

1 Preface

Devices often contain highly sensitive information which is consistently at risk to get physically lost or stolen. Setting user passwords does not guarantee data protection against unauthorized access. The attackers can simply bypass the software system of a device and access the data storage directly. Only the use of encryption can guarantee data confidentiality in the case where storage media is directly accessed.

This document provides steps to run a transparent storage encryption at block level using DM-Crypt taking advantage of the secure key feature provided by i.MXs Cryptographic Accelerator and Assurance Module (CAAM). The document applies to all i.MX SoCs having CAAM module. The feature is not available on i.MX SoCs with DCP.

1.1 Intended audience and scope

This document is intended for software, hardware, and system engineers who are planning to use CAAM to perform storage encryption.

1.2 References

- [i.MX 8MM Security Reference Manual](#)
- [Linux mainline kernel archive, 2018.](#)
- [DM-Crypt: Linux device-mapper crypto target, 2018](#)
- [LUKS: Linux Unified Key Setup, 2018](#)
- [dmsetup\(8\) - Linux man page](#)
- [keyctl\(1\) - Linux man page](#)
- [keyrings - in-kernel key management and retention facilit](#)

2 Overview

2.1 DM-Crypt

DM-Crypt is a Linux kernel module which provides disk encryption. The module takes advantage of the Linux kernel's device-mapper infrastructure and offers a wide range of use cases. It can run on a variety of storage block devices, even if these devices use RAID and LVM. The device mapper is designed to provide a general and flexible way to add virtual layers over the actual block device so that developers can implement mirroring, snapshot, cascade, and encryption.

DM-Crypt is implemented as a device mapper target and resides entirely in kernel space between Linux subsystem and actual physical block device driver. This block device driver intercepts the request to read and write with the bio parameter that is sent to the actual physical device driver, encrypts the data by using the crypto API provided by the kernel, and then writes the data back to the actual block device. [Figure 1](#) represents the overall architecture of DM-Crypt.

Contents

1 Preface	1
1.1 Intended audience and scope.....	1
1.2 References.....	1
2 Overview	1
2.1 DM-Crypt.....	1
2.2 DM-Crypt accelerated by CAAM....	2
2.3 DM-Crypt using CAAM's Secure Key.....	3
3 Hands-On	4
3.1 Installation.....	4
3.2 Usage.....	5
3.3 Performance.....	8
4 Revision History	9
5 Appendix A. Configuration	9
5.1 Kernel configuration.....	9



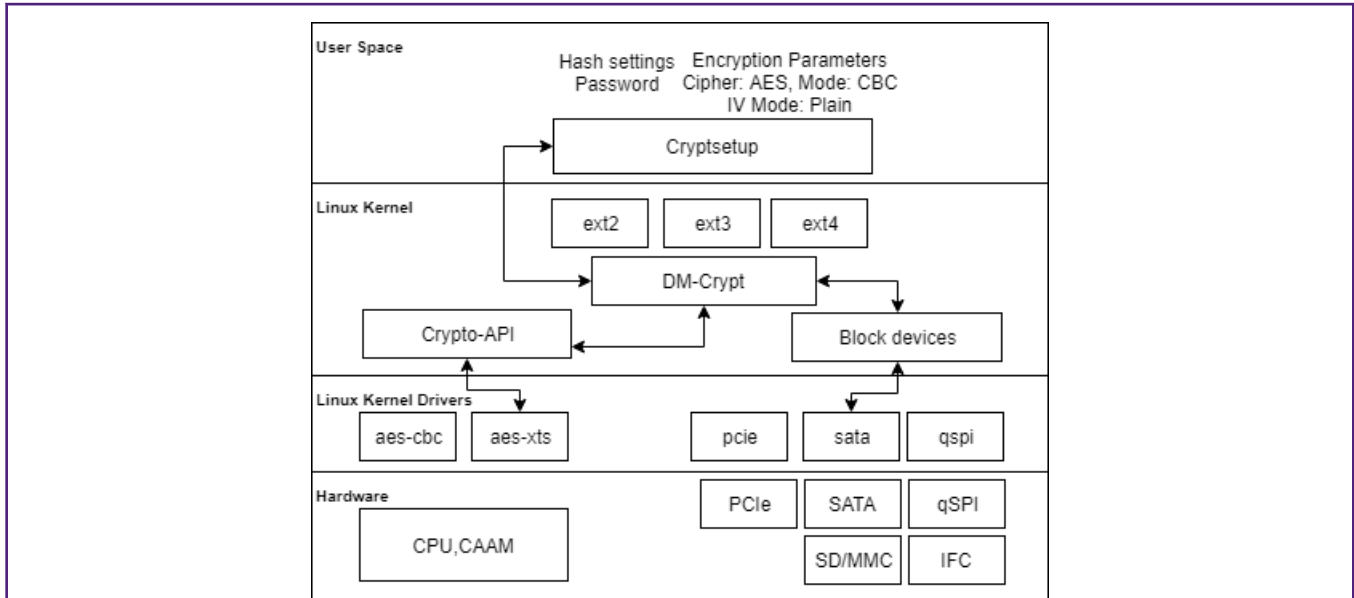


Figure 1. DM-Crypt's overall architecture

DM-Crypt supports the dynamic encryption function. Dynamic encryption, also called real-time encryption, or transparent encryption, automatically encrypts or decrypts the data in the process of using without users' intervention. The legitimate users can use the encrypted files without explicitly performing decryption.

2.2 DM-Crypt accelerated by CAAM

DM-Crypt performs cryptographic operations via the interfaces provided by the Linux kernel crypto API. The kernel crypto API defines a standard, extensible interface to ciphers and other data transformations implemented in the kernel (or as loadable modules). DM-Crypt parses the cipher specification (aes-cbc-essiv:sha256) or as kernel crypto API syntax (capi:xts(aes)-plain64) passed as part of its mapping table and instantiates the corresponding transforms via the kernel crypto API.

To list the available cryptographic transformations on the target:

```
# cat /proc/crypto
name : cbc(aes)
>driver : cbc-aes-caam
module : kernel
priority : 3000
refcnt : 1
selftest : passed
internal : no
type : givcipher
async : yes
blocksize : 16
min keysize: 16
max keysize: 32
ivsize : 16
geniv : <built-in>
```

If there is more than one implementation for a specific transformation, DM-Crypt selects the implementation with higher priority.

i.MX SoCs provide modular and scalable hardware encryption through NXP's CAAM. To have DM-Crypt accelerated by CAAM, the driver should register a transformation implementation with a higher priority.

In the output above, there are 2 implementations of cbc(aes) transformation. The CAAM-backed implementation has a priority of 3000 while the software implementation has a value of 300. This means that if you instruct DM-Crypt to use cbc(aes) cipher, the CAAM-backed implementation is used.

To list all transformations registered by CAAM driver:

```
# grep -B1 -A2 caam /proc/crypto|grep -v kernel
name      : rsa
driver    : rsa-caam
priority  : 3000
--
name      : xcbc(aes)
driver    : xcbc-aes-caam
priority  : 3000
```

NOTE

AES in XTS mode is not supported on i.MX devices.

2.3 DM-Crypt using CAAM's Secure Key

In Linux Unified Key Setup (LUKS) mode, to generate the disk encryption key (master key), the user supplies a passphrase which is combined with a salt, then a hash function is applied for a supplied number of rounds. When the user wants to mount an encrypted volume, the passphrase should be supplied. An alternative could be providing a key file stored in an external drive containing necessary decryption information. Those approaches are not convenient with embedded devices usage. The aim of using DM-Crypt with CAAM's secure key is to suppress the mechanism of encrypting the master volume key with a key derived from a user-supplied passphrase. DM-Crypt can take advantages of secure key also to protect storage volumes from offline decryption. The volume encryption key is only seen encrypted on device DDR. In addition, the volume could only be opened by the devices that have the same OTPMK burned in the fuses.

The secure key feature is based on CAAM's black key mechanism. Black key protects user keys against bus snooping while the keys are being written to or read from memory external to the SOC. called a 'Black Key'.

CAAM supports two different black key encapsulation schemes, which are AES-ECB and AES-CBC. Regarding AES-ECB encryption, the data is a multiple of 16 bytes long. If the key is a multiple of 128-bit, which will be if AES-256 or AES-128 encryption is used, then the AES-ECB encrypted data encryption key would fit in the same buffer as the original unencrypted data encryption key. If the data encryption key is not a multiple of 16 bytes long, it is padded before being encrypted. A CCM-encrypted black key is always at least 12 bytes longer than the encapsulated key (nonce value + MAC tag), where the "MAC tag" (integrity check value) ensures the integrity of the encapsulated key. While AES-ECB encryption is intended for quick decryption, AES-CCM encryption is used for high assurance. CCM-encrypted keys are preferred, unless there is a need to fit the encrypted key in the same space as the unencrypted key.

Black keys are session keys, therefore, they are not power-cycles safe. CAAM's blob mechanism provides a method for protecting user-defined data across system power cycles. It provides both confidentiality and integrity protection. The data to be protected is encrypted so that it can be safely placed into non-volatile storage before the SoC is powered down.

The following diagram illustrates the changes that have been added to support full disk encryption using secure key. The CAAM driver registers new Cryptographic transformations in the kernel to use ECB and CBC blacken keys, *tk(ecb(aes))* and *tk(cbc(aes))* respectively. The *tk* prefix refers to Tagged Key. A Tagged Key is a key that contains metadata indicating which key it is, black key or red key, and how to handle it, that is encapsulate in secure memory or in general memory. Linux provides an in-kernel key management and retention facility called Keyrings. Keyring also enables interfaces to allow accessing keys and performing operations such as add, update, and delete from user-space.

The kernel provides several basic types of keys including keyring, user, and logon. The CAAM driver introduces a new key type named *caam_tk* relative to CAAM Tagged Key. Basically, CAAM Tagged Key provisioning can be done through Linux Key Retention service and managed by user-space application such as *keyctl*. Secure key can be placed in Linux Key Retention service while secure key blobs can be stored on any non-volatile storage.

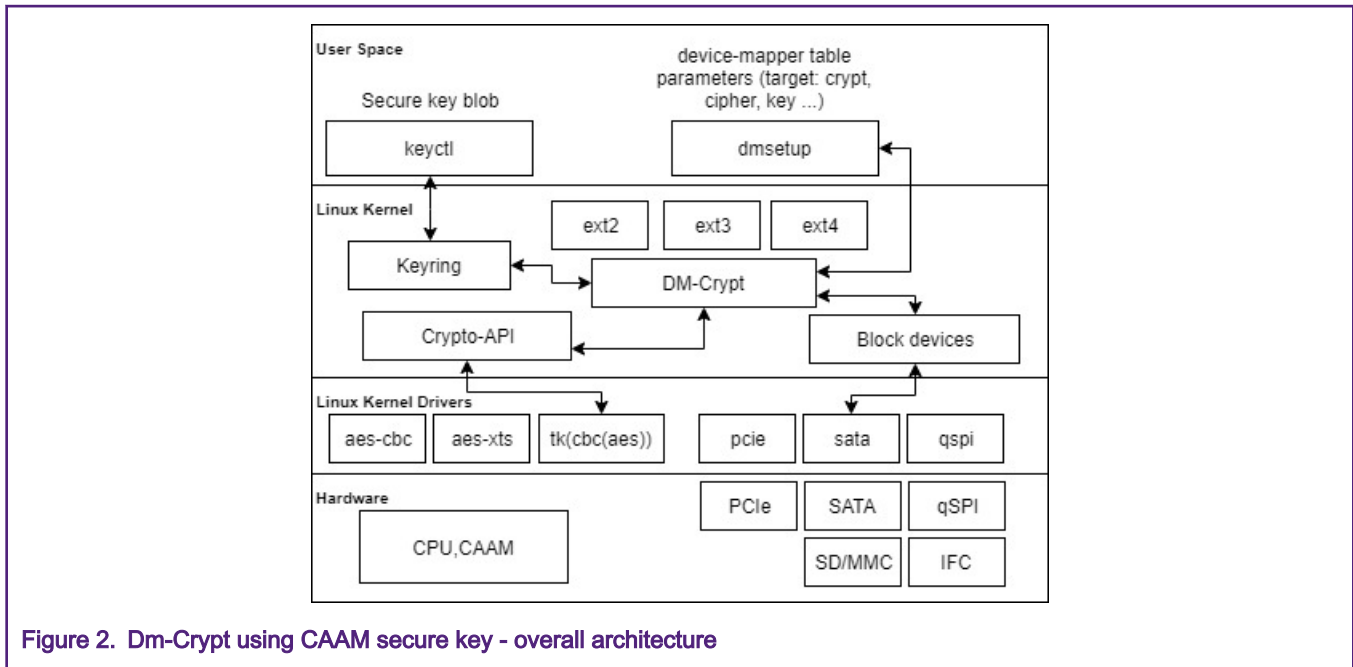


Figure 2. Dm-Crypt using CAAM secure key - overall architecture

Dmsetup (part of the libdevmapper package) is a powerful tool for performing very low-level configuration and is used to manage encrypted volumes instead of cryptsetup. Cryptsetup supports different encryption operating modes including plain mode and LUKS mode which are the most common modes. In LUKS mode, the master-key is encrypted. While in plain mode, there is no key encryption, yet passphrase hashing is not needed since secure key provides the same level of security. Hence, managing encrypted devices using cryptsetup is not applicable and dmsetup is used instead.

3 Hands-On

The steps described in this section have been executed on a x86_64 machine running Ubuntu 16.04 for any host related actions (for example, building) and on an i.MX 8MM-EVK board as a target device. This applies also to all i.MX devices having CAAM module and running a block-based storage. NAND flash is an MTD device, hence, DM-Crypt cannot be used with it and the hands-on do not apply for it.

3.1 Installation

1. Install the essential packages on your host machine.

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo \ build-essential chrpath
libssl1.2-dev xterm curl
```

2. Install repo tool.

```
$ mkdir -p ~/bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo >\ ~/bin/repo
$ export PATH=$PATH:~/bin
$ chmod a+x ~/bin/repo
```

3. Download the BSP source.

```
$ mkdir imx-yocto-bsp
$ cd imx-yocto-bsp
$ repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-linux-sumo -m
imx-4.14.78-1.0.0_ga.xml
$ repo sync
```

4. Create a local branch for the demo.

```
$ repo start imx-fde-demo --all
```

5. Download meta-imx-fde-demo layer source.

```
$ git clone https://source.codeaurora.org/external/imxsupport/imx_sec_apps
$ cp -r imx_sec_apps/meta-imx-fde-demo/ sources/
```

6. Select your board; in current example, it is an i.MX8MM EVK board.

```
$ DISTRO=fsl-imx-xwayland MACHINE=imx8mmevk source fsl-setup-release.sh -b build
```

7. Add meta-imx-fde-demo layer.

```
$ bitbake-layers add-layer ../sources/meta-imx-fde-demo
```

NOTE

This meta-layer enables all necessary kernel options to run the demo. For minimalist kernel configuration, see Appendix A.

8. Add required tools to build by editing **conf/local.conf** file and appending.

```
CORE_IMAGE_EXTRA_INSTALL+="coreutils keyutils lvm2 e2fsprogs-mke2fs util-linux"
```

GNU's dd is required instead of Busybox's dd later for benchmarking. Keyutils provides keyctl, which is required to manage Linux Key retention service.

Lvm2 provides dmsetup utility and libraries to manage device-mapper.

e2fsprogs-mke2fs contains necessary tools to create filesystems.

util-linux provides blockdev utility needed to read number of sectors from a volume.

9. Build.

```
$ bitbake core-image-base
```

10. Flash the SD Card image.

```
$ cd tmp/deploy/images/imx8mmevk
$ bunzip2 -dk -f core-image-base-imx8mmevk.sdcard.bz2
$ sudo dd if=core-image-base-imx8mmevk.sdcard of=/dev/sd<partition> bs=1M conv=fsync
```

3.2 Usage

1. After booting the device, make sure that cryptographic transformations using Tagged Key are registered.

```
# grep -B1 -A2 tk- /proc/crypto|grep -v kernel
name : tk(ecb(aes))
driver : tk-ecb-aes-caam
priority : 1
--
name : tk(cbc(aes))
driver : tk-cbc-aes-caam
priority : 1
```

2. Make sure Dm-Crypt is enabled.

```
# dmsetup targets
crypt          v1.18.1
striped       v1.6.0
linear        v1.4.0
error         v1.5.0
```

crypt should be displayed among the targets. You should also make sure that you have the **/dev/mapper** directory and the **/dev/mapper/control** device node.

3. Then, you need to provide the device with its key, the secure key, which could be created either from a defined plain key or randomly.

To create a secure key from a defined plain text key:

```
# keyctl add caam_tk seckey "set ecb 0123456789abcdef" @s | xargs keyctl print > blob
```

To create secure key from random plain key:

```
# keyctl add caam_tk seckey "new ecb 16" @s | xargs keyctl print > blob
```

The result is a Tagged Key file written to filesystem. The used key encryption scheme is ECB.

The format of the key is the following:

```
# cat blob
:hex:85c121ef4834be7621c86fa6d3db638cee5dc4efaad8a0240f64bf8ada684f0a2a82a6531411d951c4238dcd99ba
51dce5b603676b873ece4f6b9904cfd18f4
```

4. Create a secure volume. It could be a physical partition, in this example, make use of an image file and mount it later.

```
# dd if=/dev/zero of=encrypted.img bs=1M count=32
# losetup /dev/loop0 encrypted.img
```

5. Using dmsetup, create a new device-mapper device named *encrypted*, for example, and specify mapping table. The table can be provided on stdin or as argument.

```
# dmsetup -v create encrypted --table "0
$(blockdev --getsz /dev/loop0) crypt capi:tk(cbc(aes))-plain :32:caam_tk:seckey 0 /dev/loop0
0 1 sector_size:512"
```

Following is a breakdown of the mapping table:

0	\$(blockdev --getsz /dev/loop0)	crypt	capi:tk(cbc(aes))-plain	:32:caam_tk:seckey	0	/dev/loop0	0	1	sector_size:512
+	+	+	+	+	+	+	+	+	+
+	+	+	+	+	+	+	+	+	+
start	size	target	cipher	IV	key	key	IV	device offset	optional parameters
					type	name	offset		

- **start** means encrypting begins with sector 0.
- **size** is the size of the volume in sectors.
- **blockdev** gets the number of sectors of the device.
- **target** is crypt.
- **cipher** is set in Kernel crypto API format to use Tagged Key.

As mentioned in the previous section, Tagged Key indicates if the key is a black key or red key. The red key is an un-encrypted key compared to the black key, which is encrypted using either AES-ECB or AES-CBC. In contrast of black keys, red keys are not secure keys. The red key can still be encapsulated to blobs and decapsulated to red key only by CAAM. The encapsulation and decapsulation is performed through the Linux Key Retention service.

cipher set to `capi:tk(cbc(aes))-plain` and key set to `:32:caam_tk:seckey` leads to use of the Tagged Key. In this case, the Tagged Key is a black key (secure key).

Setting cipher to `capi:cbc(aes)-plain` and key to `:32:caam_tk:seckey` makes use of transformation implementation which does not support Tagged Key. In this case, the key is a red key (plain text key), in which the data is not processed using the key provided by the user.

- **IV** is the Initialization Vector defined to plain, initial vector, which is the 32-bit little-endian version of the sector number, padded with zeros if necessary.
- **Key type** is the Keyring key service type, set to Tagged Key. 32 is the key size in bytes.
- **Key name** is the key description to identify the key to load.
- **IV offset** is the value to add to sector number to compute the IV value.
- **Device** is the path to device to be used as backend; it contains the encrypted data.
- **Offset** represents encrypted data begins at sector 0 of the device.
- **Optional parameters** represent the number of optional parameters.
- **sector_size** specifies the encryption sector size.

For more detailed options and descriptions, refer to <https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt>.

The created device appears in `/dev/mapper`.

```
# dmsetup table --showkey encrypted
0 65536 crypt capi:tk(cbc(aes))-plain :32:caam_tk:blackkey 0 7:0 0
```

6. Create a file system on the device.

```
# mkfs.ext4 /dev/mapper/encrypted
```

7. Setup a mount point.

```
# mkdir /mnt/encrypted
```

8. Mount the mapped device.

```
# mount -t ext4 /dev/mapper/encrypted /mnt/encrypted/
```

At this level, everything data you write to `/mnt/encrypted` is encrypted on the real block device `/dev/loop0`.

9. Write to device.

```
# echo "This is a test of full disk encryption on i.MX" > /mnt/encrypted/readme.txt
```

10. Unmount the device.

```
# umount /mnt/encrypted/
```

11. Deactivate the device mapper device.

```
# dmsetup remove encrypted
```

12. Restart the board.

13. Import the blob to key retention service.

```
# keyctl add caam_tk seckey "load ecb $(cat blob) " @s
```

14. Mount the encrypted device.

```
# losetup /dev/loop0 encrypted.img
```

15. Specify the mapping table to encrypt the volume using dmsetup.

```
# dmsetup -v create encrypted --table "0 $(blockdev --getsz /dev/loop0) crypt
  capi:tk(cbc(aes))-plain :32:caam_tk:seckey 0 /dev/loop0 0 1 sector_size:512"
```

16. Mount.

```
# mount /dev/mapper/encrypted /mnt/encrypted/
```

17. Read from device.

```
# cat /mnt/encrypted/readme.txt
```

This is a test of full disk encryption on i.MX.

3.3 Performance

The dd command can be used as a simple benchmarking utility by copying data to encrypted and partition and timing the operations.

1. Load the block ramdisk module, set the desired size in blocks to 64 M.

```
modprobe brd rd_size=32768
```

/dev/ram0 appears.

2. Create a file with size 32M with random content.

```
# dd if=/dev/urandom of=/tmp/random conv=fsync bs=1M count=32
```

3. Create mapping table with sector size equal to 512.

```
# dmsetup -v create secram --table "0 $(blockdev --getsz /dev/ram0) crypt capi:tk(cbc(aes))-
  plain :32:caam_tk:seckey 0 /dev/ram0 0 1 sector_size:512"
```

4. Remove the previous files to gain some space if there is no enough disk space.

```
# rm encrypted.img
```

5. Clear cache.

```
# echo 3 > /proc/sys/vm/drop_caches
```

6. Write to disk.

```
# time dd if=/tmp/random of=/dev/mapper/secram conv=fsync
65536+0 records in
65536+0 records out
33554432 bytes (34 MB, 32 MiB) copied, 3.64771 s, 9.2 MB/s
real    0m3.651s
```



```
user    0m0.033s
sys     0m0.514s
```

7. Redo the steps with sector size equal to 4096.

```
# dd if=/dev/urandom of=/tmp/random conv=fsync bs=1M count=32
# dmsetup remove secram
# dmsetup -v create secram --table "0 $(blockdev --getsz /dev/ram0) crypt
  capi:tk(cbc(aes))-plain :32:caam_tk:seckey 0 /dev/ram0 0 1 sector_size:4096"
# time dd if=/tmp/random of=/dev/mapper/secram conv=fsync
65536+0 records in
65536+0 records out
33554432 bytes (34 MB, 32 MiB) copied, 1.22975 s, 27.3 MB/s
real    0m1.264s
user    0m0.040s
sys     0m0.517s
```

By default, DM-Crypt sends encryption or decryption requests to the crypto layer one block at a time, making each request 512 bytes long, which is a much smaller size for hardware engine. This means the CAAM cannot deliver its best performance.

DM-Crypt can take block size parameter in the form, `sector_size:<bytes>`, where bytes means user-specified sector size used for encryption. The size can be in range of 512 - 4096 bytes and must be power of two. The virtual device specifies this size as minimal IO size and logical sector size.

AES in XTS is the most convenient cipher for block-oriented storage devices and shows significant performance gain, however, this mode is not natively supported on i.MX's CAAM module.

4 Revision History

Revision number	Date	Substantive changes
0	February 2020	Initial release

5 Appendix A. Configuration

5.1 Kernel configuration

Following is the minimal kernel parameters list, which should be enabled to include DM-Crypt with CAAM secure key:

```
# Enable DM-Crypt and its dependencies
CONFIG_BLK_DEV_DM=y
CONFIG_BLK_DEV_MD=y
CONFIG_MD=y
CONFIG_DM_CRYPT=y
# Enable CAAM black key/blob driver and its dependencies
CONFIG_CRYPTODEV_FSL_CAAM_DESC=y
CONFIG_CRYPTODEV_FSL_CAAM_UTIL_BLACK_KEY=y
CONFIG_CRYPTODEV_FSL_CAAM_UTIL_BLOB=y
CONFIG_CRYPTODEV_FSL_CAAM_TK_API=y
CONFIG_CAAM_KEYS=y
```

If the system needs redundancy support, DM-Crypt supports such capabilities by enabling DM-Multipath. For more details, see DM-Crypt documentation.”

For performance measurements, the following kernel options need to be enabled:

User-space interface for hash, AEAD, and symmetric key cipher algorithms.

```
CONFIG_CRYPTO_USER_API=y  
CONFIG_CRYPTO_USER_API_HASH=y  
CONFIG_CRYPTO_USER_API_AEAD=y  
CONFIG_CRYPTO_USER_API_SKCIPHER=y
```

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 25 February 2020

Document identifier: AN12714

