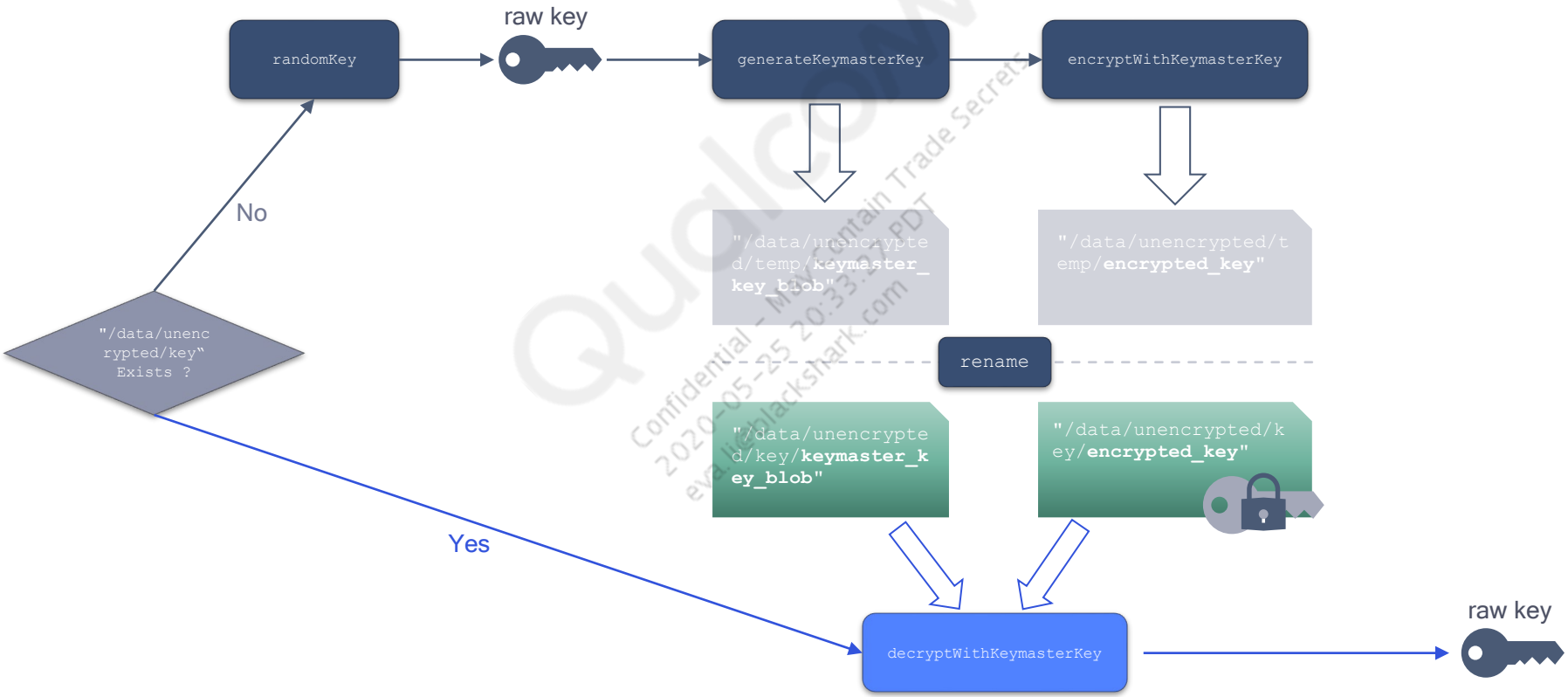
```
on post-fs-data
    start vold
    installkey /data

    mkdir /data/bootchart 0755 shell shell
    mkdir /data/misc 01771 system misc
    mkdir /data/local 0751 root root
    mkdir /data/vendor 0771 root root
    mkdir /data/vendor_ce 0771 root root
    mkdir /data/vendor_de 0771 root root
    mkdir /data/data 0771 system system
    mkdir /data/app-private 0771 system system
    mkdir /data/app-ephemeral 0771 system system
    mkdir /data/app-asec 0700 root root
    mkdir /data/app-lib 0771 system system
    mkdir /data/app 0771 system system
    mkdir /data/property 0700 root root
    mkdir /data/tombstones 0771 system system
    mkdir /data/dalvik-cache 0771 root root
    mkdir /data/ota 0771 root root
    mkdir /data/ota_package 0770 system cache
    mkdir /data/resource-cache 0771 system system
    mkdir /data/lost+found 0770 root root
    mkdir /data/drm 0770 drm drm
    mkdir /data/mediadrm 0770 mediadrm mediadrm
    mkdir /data/anr 0775 system system
    mkdir /data/nfc 0770 nfc nfc
    mkdir /data/backup 0700 system system
    mkdir /data/ss 0700 system system
    mkdir /data/system 0775 system system
    mkdir /data/system_de 0770 system system
    mkdir /data/system_ce 0770 system system
    mkdir /data/misc_de 01771 system misc
    mkdir /data/misc_ce 01771 system misc
    mkdir /data/user 0711 system system
    mkdir /data/user_de 0711 system system
    symlink /data/data /data/user/0
    mkdir /data/media 0770 media_rw media_rw
    mkdir /data/cache 0770 system cache

    init_user0

    # Set SELinux security contexts on upgrade or policy update.
    restorecon --recursive --skip-ce /data
```



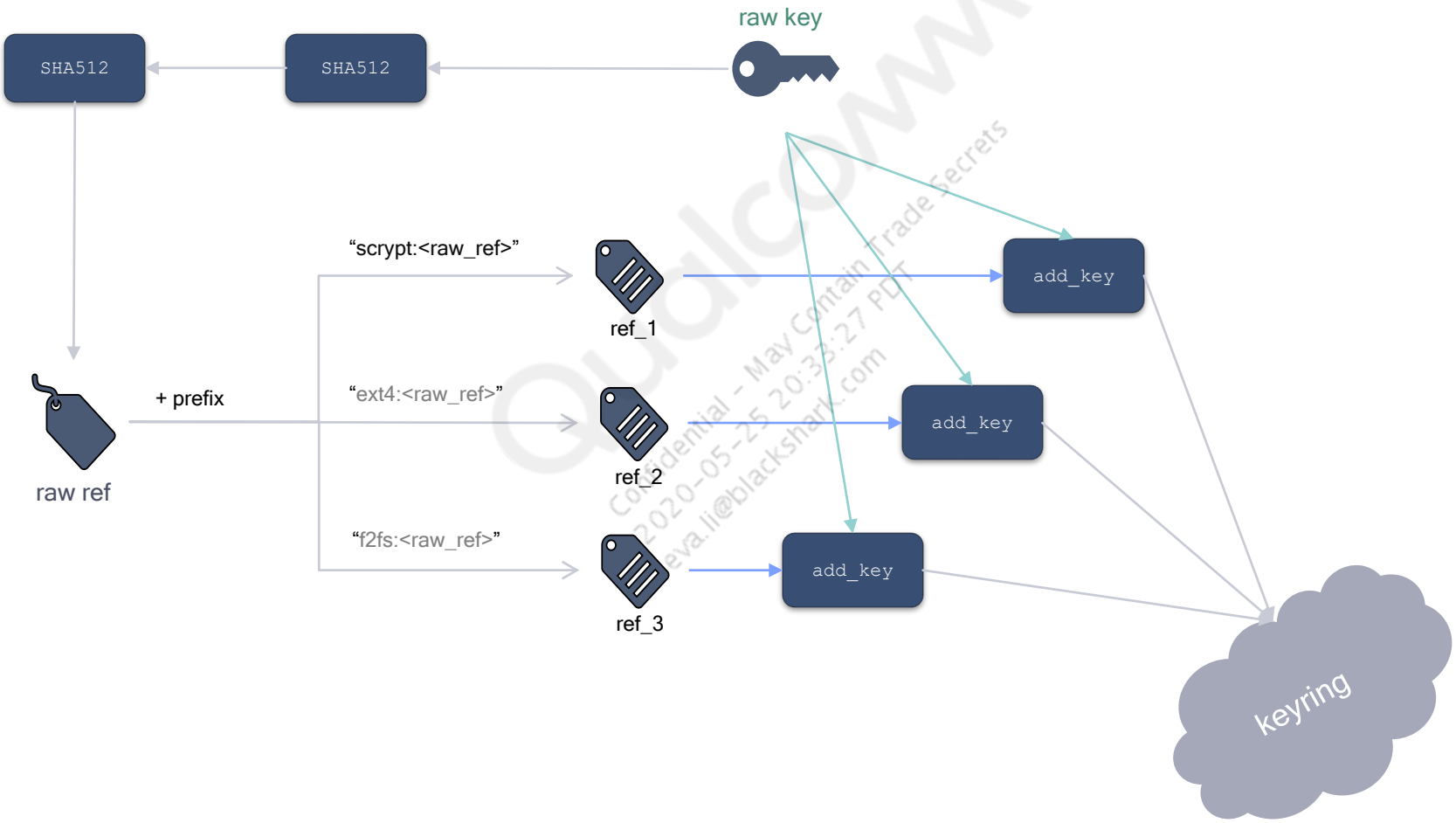
Boot Flow 2 – installkey



Generate/Retrieve System DE key

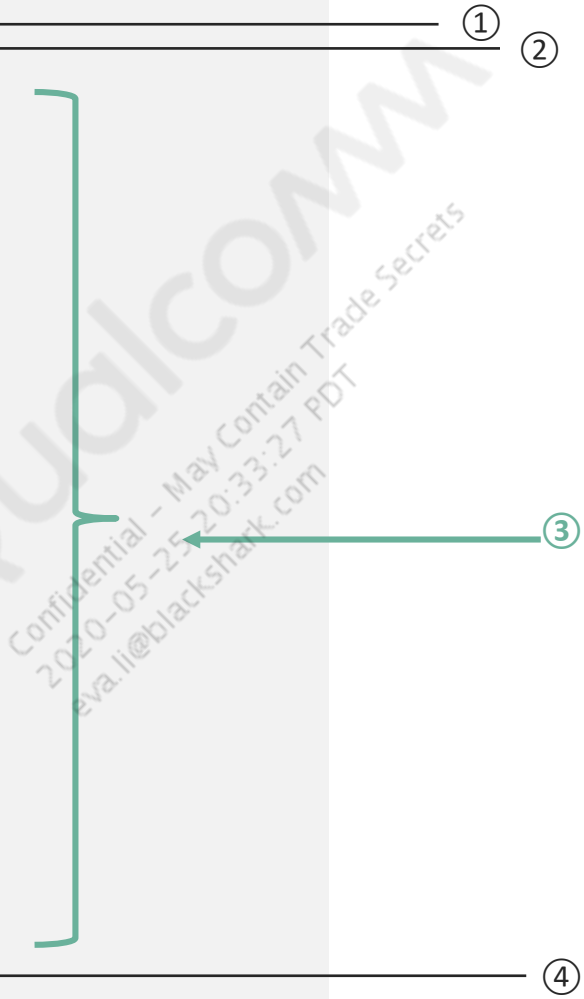


Add System DE key



Boot Flow 3 – mkdir

```
on post-fs-data
    start vold
    installkey /data

    mkdir /data/bootchart 0755 shell shell
    mkdir /data/misc 01771 system misc
    mkdir /data/local 0751 root root
    mkdir /data/vendor 0771 root root
    mkdir /data/vendor_ce 0771 root root
    mkdir /data/vendor_de 0771 root root
    mkdir /data/data 0771 system system
    mkdir /data/app-private 0771 system system
    mkdir /data/app-ephemeral 0771 system system
    mkdir /data/app-asec 0700 root root
    mkdir /data/app-lib 0771 system system
    mkdir /data/app 0771 system system
    mkdir /data/property 0700 root root
    mkdir /data/tombstones 0771 system system
    mkdir /data/dalvik-cache 0771 root root
    mkdir /data/ota 0771 root root
    mkdir /data/ota_package 0770 system cache
    mkdir /data/resource-cache 0771 system system
    mkdir /data/lost+found 0770 root root
    mkdir /data/drm 0770 drm drm
    mkdir /data/mediadrm 0770 mediadrm mediadrm
    mkdir /data/anr 0775 system system
    mkdir /data/nfc 0770 nfc nfc
    mkdir /data/backup 0700 system system
    mkdir /data/ss 0700 system system
    mkdir /data/system 0775 system system
    mkdir /data/system_de 0770 system system
    mkdir /data/system_ce 0770 system system
    mkdir /data/misc_de 01771 system misc
    mkdir /data/misc_ce 01771 system misc
    mkdir /data/user 0711 system system
    mkdir /data/user_de 0711 system system
    symlink /data/data /data/user/0
    mkdir /data/media 0770 media_rw media_rw
    mkdir /data/cache 0770 system cache

    init_user0

    # Set SELinux security contexts on upgrade or policy update.
    restorecon --recursive --skip-ce /data
```

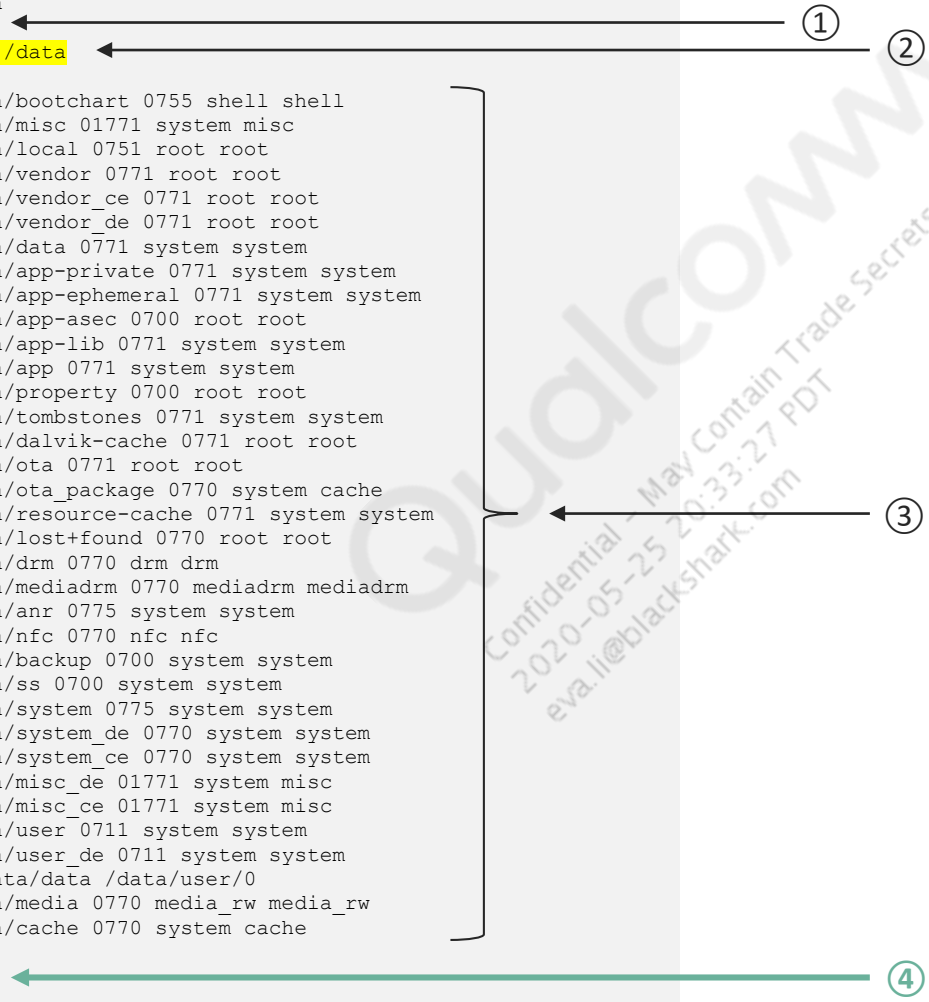


Different “mkdir” actions

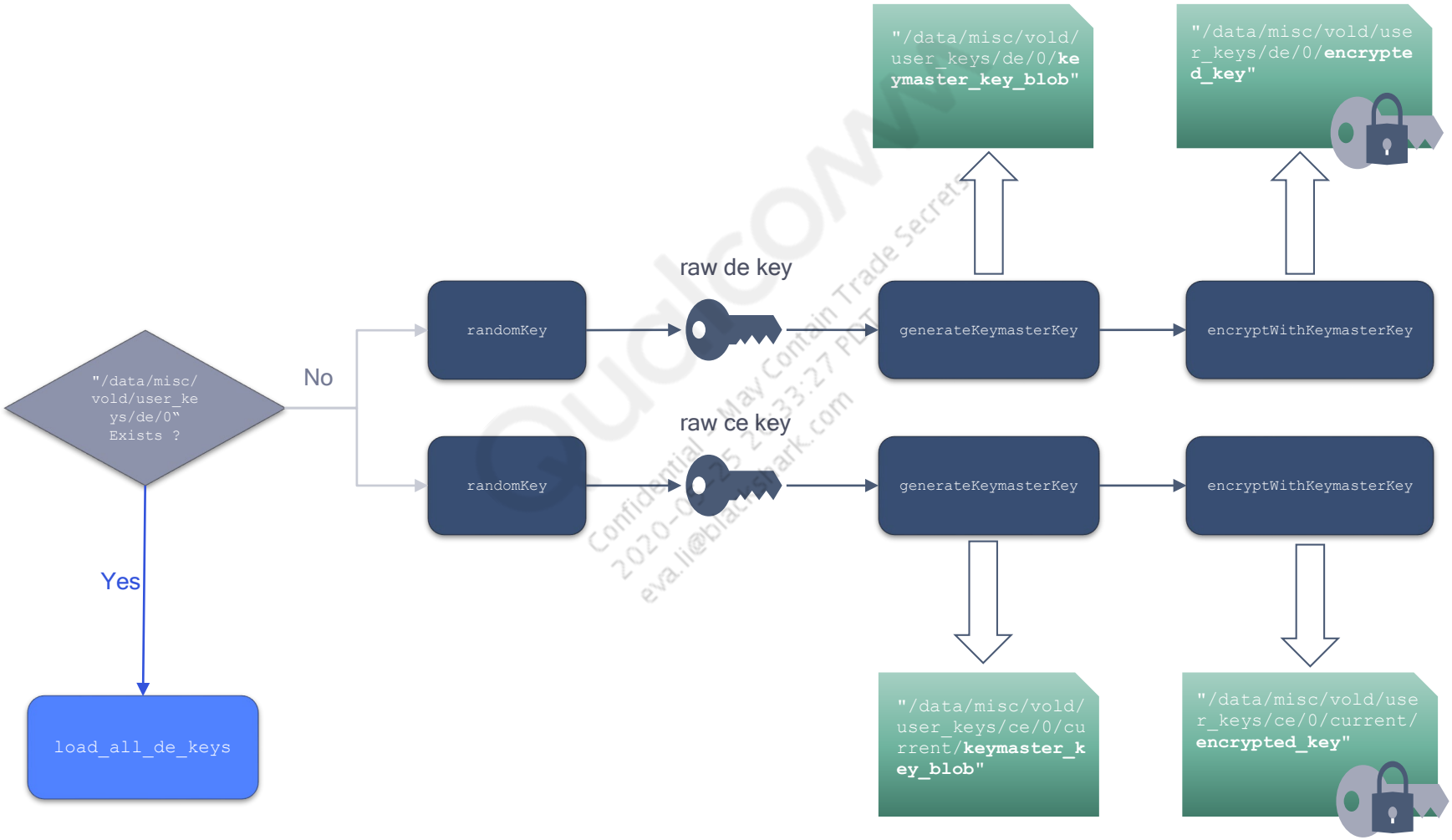
- mkdir – toybox
  - Inherit the same policy from its parent directory.
  - If the parent dir is unencrypted, the new dir is also not encrypted.
- mkdir – init builtin command
  - Set System DE policy to the new dir.

Boot Flow 4 – init_user0

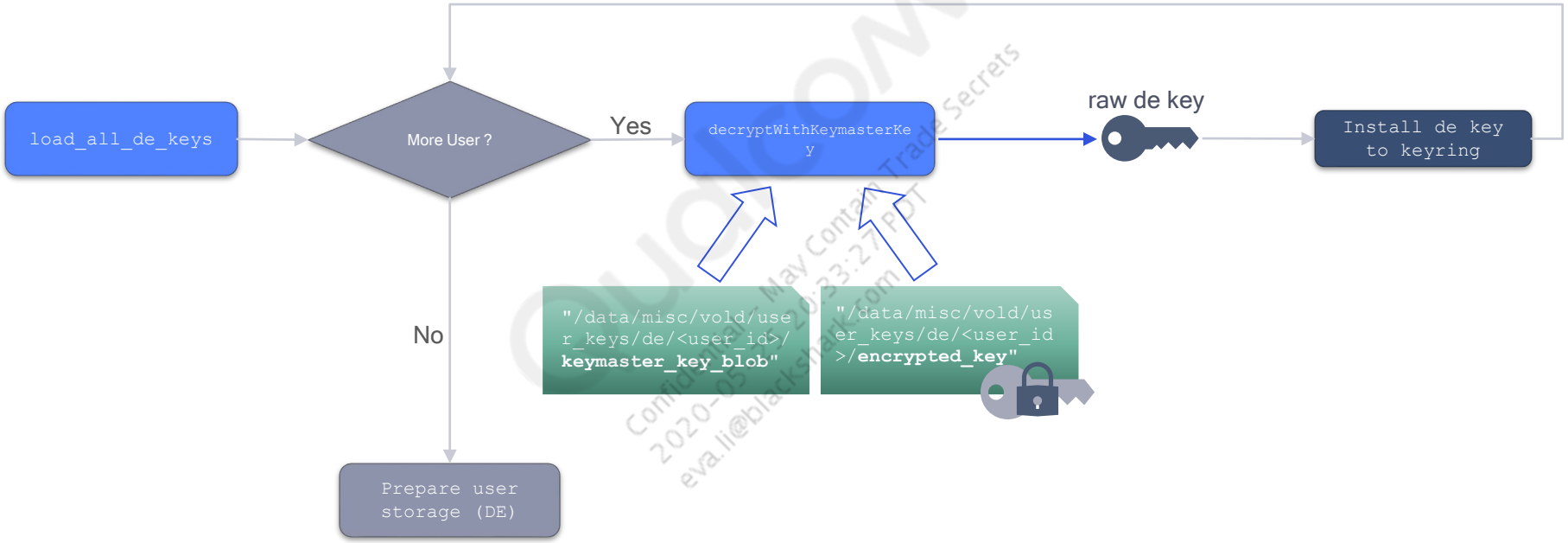
```
on post-fs-data
    start vold
    installkey /data

    mkdir /data/bootchart 0755 shell shell
    mkdir /data/misc 01771 system misc
    mkdir /data/local 0751 root root
    mkdir /data/vendor 0771 root root
    mkdir /data/vendor_ce 0771 root root
    mkdir /data/vendor_de 0771 root root
    mkdir /data/data 0771 system system
    mkdir /data/app-private 0771 system system
    mkdir /data/app-ephemeral 0771 system system
    mkdir /data/app-asec 0700 root root
    mkdir /data/app-lib 0771 system system
    mkdir /data/app 0771 system system
    mkdir /data/property 0700 root root
    mkdir /data/tombstones 0771 system system
    mkdir /data/dalvik-cache 0771 root root
    mkdir /data/ota 0771 root root
    mkdir /data/ota_package 0770 system cache
    mkdir /data/resource-cache 0771 system system
    mkdir /data/lost+found 0770 root root
    mkdir /data/drm 0770 drm drm
    mkdir /data/mediadrm 0770 mediadrm mediadrm
    mkdir /data/anr 0775 system system
    mkdir /data/nfc 0770 nfc nfc
    mkdir /data/backup 0700 system system
    mkdir /data/ss 0700 system system
    mkdir /data/system 0775 system system
    mkdir /data/system_de 0770 system system
    mkdir /data/system_ce 0770 system system
    mkdir /data/misc_de 01771 system misc
    mkdir /data/misc_ce 01771 system misc
    mkdir /data/user 0711 system system
    mkdir /data/user_de 0711 system system
    symlink /data/data /data/user/0
    mkdir /data/media 0770 media_rw media_rw
    mkdir /data/cache 0770 system cache

    init_user0

    # Set SELinux security contexts on upgrade or policy update.
    restorecon --recursive --skip-ce /data
```



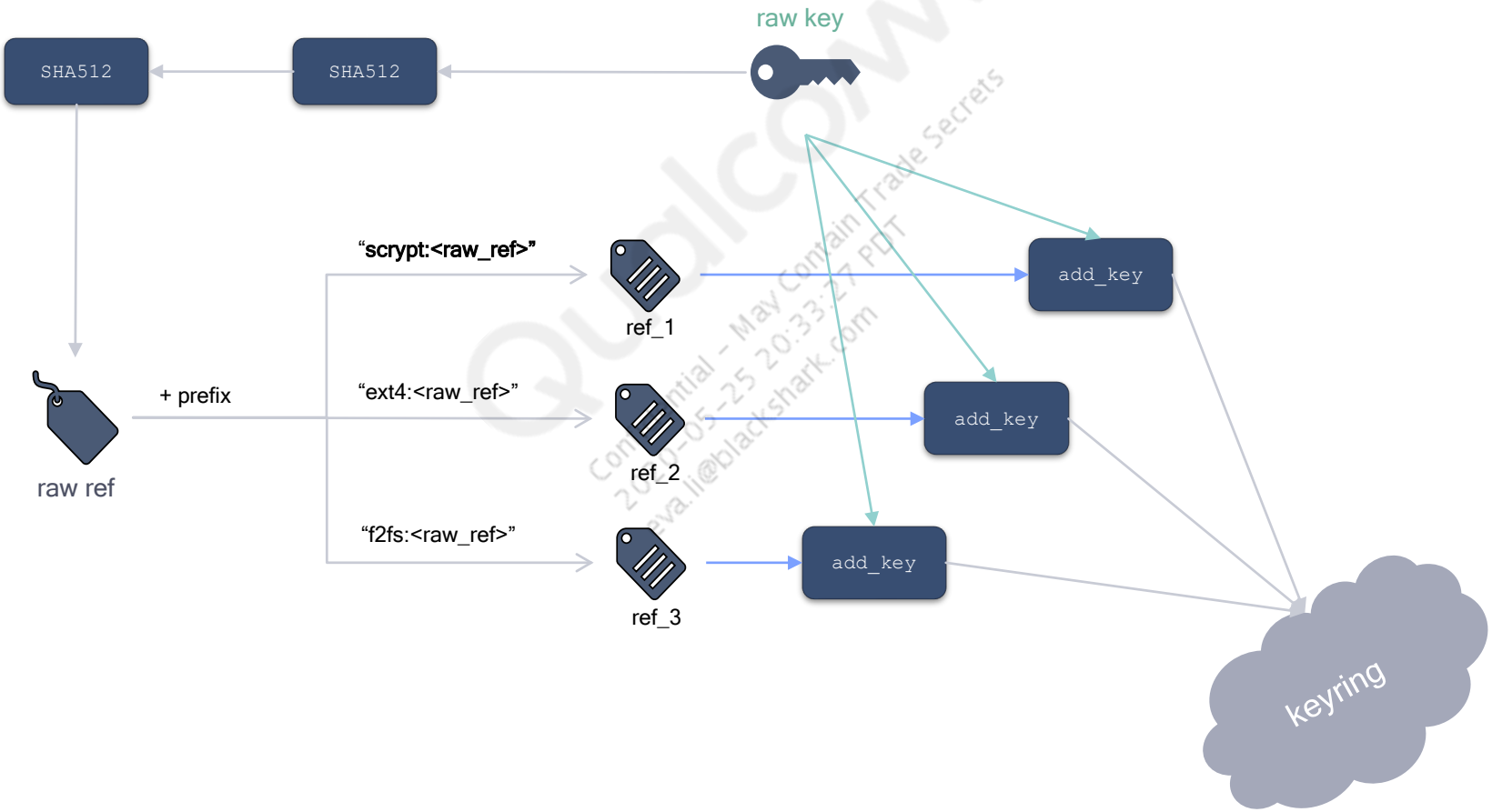
Create/Retrieve User DE/CE keys



Load All Users' DE keys



Add User DE/CE key



Boot log (logcat - system)

```
I vold    : Vold 3.0 (the awakening) firing up
...
D vold    : VoldNativeService::start() completed OK
I vold    : e4crypt_initialize_global_de
D vold    : Key exists, using: /data/unencrypted/key
...
D vold    : Added key 47947467 (ext4:fe8b7d2aa943ee67) to keyring 240296977 in process 646
D vold    : Added key 100265740 (f2fs:fe8b7d2aa943ee67) to keyring 240296977 in process 646
D vold    : Added key 799427332 (fscrypt:fe8b7d2aa943ee67) to keyring 240296977 in process 646
I vold    : Wrote system DE key reference to:/data/unencrypted/ref

D vold    : e4crypt_init_user0
...
D vold    : Added key 929309069 (ext4:186feb90db0f3628) to keyring 240296977 in process 646
D vold    : Added key 116913162 (f2fs:186feb90db0f3628) to keyring 240296977 in process 646
D vold    : Added key 279150948 (fscrypt:186feb90db0f3628) to keyring 240296977 in process 646
D vold    : Installed de key for user 0
...

D ActivityManager: Finishing user boot 0
I ActivityManager: User 0 state changed from BOOTING to RUNNING_LOCKED
D vold    : e4crypt_unlock_user_key 0 serial=0 token_present=0
D vold    : Trying user CE key /data/misc/vold/user_keys/ce/0/current
...
D vold    : Successfully retrieved key
D vold    : Added key 237097667 (ext4:f57d9a70b7f3d022) to keyring 240296977 in process 646
D vold    : Added key 166059276 (f2fs:f57d9a70b7f3d022) to keyring 240296977 in process 646
D vold    : Added key 985551159 (fscrypt:f57d9a70b7f3d022) to keyring 240296977 in process 646
D vold    : Installed ce key for user 0
```

Keys in keyring

```
$ cat /proc/keys

007ba04c I--Q---  1177 perm 3d010000     0  1065 logon     ext4:f57d9a70b7f3d022: 72
03f48e19 I--Q---     2 perm 1f3f0000     0 65534 keyring   _uid.0: empty
05257956 I--Q---     1 perm 3d010000     0  1065 logon     fscrypt:fe8b7d2aa943ee67: 72
08cd7422 I------     1 perm 1f030000     0     0 keyring   .dns_resolver: empty
0d2defce I--Q---     1 perm 1f3f0000     0 65534 keyring   _uid_ses.0: 1
16903dbf I--Q---     1 perm 3d010000     0  1065 logon     fscrypt:f57d9a70b7f3d022: 72
1bc632ea I------     1 perm 1f030000     0     0 asymmetri Build time autogenerated kernel key:
191b967ac6828276dd5f7dc7398830025d286d98: X509.RSA 5d286d98 []
1fc67cf0 I--Q---     1 perm 3d010000     0  1065 logon     fscrypt:186feb90db0f3628: 72
20335aa5 I------     1 perm 1f0b0000     0     0 keyring   .system_keyring: 2
2176bddf I--Q---     1 perm 3d010000     0  1065 logon     f2fs:fe8b7d2aa943ee67: 72
286d002f I--Q---     1 perm 3f010000     0     0 keyring   e4crypt: 9
2b2ed647 I--Q---     1 perm 3d010000     0  1065 logon     f2fs:f57d9a70b7f3d022: 72
2e861747 I--Q---  2140 perm 3d010000     0  1065 logon     ext4:fe8b7d2aa943ee67: 72
3269a51b I--Q---   545 perm 3f030000     0     0 keyring   _ses: 1
37577762 I------     1 perm 1f030000     0     0 asymmetri Android: 7e4333f9bba00adfe0ede979e28ed1920492b40f: X509.RSA 0492b40f []
3ae8aee7 I--Q---     1 perm 3d010000     0  1065 logon     f2fs:186feb90db0f3628: 72
3f079209 I--Q---   659 perm 3d010000     0  1065 logon     ext4:186feb90db0f3628: 72
```

ref to User.0 CE key

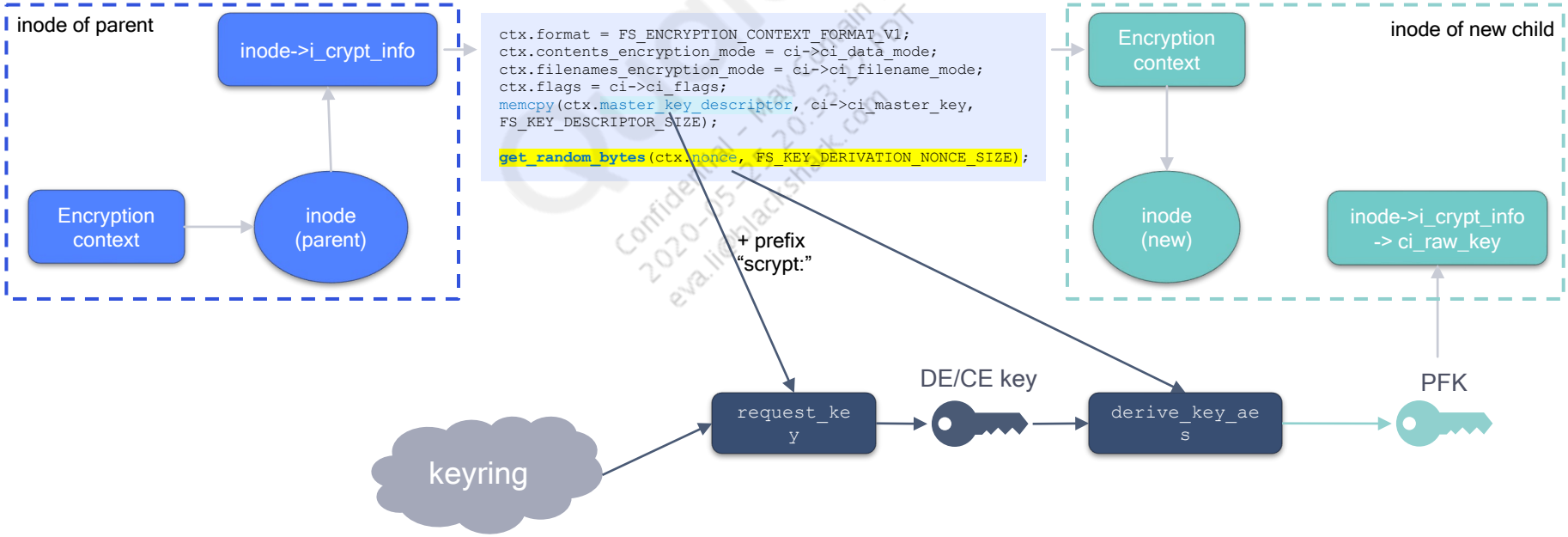
ref to System DE key

ref to User.0 DE key

Key Derivation - PFK

```
struct fscrypt_context {
    u8 format;
    u8 contents_encryption_mode;
    u8 filenames_encryption_mode;
    u8 flags;
    u8 master_key_descriptor[FS_KEY_DESCRIPTOR_SIZE];
    u8 nonce[FS_KEY_DERIVATION_NONCE_SIZE];
} __packed;
```

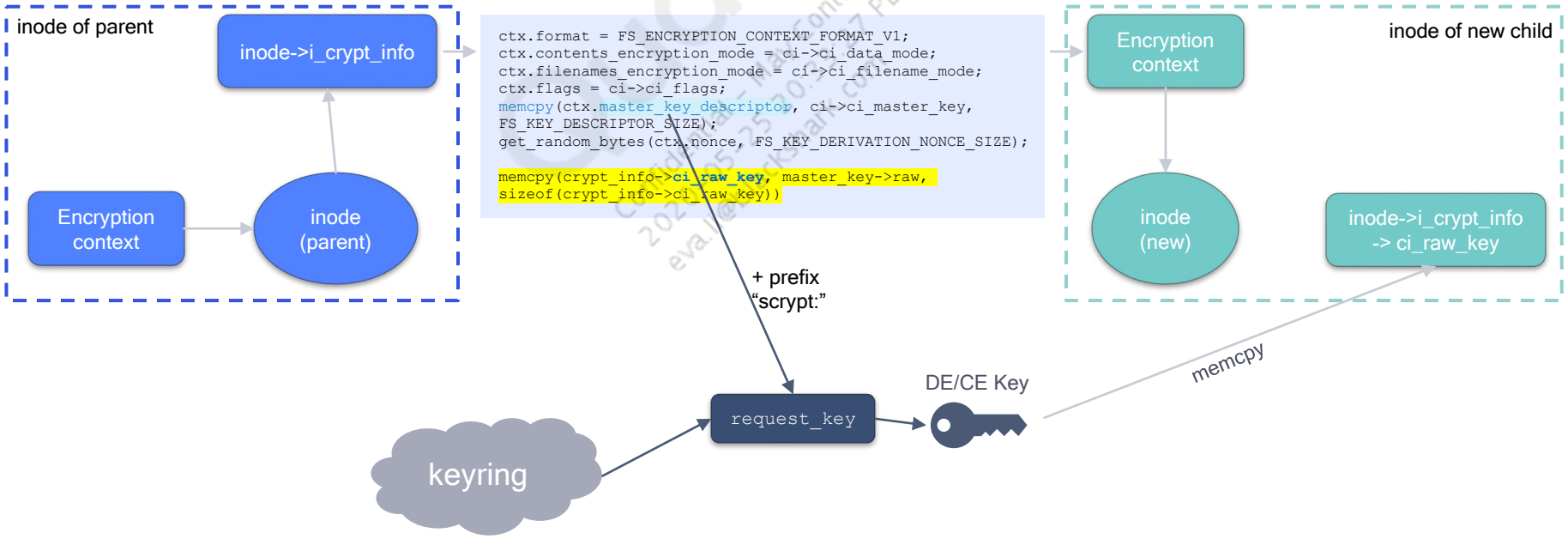
```
struct fscrypt_info {
    u8 ci_mode;
    u8 ci_data_mode;
    u8 ci_filename_mode;
    u8 ci_flags;
    struct crypto_skcipher *ci_ctfm;
    struct crypto_cipher *ci_essiv_tfm;
    u8 ci_master_key[FS_KEY_DESCRIPTOR_SIZE];
    u8 ci_raw_key[FS_MAX_KEY_SIZE];
};
```



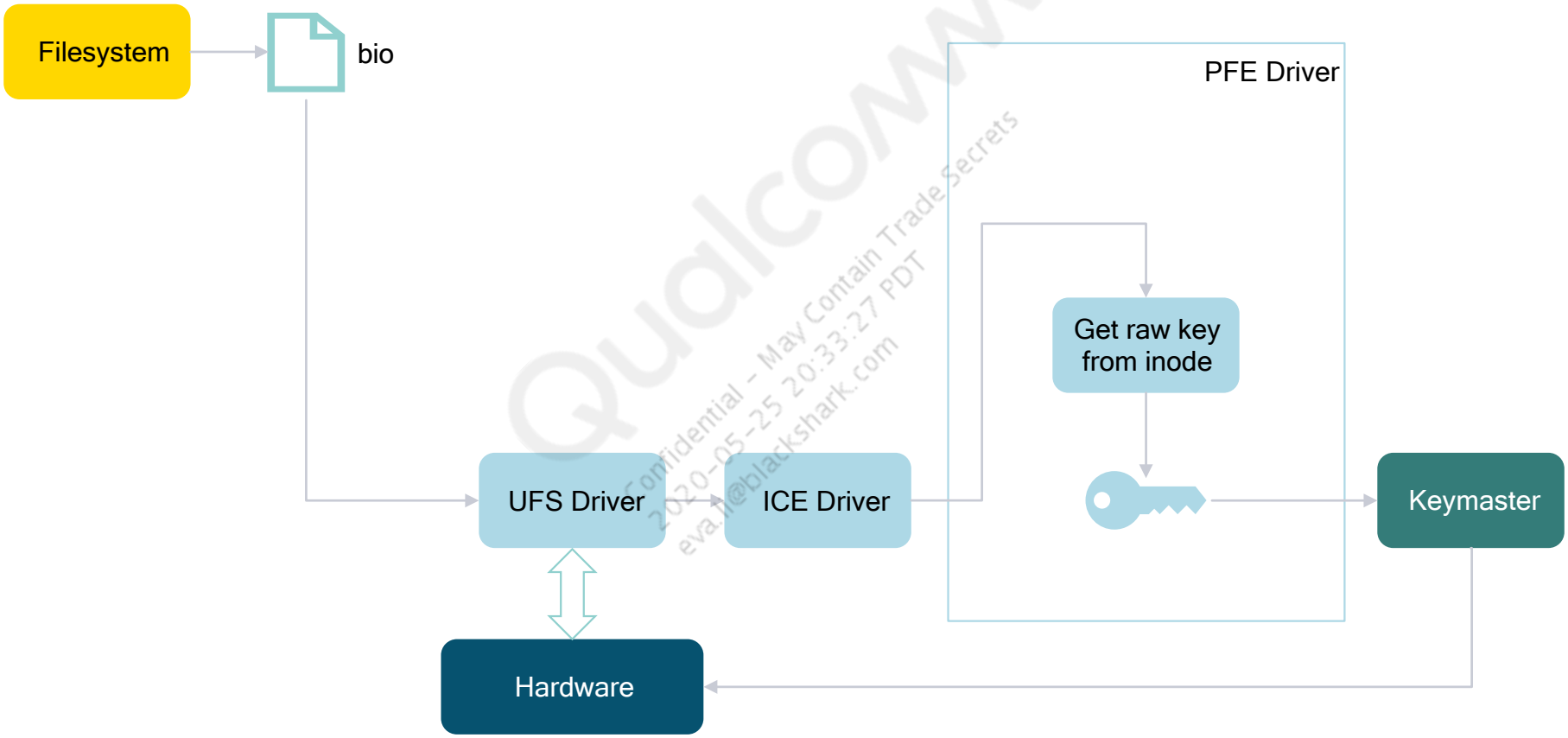
Key Derivation – Class Key

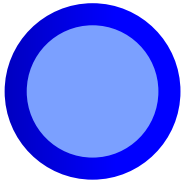
```
struct fscrypt_context {
    u8 format;
    u8 contents_encryption_mode;
    u8 filenames_encryption_mode;
    u8 flags;
    u8 master_key_descriptor[FS_KEY_DESCRIPTOR_SIZE];
    u8 nonce[FS_KEY_DERIVATION_NONCE_SIZE];
} __packed;
```

```
struct fscrypt_info {
    u8 ci_mode;
    u8 ci_data_mode;
    u8 ci_filename_mode;
    u8 ci_flags;
    struct crypto_skcipher *ci_ctfm;
    struct crypto_cipher *ci_essiv_tfm;
    u8 ci_master_key[FS_KEY_DESCRIPTOR_SIZE];
    u8 ci_raw_key[FS_MAX_KEY_SIZE];
};
```



Set key to ICE





Section 4

Metadata Encryption (ME)

Metadata Encryption (ME)

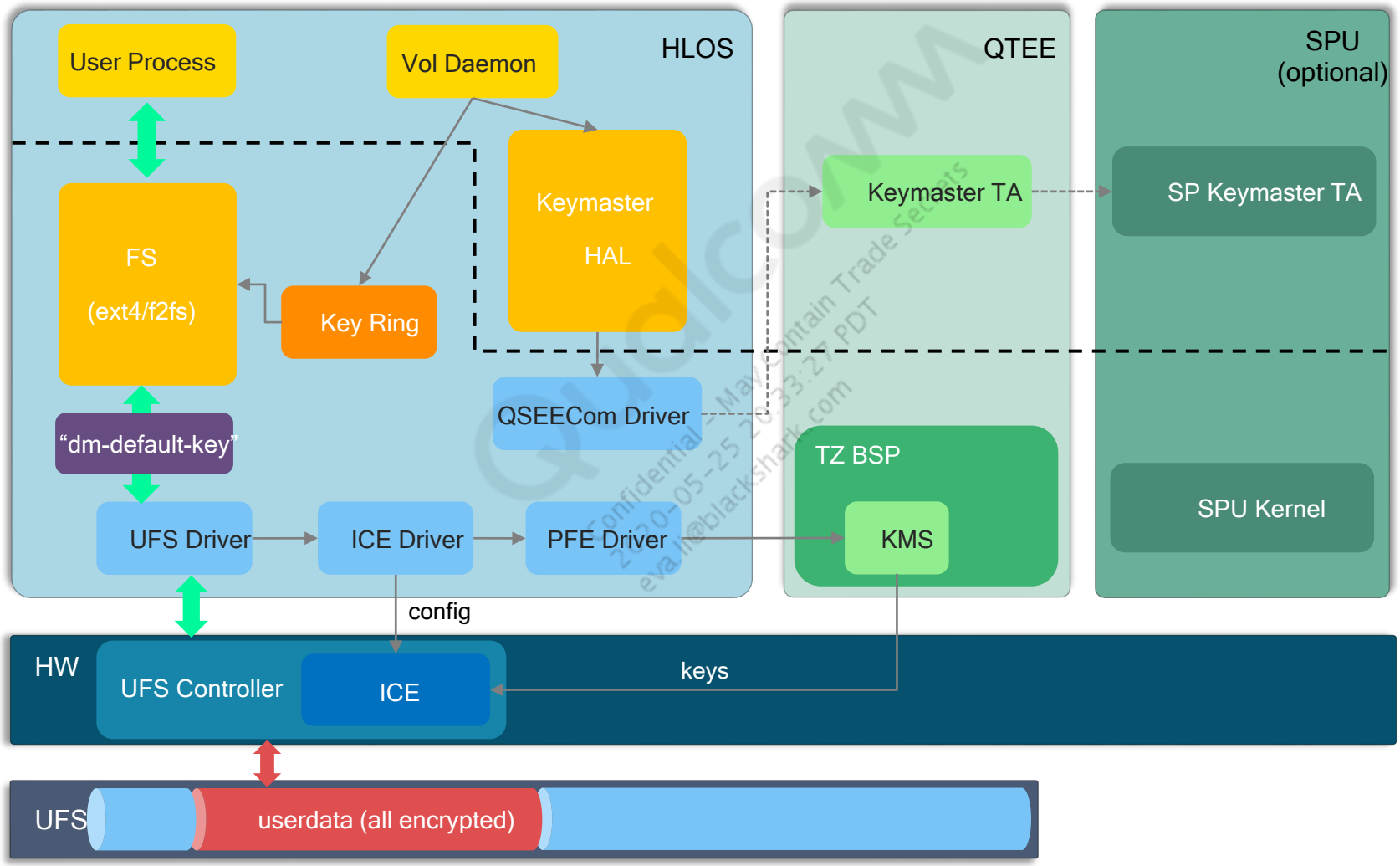
When FBE is used, filesystem metadata is not encrypted, such as directory layouts, file sizes, permissions, and creation/modification times.

With metadata encryption, a single key present at boot time encrypts whatever content is not encrypted by FBE.

Requirements:

- New partition added : “metadata”
- ME can only be set up when the data partition is first formatted.
- Hardware needs to support ICE (Inline Crypto Engine), which is same with FBE.
- The dm-default-key module must be present and enabled in the kernel.
- It’s only for new devices, not available for OTA.

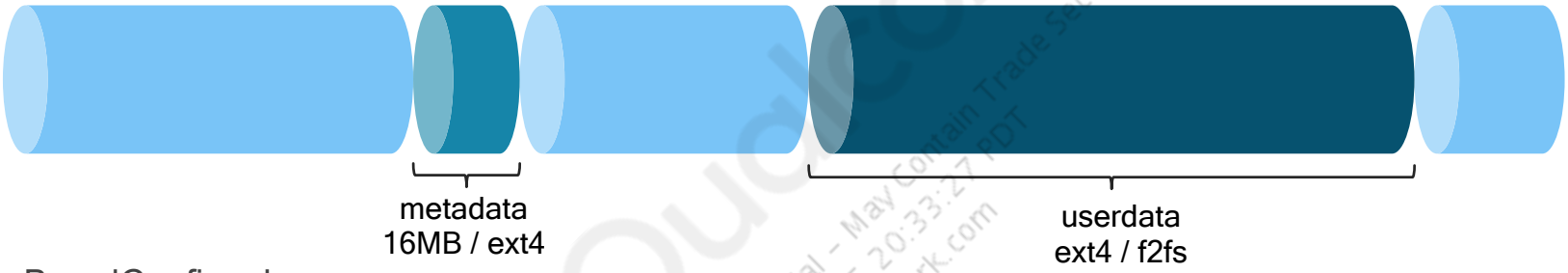
Architecture (ME)



Set up ME (1/2)

A new partition “metadata” is added for storing metadata encryption key.

- No encryption on metadata partition.
- The ME key is saved in metadata partition.
- Must be 16 MB, and recommend using EXT4 filesystem.



<BoardConfig.mk>

```
BOARD_USES_METADATA_PARTITION := true
BOARD_METADATAIMAGE_PARTITION_SIZE := 16777216
```

<fstab.qcom>

```
#<src>                                          <mnt_point>    <type>  <mnt_flags and options>
<fs_mgr_flags>
/dev/block/bootdevice/by-name/metadata     /metadata       ext4
noatime,nosuid,nodev,discard     wait,formattable
```

Set up ME (2/2)

- Set “keydirectory=” for userdata partition in fstab.qcom

```
/dev/block/bootdevice/by-name/userdata                  /data             f2fs
noatime,nosuid,nodev,discard
latemount,wait,check,fileencryption=ice,keydirectory=/metadata/vold/metadata_encryption,quota
,formattable
```

- The "dm-default-key" module must be enabled in the kernel.

```
CONFIG_DM_DEFAULT_KEY=y
kernel/msm-4.14/drivers/md/dm-default-key.c
```

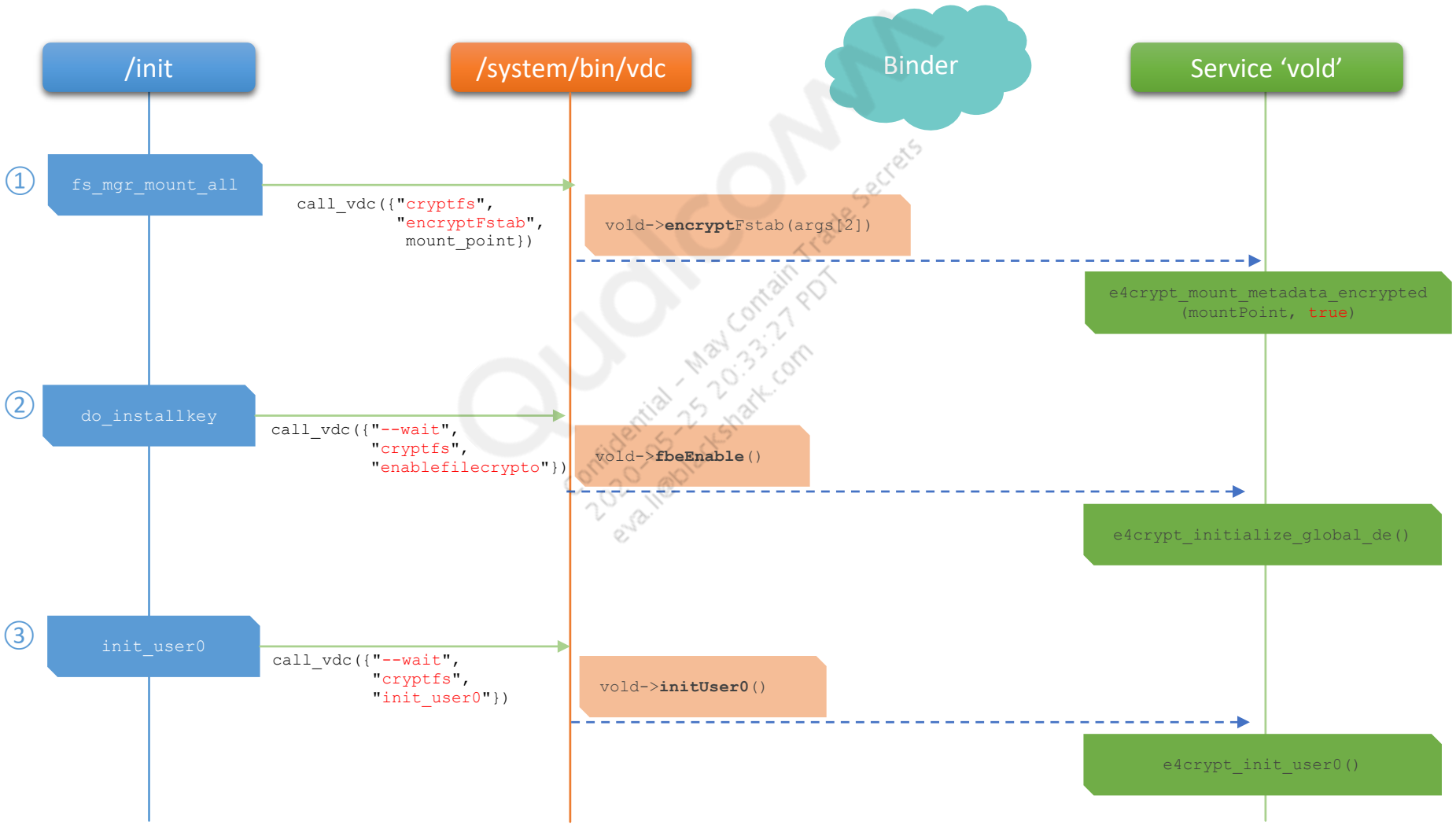
- init sequence
  - vold must be running before /data is mounted.

```
# We need vold early for metadata encryption
on early-fs
    start vold
```

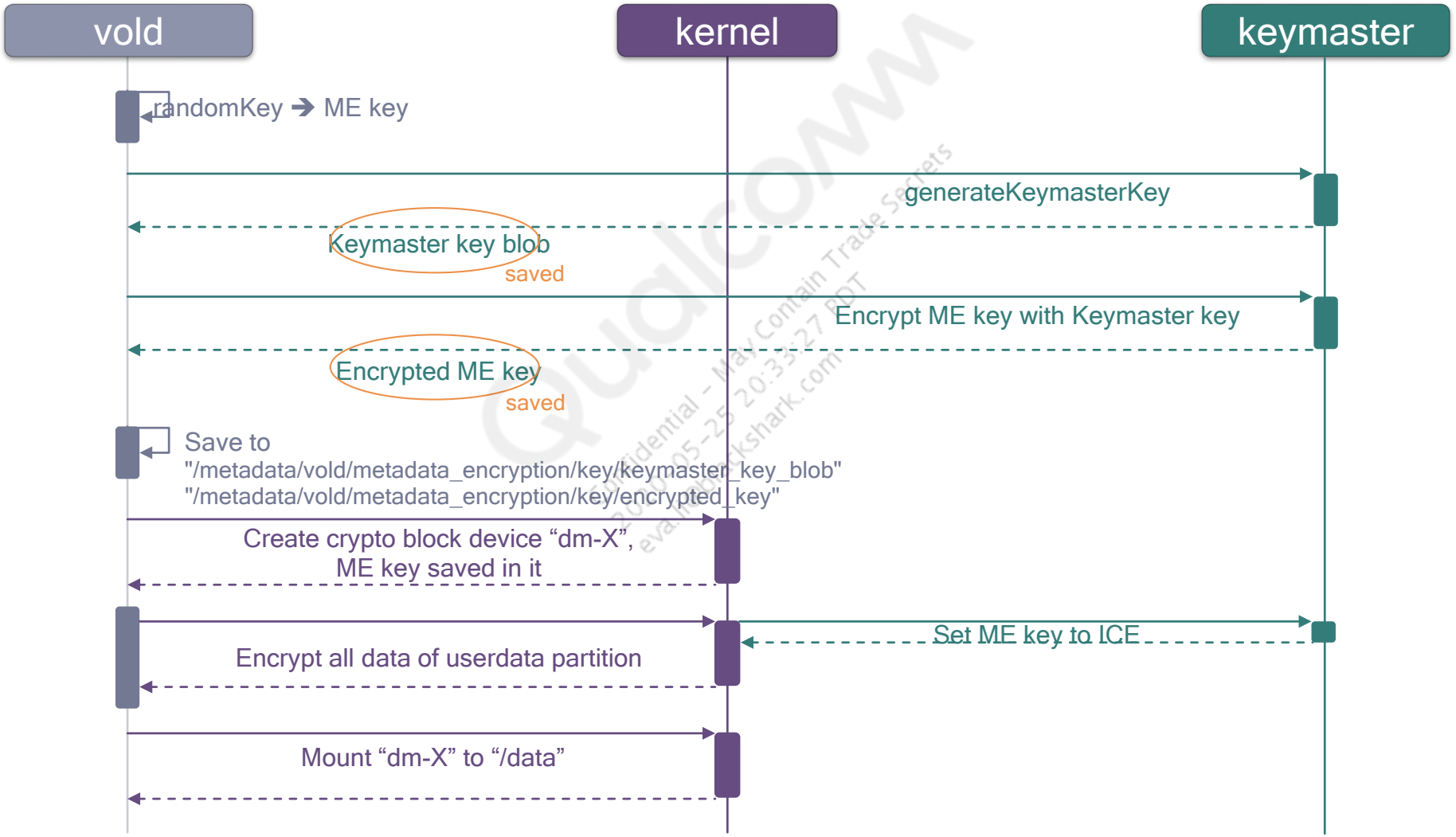
  - Keymaster must be running and ready before init attempts to mount /data.

```
on late-fs
   …
    # Wait for keymaster
    exec_start wait_for_keymaster

    # Mount RW partitions which need run fsck
    mount_all /vendor/etc/fstab.${ro.boot.hardware.platform} --late
```

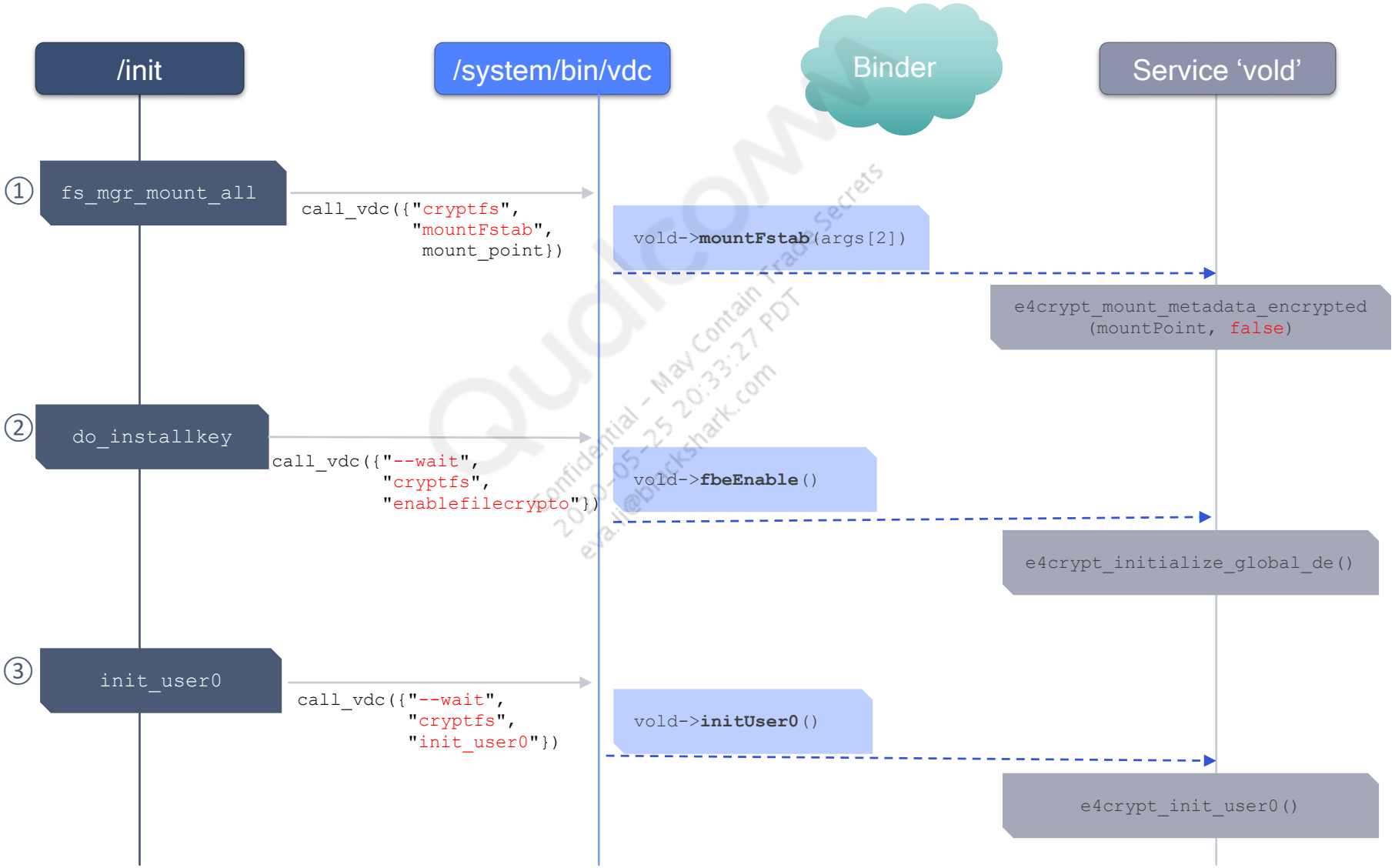
1st Boot (1/2)



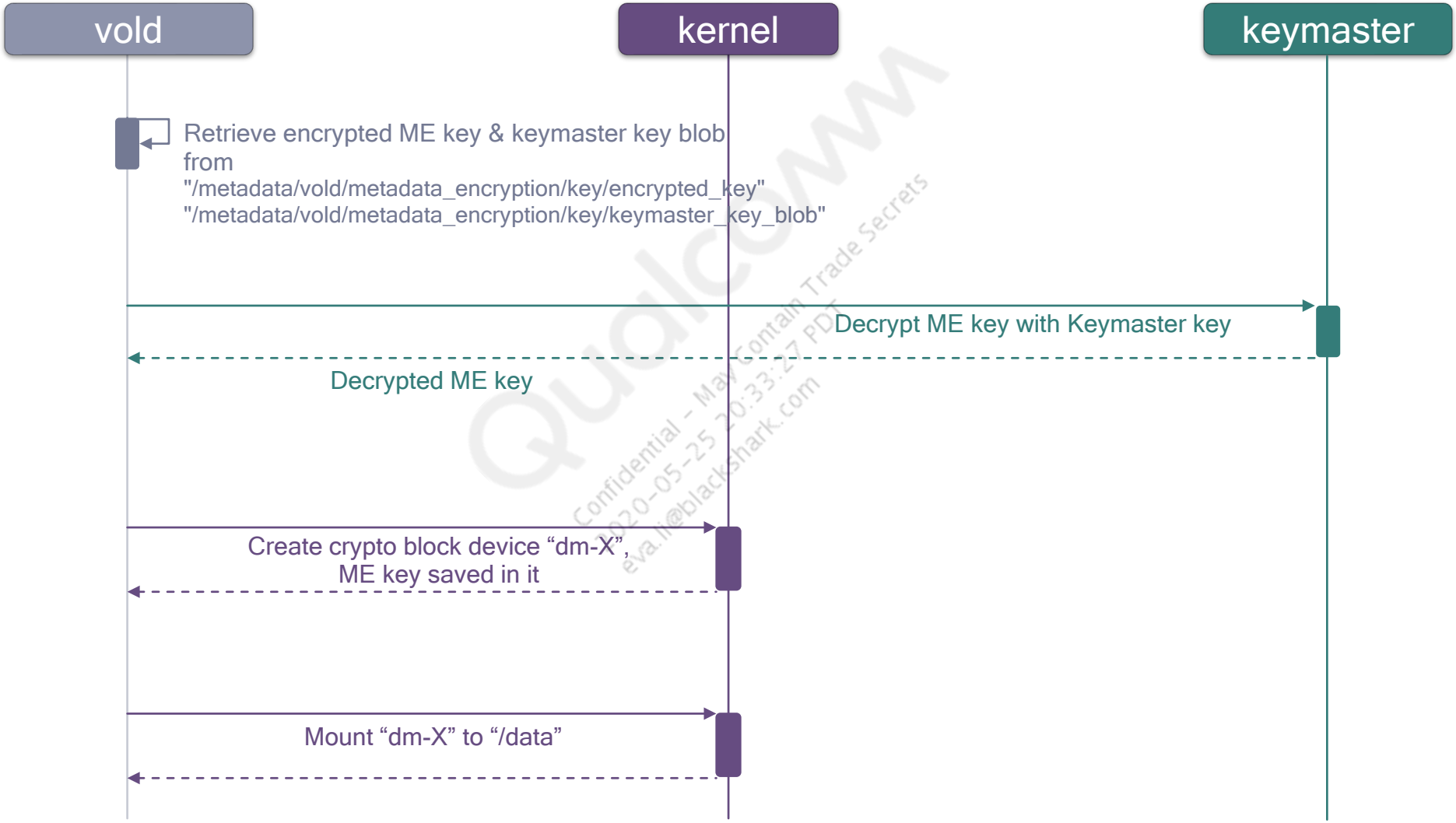
1st Boot (2/2)



Subsequent Boot (1/2)



Subsequent Boot (2/2)



Access Data (1/2)

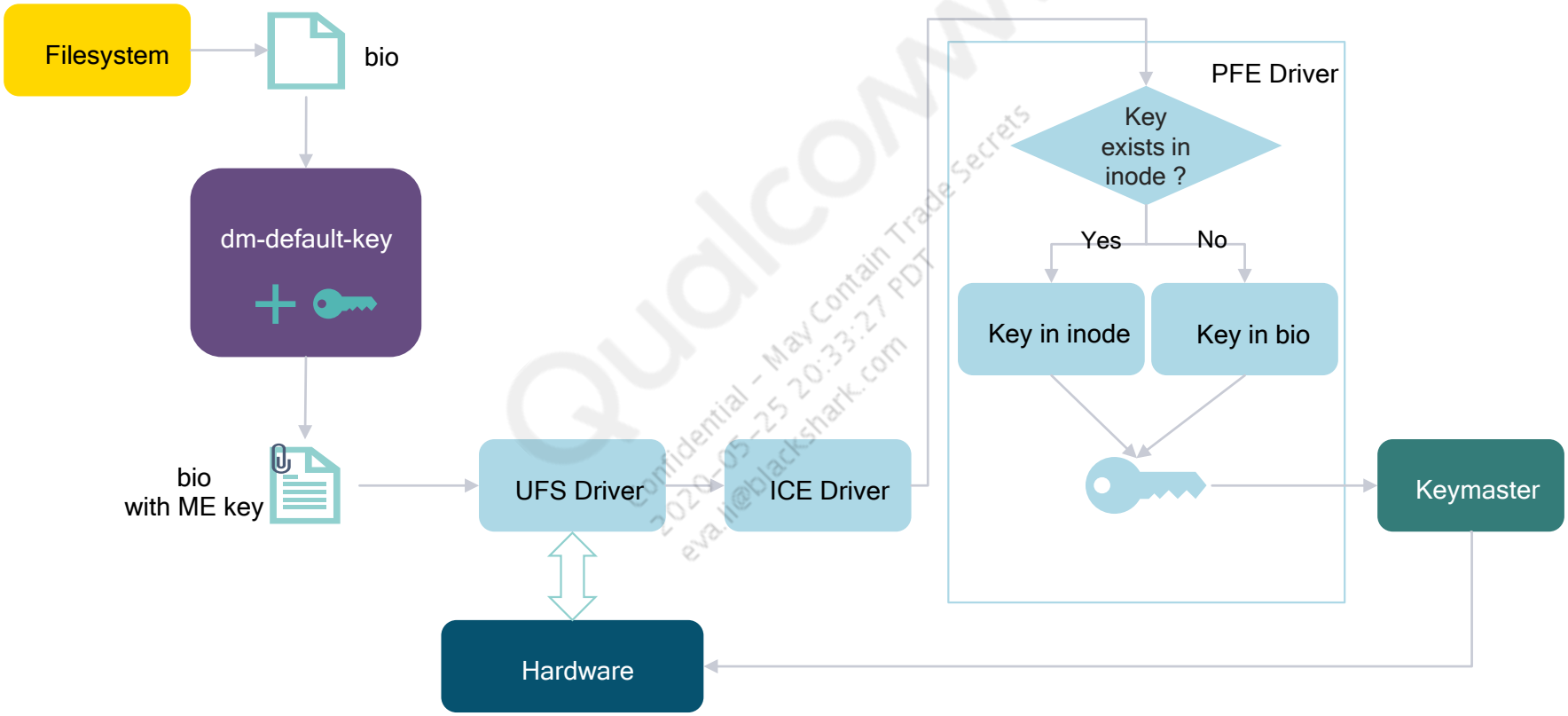
When accessing the data of “dm-X”, each bio will be attached with ME key by “dm-default-key”.

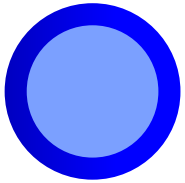
The ME key is attached at “bio->bi_crypt_key”

PFE driver prepares key setting before data transfer started.

- If there is no inode for this bio, or inode exits but no encryption setting in it, use “bio->bi_crypt_key”.
- If there is encryption information in the inode of this bio, use the key saved in inode.
- Set the selected key settings to ICE via scm call.

Access Data (2/2)





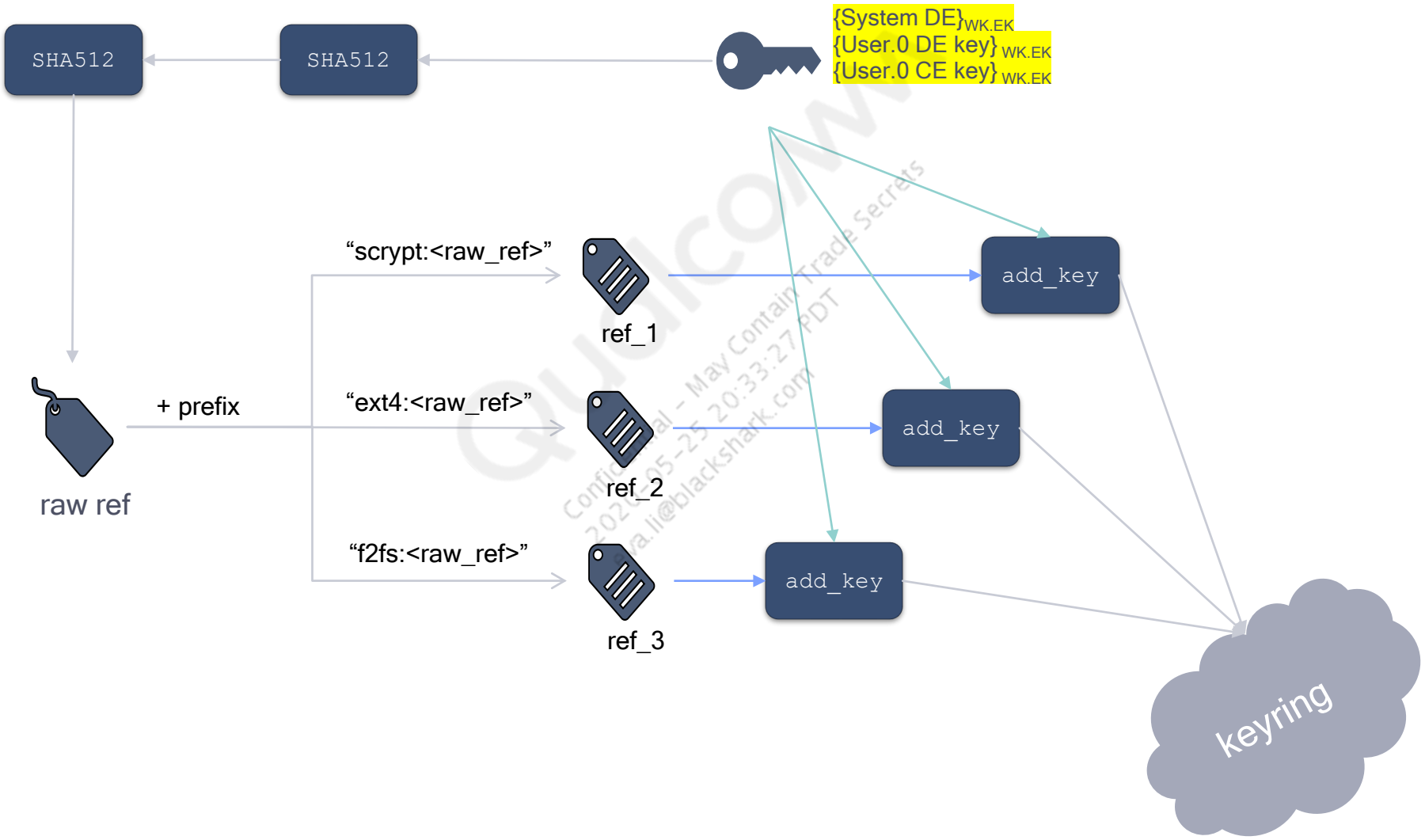
Section 5

Wrapped Key

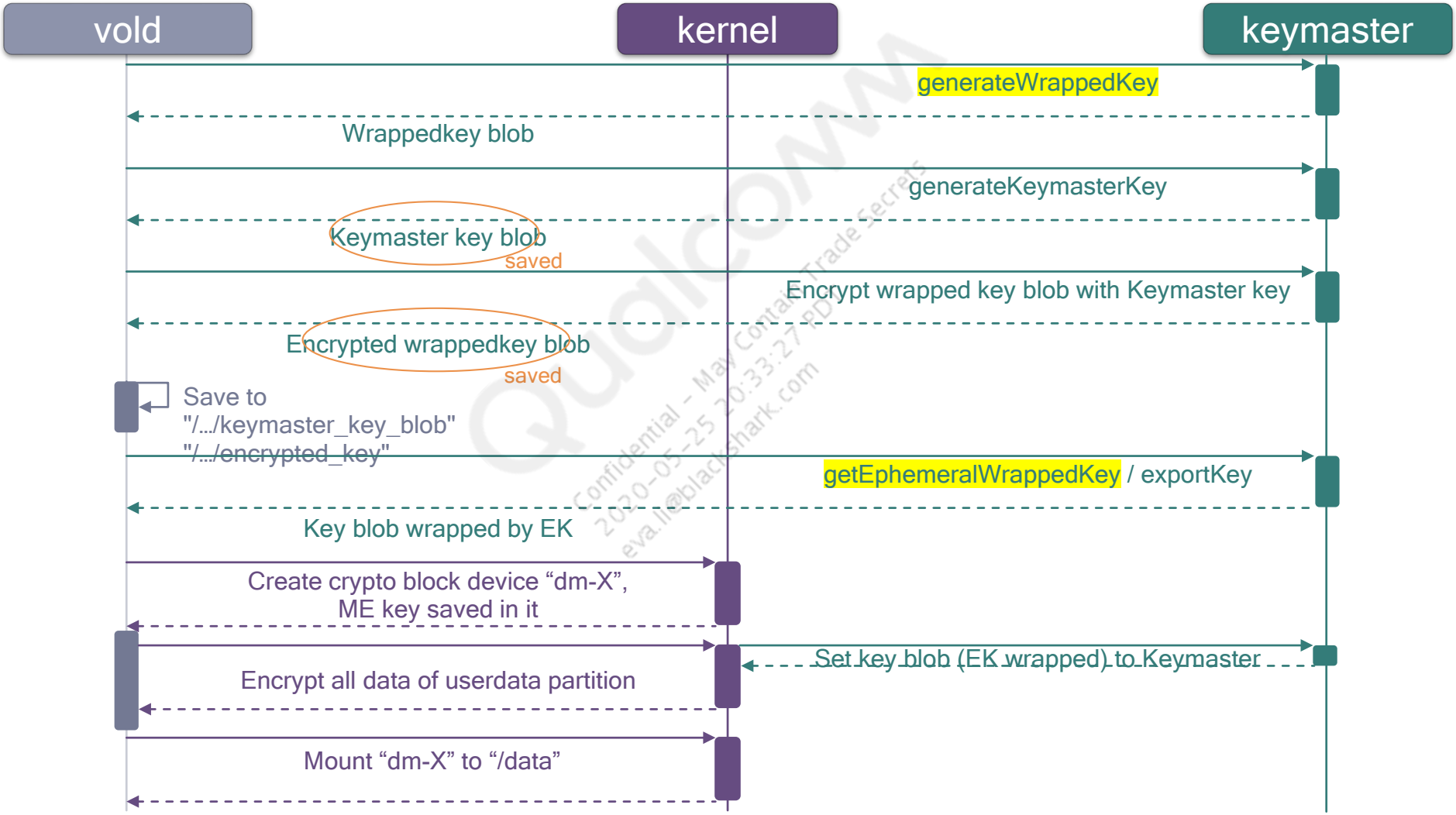
Wrapped Key

- FBE greatly enhances user experience and security by providing user key isolation and direct boot.
- One of drawbacks of FBE in terms of security has been that the class keys are present in the clear in HLOS once the user provides credentials.
- Wrappedkey provides a mechanism to protect all the FBE data keys on secure side without compromising on performance. FBE keys are never present in the clear in HLOS.
  - Instead of returning the actual FBE key to HLOS, the secure environment KM returns a wrapped key blob.
  - KM generates per-boot ephemeral keys (EK) to wrap the FBE class keys.
  - FBE class keys wrapped with EK are cached in HLOS (vold & kernel) and these key blobs are valid only for the current boot.
  - The EK are cached in the secure environment (SPU/TEE) when the user is active, when secondary users are shutdown the EK is removed from the cache, this ensures all the wrapped blobs in HLOS are invalid.
- Add keyword “wrappedkey” to userdata partition entry in fstab.qcom.

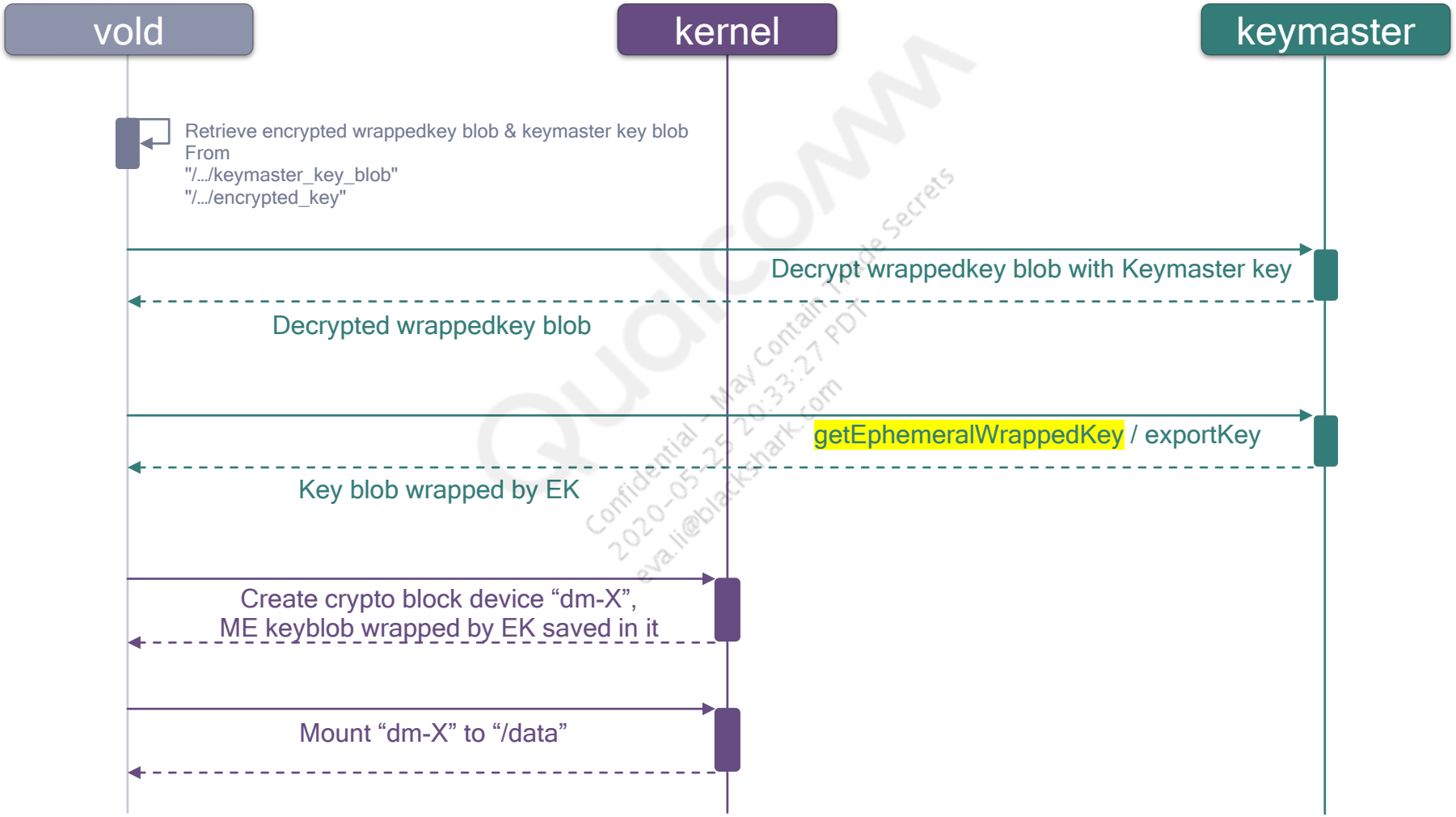
Add FBE keys – Wrappedkey enabled



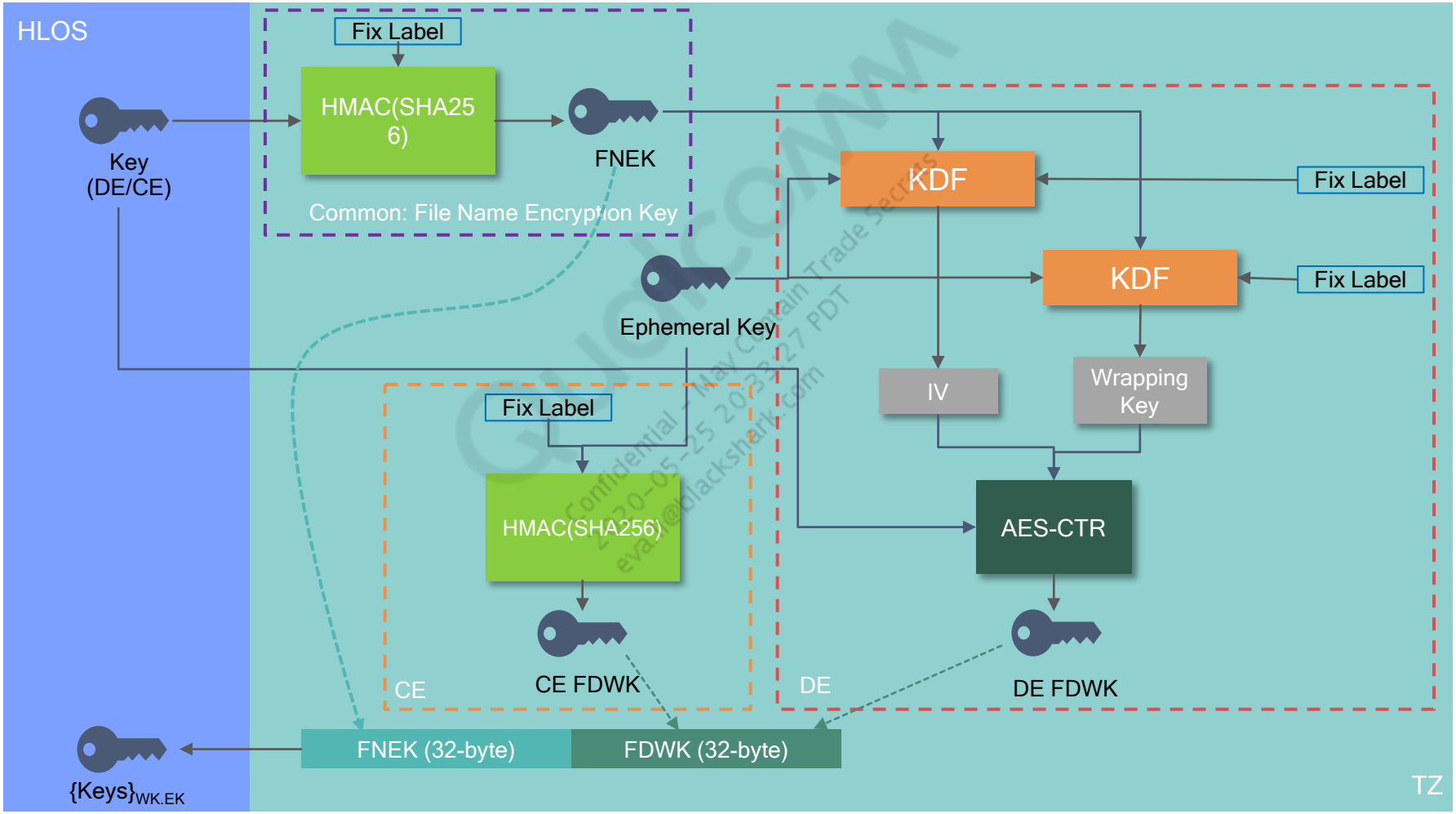
1st Boot



Subsequent Boot



Wrap Keys (1/2)



Wrap Keys (2/2)

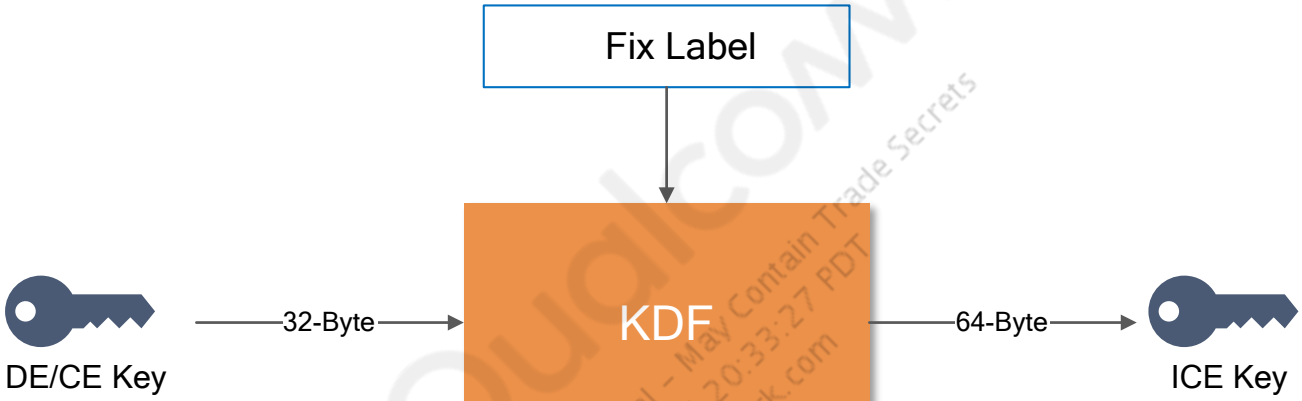
FNEK – File Name Encryption Key

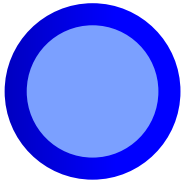
- Use class key(DE/CE/ME) with a unique label to do HMAC, and generate a file name encryption key.
- This key is returned in the clear to HLOS side.

FDWK – File Data Wrapped encryption Key

- The FDWK is generated, and returned with FNEK to HLOS side.
- A limited number of slots are available for CE Keys. The CE key is cached in one of the empty slots when being exported.
- If the number of CE keys requested is more than the key slots available, it falls back to the DE/ME mechanism.
- During set_ice_key, TZ checks if the key is present in one of the slots, and retrieve the appropriate CE key.

ICE Key





Section 6

Code Snippets

FBE Boot - do_installkey (1)

```
<init>

static Result<Success> do_installkey(const BuiltinArguments& args) {
    if (!is_file_crypto()) return Success();                        // If not FBE, return without error

    auto unencrypted_dir = args[1] + e4crypt_unencrypted_folder;  // "/data" + "/unencrypted"
    if (!make_dir(unencrypted_dir, 0700) && errno != EEXIST) {    // Create the directory "/data/unencrypted"
        return ErrnoError() << "Failed to create " << unencrypted_dir;
    }
    return ExecWithRebootOnFailure(                                 // exec "vdc" to call vold->fbeEnable() remotely
        "enablefilecrypto_failed",
        {{"exec", "/system/bin/vdc", "--wait", "cryptfs", "enablefilecrypto"}, args.context});
}

=============================================================================================

<vdc>

               if (args[0] == "cryptfs" && args[1] == "enablefilecrypto") {
                       checkStatus(vold->fbeEnable());
                   }
```

FBE Boot - do_installkey (2)

```
VoldNativeService::fbeEnable()
    e4crypt_initialize_global_de()
        LOG(INFO) << "e4crypt_initialize_global_de"
        PolicyKeyRef device_ref
        retrieveAndInstallKey(true, kEmptyAuthentication, device_key_path, device_key_temp, &device_ref.key_raw_ref)

            if (pathExists(key_path))    // "/data/unencrypted/key" exists ➔ normal boot

                LOG(DEBUG) << "Key exists, using: " << key_path

                retrieveKey(key_path, key_authentication, &key)
                    readFileToString(dir + "/" + kFn_encrypted_key, &encryptedMessage)                // "/data/unencrypted/key/encrypted_key"
                    decryptWithKeymasterKey(keymaster, dir, keyParams, authToken, encryptedMessage, key)

            else                         // "/data/unencrypted/key" doesn't exist ➔ 1st boot branch

                LOG(INFO) << "Creating new key in " << key_path

                randomKey(&key)

                storeKeyAtomically(key_path, tmp_path, key_authentication, key)

                    storeKey(tmp_path, auth, key)
                        mkdir(dir.c_str(), 0700)     // "/data/unencrypted/temp"
                        writeStringToFile(kCurrentVersion, dir + "/" + kFn_version)     // "/data/unencrypted/temp/version"
                        generateKeymasterKey(keymaster, auth, appId, &kmKey)
                        writeStringToFile(kmKey, dir + "/" + kFn_keymaster_key_blob)    // "/data/unencrypted/temp/keymaster_key_blob"
                        encryptWithKeymasterKey(keymaster, dir, keyParams, authToken, key, &encryptedKey)
                        writeStringToFile(encryptedKey, dir + "/" + kFn_encrypted_key)  // "/data/unencrypted/temp/encrypted_key"

                    rename(tmp_path.c_str(), key_path.c_str()) // "/data/unencrypted/temp" ==> "/data/unencrypted/key"

                    LOG(DEBUG) << "Created key: " << key_path
```

FBE Boot - do_installkey (3)

```
VoldNativeService::fbeEnable()
    e4crypt_initialize_global_de()
        LOG(INFO) << "e4crypt_initialize_global_de"
        PolicyKeyRef device_ref
        retrieveAndInstallKey(true, kEmptyAuthentication, device_key_path, device_key_temp, &device_ref.key_raw_ref)
            if (pathExists(key_path)) // "/data/unencrypted/key" exists  ➔ normal boot
                LOG(DEBUG) << "Key exists, using: " << key_path
                return
            else                      // "/data/unencrypted/key" doesn't exist ➔ 1st boot branch
                LOG(INFO) << "Creating new key in " << key_path
                mkdir

            installKey(key, key_ref)  // key_ref : &device_ref.key_raw_ref
                fillKey(key, &ext4_key)
                    memcpy(ext4_key->raw, key.data(), key.size());
                *raw_ref = generateKeyRef(ext4_key.raw, ext4_key.size)
                    SHA512_Init(&c);
                    SHA512_Update(&c, key, length);
                    unsigned char key_ref1[SHA512_DIGEST_LENGTH];
                    SHA512_Final(key_ref1, &c);

                    SHA512_Init(&c);
                    SHA512_Update(&c, key_ref1, SHA512_DIGEST_LENGTH);
                    unsigned char key_ref2[SHA512_DIGEST_LENGTH];
                    SHA512_Final(key_ref2, &c);
                    return std::string((char*)key_ref2, EXT4_KEY_DESCRIPTOR_SIZE)
                for (char const* const* name_prefix = NAME_PREFIXES; *name_prefix != nullptr; name_prefix++)
                    ref = keyname(*name_prefix, *raw_ref)
                    key_id = add_key("logon", ref.c_str(), (void*)&ext4_key, sizeof(ext4_key), device_keyring)
                    LOG(DEBUG) << "Added key " << key_id << " (" << ref << ") to keyring " << device_keyring << " in process " << getpid()

        get_data_file_encryption_modes(&device_ref)
            fs_mgr_get_file_encryption_modes(rec, &contents_mode, &filenames_mode)

        std::string modestring = device_ref.contents_mode + ":" + device_ref.filenames_mode;  // "ice:aes-256-cts"
        std::string mode_filename = std::string("/data") + e4crypt_key_mode;                  // "/data//unencrypted/mode"
        writeStringToFile(modestring, mode_filename)
        std::string ref_filename = std::string("/data") + e4crypt_key_ref;                    // "/data//unencrypted/ref
        writeStringToFile(device_ref.key_raw_ref, ref_filename)

        LOG(INFO) << "Wrote system DE key reference to:" << ref_filename
```

FBE Boot - mkdir

```
static Result<Success> do_mkdir(const BuiltinArguments& args) {
<snip>
    if (e4crypt_is_native()) {
        if (e4crypt_set_directory_policy(args[1].c_str())) {
            return reboot_into_recovery(
                {"--prompt_and_wipe_data", "--reason=set_policy_failed:"s + args[1]});
        }
    }
    return Success();
}

int e4crypt_set_directory_policy(const char* dir)
{
<snip>
    std::vector<std::string> directories_to_exclude = {
        "lost+found",
        "system_ce", "system_de",
        "misc_ce", "misc_de",
        "vendor_ce", "vendor_de",
        "media",
        "data", "user", "user_de",
    };
    std::string prefix = "/data/";
    for (auto d: directories_to_exclude) {  // Exclude the dirs in the list.
        if ((prefix + d) == dir) {
            LOG(INFO) << "Not setting policy on " << dir;
            return 0;
        }
    }
    return set_system_de_policy_on(dir);
}

set_system_de_policy_on(dir)
    e4crypt_policy_ensure(dir, policy.c_str(), policy.length(), modes[0].c_str(), modes.size() >= 2 ? modes[1].c_str() : "aes-256-cts");
        e4crypt_policy_set(directory, policy, policy_length, contents_mode, filenames_mode)
            ioctl(fd, EXT4_IOC_SET_ENCRYPTION_POLICY, &eep)
```

FBE Boot - init_user0 (1)

```
VoldNativeService::initUser0()
    e4crypt_init_user0()

        LOG(DEBUG) << "e4crypt_init_user0"

        if (!android::vold::pathExists(get_de_key_path(0)))          // "/data/misc/vold/user_keys/de/0"

            create_and_install_user_keys(0, false)
                KeyBuffer de_key, ce_key
                android::vold::randomKey(&de_key)
                android::vold::randomKey(&ce_key)

                storeKeyAtomically(ce_key_path, user_key_temp, kEmptyAuthentication, ce_key)   // "/data/misc/vold/user_keys/temp"
                    storeKey(tmp_path, auth, key)
                        mkdir(dir.c_str(), 0700)
                        generateKeymasterKey(keymaster, auth, appId, &kmKey)
                        writeStringToFile(kmKey, dir + "/" + kFn_keymaster_key_blob)    // "/data/misc/vold/user_keys/temp/keymaster_key_blob"
                        encryptWithKeymasterKey(keymaster, dir, keyParams, authToken, key, &encryptedKey)
                        writeStringToFile(encryptedKey, dir + "/" + kFn_encrypted_key)  // "/data/misc/vold/user_keys/temp/encrypted_key"
                    rename(tmp_path.c_str(), key_path.c_str()  // "/data/misc/vold/user_keys/temp" ==> "/data/misc/vold/user_keys/ce/0"

                storeKeyAtomically(get_de_key_path(user_id), user_key_temp, kEmptyAuthentication, de_key)
                    storeKey(tmp_path, auth, key)
                        mkdir(dir.c_str(), 0700)
                        generateKeymasterKey(keymaster, auth, appId, &kmKey)
                        writeStringToFile(kmKey, dir + "/" + kFn_keymaster_key_blob)    // "/data/misc/vold/user_keys/temp/keymaster_key_blob"
                        encryptWithKeymasterKey(keymaster, dir, keyParams, authToken, key, &encryptedKey)
                        writeStringToFile(encryptedKey, dir + "/" + kFn_encrypted_key)  // "/data/misc/vold/user_keys/temp/encrypted_key"
                    rename(tmp_path.c_str(), key_path.c_str()  // "/data/misc/vold/user_keys/temp" ==> "/data/misc/vold/user_keys/de/0>"

                android::vold::installKey(de_key, &de_raw_ref)
                    *raw_ref = generateKeyRef(ext4_key.raw, ext4_key.size)
                    for (char const* const* name_prefix = NAME_PREFIXES; *name_prefix != nullptr; name_prefix++)
                        add_key("logon", ref.c_str(), (void*)&ext4_key, sizeof(ext4_key), device_keyring)

                android::vold::installKey(ce_key, &ce_raw_ref)
```

FBE Boot - init_user0 (2)

```
load_all_de_keys()
    de_dir = user_key_dir + "/de";     // "/data/misc/vold/user_keys/de"
    dirp = opendir(de_dir.c_str())
    for (;;)
        entry = readdir(dirp.get())
        user_id = std::stoi(entry->d_name)
        if (s_de_key_raw_refs.count(user_id) == 0)
            key_path = de_dir + "/" + entry->d_name
            retrieveKey(key_path, kEmptyAuthentication, &key)
                        readFileToString(dir + "/" + kFn_encrypted_key, &encryptedMessage)
                        decryptWithKeymasterKey(keymaster, dir, keyParams, authToken, encryptedMessage, key)
            installKey(key, &raw_ref)
                *raw_ref = generateKeyRef(ext4_key.raw, ext4_key.size)
                for (char const* const* name_prefix = NAME_PREFIXES; *name_prefix != nullptr; name_prefix++)
                    key_id = add_key("logon", ref.c_str(), (void*)&ext4_key, sizeof(ext4_key), device_keyring)
            LOG(DEBUG) << "Installed de key for user " << user_id
```

```
e4crypt_prepare_user_storage("", 0, 0, android::os::IVold::STORAGE_FLAG_DE)

    if (flags & android::os::IVold::STORAGE_FLAG_DE)

        ensure_policy(de_ref, system_de_path)                               // "/data/system_de/<userID>"
            e4crypt_policy_ensure(path.c_str(), key_ref.key_raw_ref.data(),
                                                key_ref.key_raw_ref.size(),
                                                key_ref.contents_mode.c_str(),
                                                key_ref.filenames_mode.c_str()
                contents_mode = EXT4_ENCRYPTION_MODE_PRIVATE
                filenames_mode = EXT4_ENCRYPTION_MODE_AES_256_CTS
                if (is_empty)
                    e4crypt_policy_set(directory, policy, policy_length, contents_mode, filenames_mode)
                else
                    e4crypt_policy_check(directory, policy, policy_length, contents_mode, filenames_mode)

        ensure_policy(de_ref, misc_de_path)                                 // "/data/misc_de/<userID>"
        ensure_policy(de_ref, vendor_de_path)                               // "/data/vendor_de/<userID>"
```

FBE - Create File

```
fopen
    open(file, mode_flags, DEFFILEMODE);

======== sys call ========

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
    do_sys_open(AT_FDCWD, filename, flags, mode);
        do_filp_open(dfd, tmp, &op);
            path_openat(&nd, op, flags | LOOKUP_RCU);
                do_last(nd, file, op, &opened)
                    lookup_open(nd, &path, file, op, got_write, opened);
                        dir_inode->i_op->create(dir_inode, dentry, mode, open_flag & O_EXCL);


ext4_create
    // Create new file inode
    ext4_new_inode_start_handle(dir, mode, &dentry->d_name, 0, NULL, EXT4_HT_DIR, credits);
        __ext4_new_inode
            inode = new_inode(sb);
    // Created encrypted file name
    ext4_add_nondir(handle, dentry, inode);
        ext4_add_entry(handle, dentry, inode);
            ext4_fname_setup_filename(dir, &dentry->d_name, 0, &fname)
                fscrypt_setup_filename(dir, iname, lookup, &name);
                    fscrypt_get_encryption_info(dir);
                        fname_encrypt(dir, iname, fname->crypto_buf.name, fname->crypto_buf.len);
                            crypto_wait_req(crypto_skcipher_encrypt(req), &wait);
```

FBE Key Derivation - PFK

```
ext4_new_inode                                              // started here, whatever creating a file or directory
    __ext4_new_inode
        inode = new_inode(sb)                               // create new inode for new file/dir

        fscrypt_inherit_context(dir, inode, handle, true)   // inherit encryption context
            fscrypt_get_encryption_info(parent)             // first, get encryption info of parent dir

            ci = parent->i_crypt_info;                      // based on parent's info, make an encryption context copy
            ctx.format = FS_ENCRYPTION_CONTEXT_FORMAT_V1;
            ctx.contents_encryption_mode = ci->ci_data_mode;
            ctx.filenames_encryption_mode = ci->ci_filename_mode;
            ctx.flags = ci->ci_flags;
            memcpy(ctx.master_key_descriptor, ci->ci_master_key, FS_KEY_DESCRIPTOR_SIZE);
            get_random_bytes(ctx.nonce, FS_KEY_DERIVATION_NONCE_SIZE);
            parent->i_sb->s_cop->set_context(child, &ctx, sizeof(ctx), fs_data)  // ext4_set_context, set new context to child inode

            fscrypt_get_encryption_info(child)
                inode->i_sb->s_cop->get_context(inode, &ctx, sizeof(ctx))       // ext4_get_context, get child's context

                crypt_info = kmem_cache_alloc(fscrypt_info_cachep, GFP_NOFS)    // based on just saved context, make a new crypt_info copy
                crypt_info->ci_flags = ctx.flags;
                crypt_info->ci_data_mode = ctx.contents_encryption_mode;
                crypt_info->ci_filename_mode = ctx.filenames_encryption_mode;
                memcpy(crypt_info->ci_master_key, ctx.master_key_descriptor, sizeof(crypt_info->ci_master_key));

                validate_user_key(crypt_info, &ctx, FS_KEY_DESC_PREFIX, keysize)
                    description = kasprintf(GFP_NOFS, "%s%*phN", prefix, FS_KEY_DESCRIPTOR_SIZE, ctx->master_key_descriptor)
                    keyring_key = request_key(&key_type_logon, description, NULL);  // get key by description(like, "scrypt:XXXXXXXXXXXXXXXX")
                    ukp = user_key_payload_locked(keyring_key)
                    master_key = (struct fscrypt_key *)ukp->data                    // extract master key
                    derive_key_aes(ctx->nonce, master_key, crypt_info->ci_raw_key)  // derive a key
                                                                                    // "master_key" is source key
                                                                                    // output saved in "crypt_info->ci_raw_key"
                cmpxchg(&inode->i_crypt_info, NULL, crypt_info)  // after derive done, save updated info data to "inode->i_crypt_info"
```

FBE Key Derivation – Class Key

```
ext4_new_inode_start_handle                                  // started here
    __ext4_new_inode
        inode = new_inode(sb)                                // create new inode for new file/dir

        fscrypt_inherit_context(dir, inode, handle, true)    // inherit encryption context
            fscrypt_get_encryption_info(parent)              // first, get encryption info of parent dir

            ci = parent->i_crypt_info;                       // based on parent's info, make an encryption context copy
            ctx.format = FS_ENCRYPTION_CONTEXT_FORMAT_V1;
            ctx.contents_encryption_mode = ci->ci_data_mode;
            ctx.filenames_encryption_mode = ci->ci_filename_mode;
            ctx.flags = ci->ci_flags;
            memcpy(ctx.master_key_descriptor, ci->ci_master_key, FS_KEY_DESCRIPTOR_SIZE);
            get_random_bytes(ctx.nonce, FS_KEY_DERIVATION_NONCE_SIZE);
            parent->i_sb->s_cop->set_context(child, &ctx, sizeof(ctx), fs_data)  // ext4_set_context, set new context to child inode

            fscrypt_get_encryption_info(child)
                inode->i_sb->s_cop->get_context(inode, &ctx, sizeof(ctx))        // ext4_get_context, get child's context

                crypt_info = kmem_cache_alloc(fscrypt_info_cachep, GFP_NOFS)     // based on just saved context, make a new crypt_info copy
                crypt_info->ci_flags = ctx.flags;
                crypt_info->ci_data_mode = ctx.contents_encryption_mode;
                crypt_info->ci_filename_mode = ctx.filenames_encryption_mode;
                memcpy(crypt_info->ci_master_key, ctx.master_key_descriptor, sizeof(crypt_info->ci_master_key));

                validate_user_key(crypt_info, &ctx, FS_KEY_DESC_PREFIX, keysize)
                    description = kasprintf(GFP_NOFS, "%s%*phN", prefix, FS_KEY_DESCRIPTOR_SIZE, ctx->master_key_descriptor)
                    keyring_key = request_key(&key_type_logon, description, NULL);  // get key by description(like, "scrypt:XXXXXXXXXXXXXXXX")
                    ukp = user_key_payload_locked(keyring_key)
                    master_key = (struct fscrypt_key *)ukp->data                     // extract master key
                    if (!is_private_data_mode(crypt_info))
                        derive_key_aes(ctx->nonce, master_key, crypt_info->ci_raw_key);
                    else
                        memcpy(crypt_info->ci_raw_key, master_key->raw, sizeof(crypt_info->ci_raw_key));
                cmpxchg(&inode->i_crypt_info, NULL, crypt_info)  // after derive done, save updated info data to "inode->i_crypt_info"
```

Decrypt file name (FBE)

```
// external/toybox/toys/posix/ls.c
ls_main
    listfiles(AT_FDCWD, TT.files);
        dirtree_recurse(indir, filter, dup(dirfd), DIRTREE_SYMFOLLOW*!!(flags&FLAG_L));
            fdopendir(node->dirfd)
            readdir(dir) // bionic/libc/bionic/dirent.cpp
                __readdir_locked(d);
                    __fill_DIR(d)
                        __getdents64(d->fd_, d->buff_, sizeof(d->buff_))

======== sys call ========

SYSCALL_DEFINE3(getdents64, unsigned int, fd, struct linux_dirent64 __user *, dirent, unsigned int, count)
    iterate_dir(f.file, &buf.ctx);
        file->f_op->iterate_shared(file, ctx);


ext4_readdir
    fscrypt_get_encryption_info(inode);
    fscrypt_fname_disk_to_usr(inode, 0, 0, &de_name, &fstr)
        fname_decrypt(inode, iname, oname);
            crypto_wait_req(crypto_skcipher_decrypt(req), &wait);
```

FBE Set Key (1)

kernel/msm-4.9/drivers/scsi/ufs/ufshcd.c

```
ufshcd_queuecommand
    ufshcd_map_sg
    ufshcd_vops_crypto_engine_cfg_start
    ufshcd_send_command
```

kernel/msm-4.9/drivers/scsi/ufs/ufshcd.h

```
ufshcd_vops_crypto_engine_cfg_start
    hba->var->crypto_vops->crypto_engine_cfg_start
```

kernel/msm-4.9/drivers/scsi/ufs/ufs-qcom.c

```
static struct ufs_hba_crypto_variant_ops ufs_hba_crypto_variant_ops = {
    .crypto_engine_cfg_start    = ufs_qcom_crytpo_engine_cfg_start,
};

ufs_qcom_crytpo_engine_cfg_start
    ufs_qcom_ice_cfg_start
```

kernel/msm-4.9/drivers/scsi/ufs/ufs-qcom-ice.c

```
ufs_qcom_ice_cfg_start
    qcom_host->ice.vops->config_start
```

kernel/msm-4.9/drivers/crypto/msm/ice.c

```
struct qcom_ice_variant_ops qcom_ice_ops = {
    .name               = "qcom",
    .config_start       = qcom_ice_config_start,
};

qcom_ice_config_start
    pfk_load_key_start(req->bio, &pfk_crypto_data, &is_pfe, async);
```

FBE Set Key (2)

```
kernel/msm-4.9/security/pfe/pfk.c

pfk_load_key_start
    pfk_get_key_for_bio(bio, &key_info, &algo_mode, is_pfe);
        inode = pfk_bio_get_inode(bio);
        which_pfe = pfk_get_pfe_type(inode);    // EXT4_CRYPT_PFE /
F2FS_CRYPT_PFE / INVALID_PFE
        return (*(pfk_parse_inode_ftable[which_pfe]))(bio, inode,
key_info, algo_mode, is_pfe);
    pfk_kc_load_key_start(key_info.key, key_info.key_size,
key_info.salt, key_info.salt_size, &key_index, async);

static const pfk_parse_inode_type pfk_parse_inode_ftable[] = {
    /* EXT4_CRYPT_PFE */ &pfk_ext4_parse_inode,
    /* F2FS_CRYPT_PFE */ &pfk_f2fs_parse_inode,
};
```

```
kernel/msm-4.9/security/pfe/pfk_ext4.c

pfk_ext4_parse_inode
    key_info->key = fscrypt_get_ice_encryption_key(inode);
        return &(inode->i_crypt_info->ci_raw_key[0]);

    key_info->key_size = fscrypt_get_ice_encryption_key_size(inode);
        return FS_AES_256_XTS_KEY_SIZE / 2;

    key_info->salt = fscrypt_get_ice_encryption_salt(inode);
        return &(inode->i_crypt_info-
>ci_raw_key[fscrypt_get_ice_encryption_key_size(inode)]);

    key_info->salt_size =
fscrypt_get_ice_encryption_salt_size(inode);
        return FS_AES_256_XTS_KEY_SIZE / 2;
    pfk_ext4_parse_cipher(inode, algo);
        *algo = ICE_CRYPTO_ALGO_MODE_AES_XTS;
```

```
kernel/msm-4.9/security/pfe/pfk_kc.c

pfk_kc_load_key_start
    kc_update_entry(entry, key, key_size, salt, salt_size);
        qti_pfk_ice_set_key(entry->key_index, entry->key, entry->salt, s_type);
```

```
kernel/msm-4.9/security/pfe/pfk_ice.c

qti_pfk_ice_set_key
    smc_id = TZ_ES_SET_ICE_KEY_ID;

    desc.arginfo = TZ_ES_SET_ICE_KEY_PARAM_ID;
    desc.args[0] = index;
    desc.args[1] = virt_to_phys(tzbuf_key);
    desc.args[2] = tzbuflen_key;
    desc.args[3] = virt_to_phys(tzbuf_salt);
    desc.args[4] = tzbuflen_salt;

    scm_call2_noretry(smc_id, &desc);
        __scm_call2(fn_id, desc, false);
```

Build metadata.img (1)

Target : metadataimage

```
ifneq ($(strip $(BOARD_METADATAIMAGE_PARTITION_SIZE)),)

TARGET_OUT_METADATA := $(PRODUCT_OUT)/metadata

INSTALLED_METADATAIMAGE_TARGET := $(PRODUCT_OUT)/metadata.img

define build-metadataimage-target
    $(call pretty,"Target metadata fs image: $(INSTALLED_METADATAIMAGE_TARGET)")
    @mkdir -p $(TARGET_OUT_METADATA)
    $(hide) $(MKEXTUSERIMG) -s $(TARGET_OUT_METADATA) $@ ext4 metadata $(BOARD_METADATAIMAGE_PARTITION_SIZE)
    $(hide) chmod a+r $@
endef

$(INSTALLED_METADATAIMAGE_TARGET): $(MKEXTUSERIMG) $(MAKE_EXT4FS)
    $(build-metadataimage-target)

ALL_DEFAULT_INSTALLED_MODULES += $(INSTALLED_METADATAIMAGE_TARGET)
ALL_MODULES.$(LOCAL_MODULE).INSTALLED += $(INSTALLED_METADATAIMAGE_TARGET)

.PHONY: metadataimage
metadataimage: $(INSTALLED_METADATAIMAGE_TARGET)

endif
```

Build metadata.img (2)

[Build Log]

```
[ 44% 71795/159780] /bin/bash -c "(mkdir -p out/target/product/msmnile/metadata ) && (out/host/linux-
x86/bin/mkuserimg_mke2fs.sh -s out/target/product/msmnile/metadata out/target/product/msmnile/metadata.img
ext4 metadata 16777216 ) && (chmod a+r out/target/product/msmnile/metadata.img )"
MKE2FS_CONFIG=./system/extras/ext4_utils/mke2fs.conf mke2fs -E android_sparse -t ext4 -b 4096
out/target/product/msmnile/metadata.img 4096
mke2fs 1.43.3 (04-Sep-2016)
Creating filesystem with 4096 4k blocks and 4096 inodes

Allocating group tables: 0/1   done
Writing inode tables: 0/1   done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: 0/1   done

e2fsdroid -f out/target/product/msmnile/metadata -a /metadata out/target/product/msmnile/metadata.img
Created filesystem with 11/4096 inodes and 1162/4096 blocks
```

ME – e4crypt_mount_metadata_encrypted

```
e4crypt_mount_metadata_encrypted

    LOG(DEBUG) << "e4crypt_mount_metadata_encrypted: " << mount_point << " " << needs_encrypt;

    read_key(data_rec, needs_encrypt, &key)
        fs_mkdirs(dir.c_str(), 0700)                                        // "/metadata/vold/metadata_encryption/key"
        android::vold::retrieveKey(create_if_absent, dir, temp, key)
            // 1st boot
            randomKey(key)
            storeKeyAtomically(key_path, tmp_path, kEmptyAuthentication, *key)          // "/metadata/vold/metadata_encryption/tmp"
                storeKey(tmp_path, auth, key)
                    generateKeymasterKey(keymaster, auth, appId, &kmKey)
                    writeStringToFile(kmKey, dir + "/" + kFn_keymaster_key_blob)
                    encryptWithKeymasterKey(keymaster, dir, keyParams, authToken, key, &encryptedKey)
                    writeStringToFile(encryptedKey, dir + "/" + kFn_encrypted_key)
                rename(tmp_path.c_str(), key_path.c_str()
            // Normal Boot
            retrieveKey(key_path, kEmptyAuthentication, key)
                readFileToString(dir + "/" + kFn_encrypted_key, &encryptedMessage)
                decryptWithKeymasterKey(keymaster, dir, keyParams, authToken, encryptedMessage, key)

    // Create "dm-X" block device, and the target_tyep is "default-key"
    create_crypto_blk_dev(kDmNameUserdata, nr_sec, DEFAULT_KEY_TARGET_TYPE, default_key_params(data_rec->blk_device, key), &crypto_blkdev)

    // 1st boot
    LOG(INFO) << "Beginning inplace encryption, nr_sec: " << nr_sec;
    cryptfs_enable_inplace(const_cast<char*>(crypto_blkdev.c_str()), data_rec->blk_device, nr_sec, &size_already_done, nr_sec, 0, false);
    LOG(INFO) << "Inplace encryption complete";

    // Decrypted and mapped, now to mount userdata partition
    LOG(DEBUG) << "Mounting metadata-encrypted filesystem:" << mount_point;
    mount_via_fs_mgr(data_rec->mount_point, crypto_blkdev.c_str());
```

ME – default-key map

- kernel/msm-4.14/drivers/md/dm-default-key.c

```
static struct target_type default_key_target = {
                .name   = "default-key",
                .version = {1, 0, 0},
                .module = THIS_MODULE,
                .ctr    = default_key_ctr,
                .dtr    = default_key_dtr,
                .map    = default_key_map,                    // do map operation for each bio to this dm-X block device.
                .status = default_key_status,
                .prepare_ioctl = default_key_prepare_ioctl,
                .iterate_devices = default_key_iterate_devices,
};


static int default_key_map(struct dm_target *ti, struct bio *bio)
{
                const struct default_key_c *dkc = ti->private;

                bio_set_dev(bio, dkc->dev->bdev);
                if (bio_sectors(bio)) {
                                bio->bi_iter.bi_sector = dkc->start +
                                                dm_target_offset(ti, bio->bi_iter.bi_sector);
                }

                if (!bio->bi_crypt_key && !bio->bi_crypt_skip)
                                bio->bi_crypt_key = &dkc->key; // set metadata encryption key to bio

                return DM_MAPIO_REMAPPED;
}
```

ME – Get key from bio/inode

- kernel/msm-4.14/security/pfe/pfk.c

```
pfk_load_key_start
                pfk_get_key_for_bio(bio, &key_info, &algo_mode, is_pfe, &data_unit);


static int pfk_get_key_for_bio(const struct bio *bio,
                               struct pfk_key_info *key_info,
                               enum ice_cryto_algo_mode *algo_mode,
                               bool *is_pfe, unsigned int *data_unit)
{
<snip>
                which_pfe = pfk_get_pfe_type(inode);    // EXT4_CRYPT_PFE, F2FS_CRYPT_PFE, INVALID_PFE
<snip>
                if (which_pfe != INVALID_PFE) {    // EXT4_CRYPT_PFE or F2FS_CRYPT_PFE
                               /* Encrypted file; override ->bi_crypt_key */
                               pr_debug("parsing inode %lu with PFE type %d\n",
                                              inode->i_ino, which_pfe);
                               return (*(pfk_parse_inode_ftable[which_pfe]))
                                                              (bio, inode, key_info, algo_mode, is_pfe); // For encrypted file by FBE, get key
and return
                }

                /*
                 * bio is not for an encrypted file.  Use ->bi_crypt_key if it was set.
                 * Otherwise, don't encrypt/decrypt the bio.
                 */
#ifdef CONFIG_DM_DEFAULT_KEY
                key = bio->bi_crypt_key;           // For Filesystem metadata, use metadata key in bio as its key.
#endif
<snip>
                key_info->key = &key->raw[0];
                key_info->key_size = PFK_SUPPORTED_KEY_SIZE;
                key_info->salt = &key->raw[PFK_SUPPORTED_KEY_SIZE];
                key_info->salt_size = PFK_SUPPORTED_SALT_SIZE;
                if (algo_mode)
                               *algo_mode = ICE_CRYPTO_ALGO_MODE_AES_XTS;
                return 0;
}
```

Wrapped Key - Generate

```
fs_mgr_mount_all
    call_vdc({"cryptfs", "encryptFstab", fstab->recs[attempted_idx].mount_point})
        VoldNativeService::encryptFstab()
            e4crypt_mount_metadata_encrypted(mountPoint, true)
                read_key(data_rec, needs_encrypt, &key)
                    android::vold::retrieveKey(create_if_absent, dir, temp, key)
                        if (is_metadata_wrapped_key_supported())
                            generateWrappedKey(MAX_USER_ID, KeyType::ME, key)
                        else
                            randomKey(key)
                        storeKeyAtomically(key_path, tmp_path, kEmptyAuthentication, *key)
                        if (is_metadata_wrapped_key_supported())
                            getEphemeralWrappedKey(KeyFormat::RAW, *key, &ephemeral_wrapped_key)
                create_crypto_blk_dev(kDmNameUserdata, nr_sec, DEFAULT_KEY_TARGET_TYPE, default_key_params(data_rec->blk_device, key),
&crypto_blkdev)
                cryptfs_enable_inplace(const_cast<char*>(crypto_blkdev.c_str()), data_rec->blk_device, nr_sec, &size_already_done, nr_sec, 0, false)
                mount_via_fs_mgr(data_rec->mount_point, crypto_blkdev.c_str());
```

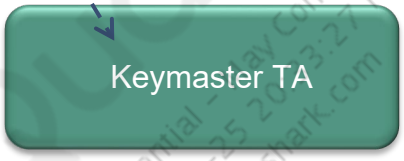
```
generateWrappedKey
    keymaster.generateKey(paramBuilder, &key_temp)
        mDevice->generateKey(inParams.hidl_data(), hidlCb)
            kmHal_->generate_key(&kmParams, &key_blob, &key_characteristics)
                KeymasterHalDevice::generate_key
                    generate_key_common(params, key_blob)
                        req->cmd_id = KEYMASTER_GENERATE_KEY;
                        utils->send_cmd(req, buffer->get_offset(), resp, resp_size)
```

HLOS

Wrapped Key – Export

```
getEphemeralWrappedKey
    keymaster.exportKey(format, kmKey, "!", "!", &key_temp)
        mDevice->exportKey(format, kmKeyBlob, kmClientId, kmAppData, hidlCb);
            kmHal_->export_key( ... );
                KeymasterHalDevice::export_key
                    req->cmd_id = KEYMASTER_EXPORT_KEY;
                    utils->send_cmd(req, buffer->get_offset(), resp, resp_size);
```

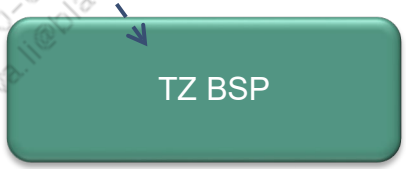
HLOS



Wrapped Key – Set Key

```
pfk_load_key_start
    pfk_get_key_for_bio(bio, &key_info, &algo_mode, is_pfe, &data_unit);
        which_pfe = pfk_get_pfe_type(inode);
        if (which_pfe != INVALID_PFE)
            return (*(pfk_parse_inode_ftable[which_pfe]))(bio, inode, key_info, algo_mode, is_pfe);
        // bio is not for an encrypted file.  Use ->bi_crypt_key if it was set.
        key = bio->bi_crypt_key;
        key_info->key = &key->raw[0];
        key_info->salt = &key->raw[PFK_SUPPORTED_KEY_SIZE];
        pfk_kc_load_key_start(key_info.key, key_info.key_size, key_info.salt, key_info.salt_size,
&key_index, async, data_unit);
            kc_update_entry(entry, key, key_size, salt, salt_size, data_unit);
                qti_pfk_ice_set_key(entry->key_index, entry->key, entry->salt, s_type, data_unit);
                    set_key(index, key, salt, data_unit);
                        smc_id = TZ_ES_CONFIG_SET_ICE_KEY_ID;
                        scm_call2_noretry(smc_id, &desc);
```

HLOS



Section 6

Reference

Reference

Document	
Qualcomm Technologies	
Qualcomm Android Security Features	80-NU861-1
Resources (Google)	
https://source.android.com/security/encryption/full-disk	Full-Disk Encryption
https://source.android.com/security/encryption/file-based	File-Based Encryption
https://source.android.com/security/encryption/metadata	Metadata Encryption

Thank you

For additional information or to submit technical questions, go to:
https://createpoint.qti.qualcomm.com