Overview of Linux's i.mx Data Co-Processor (DCP) driver

Purpose of this document is to give a rough overview of Linux's driver for the i.mx data co-processor (DCP). DCP can be found on various low end i.mx system on chips (SoC). DCP offers hardware backed acceleration of secure hash algorithm 1 (SHA1) and advanced encryption standard (AES) with 128 bits strength in electronic code book (ECB) and cipher block chaining (CBC) mode.

Basic mode of operation

Like most crypto offloaders, DCP has multiple data channels which can be filled with jobs. These jobs will be processed by the hardware asynchronously. The Linux driver utilizes two out of four channels: one to process SHA1 hashes and a second one for AES crypto. In theory Linux could use all four channels and let DCP decide when to process jobs from which channel. However, since the Linux crypto framework offers already a job scheduling framework it is done in software do avoid unexpected delays.

The block diagram in this document outlines how the driver is organized. It implements the hashing and symmetric key cipher interface of the Linux crypto framework. Two kernel threads are created by the driver, one to process hash jobs and one for AES. These threads dequeue jobs from the crypto framework, put them into the DCP core and wait for completion. Using this technique, users of the crypto API within Linux can either use the interface synchronous or asynchronous since blocking happens in the kernel threads.

Each DCP job is a command, it does not contain the data to work on. The actual data is transferred using direct memory access (DMA) from and to the DCP engine.

Use of AES keys

Ciphers like AES require a crypto key to work on. Usually the key is part of the DCP job and known to Linux in plaintext. We changed the driver to support a feature of DCP where an AES key can be referenced. DCP allows six different references. Four slots in the secure memory element can be used, the unique device identifier or a one time programmable key. Since the board support package (BSP) had already support for the blob mechanism as offered by the Cryptographic Accelerator and Assurance Module (CAAM) we decided to implement a similar feature in software using reference keys.

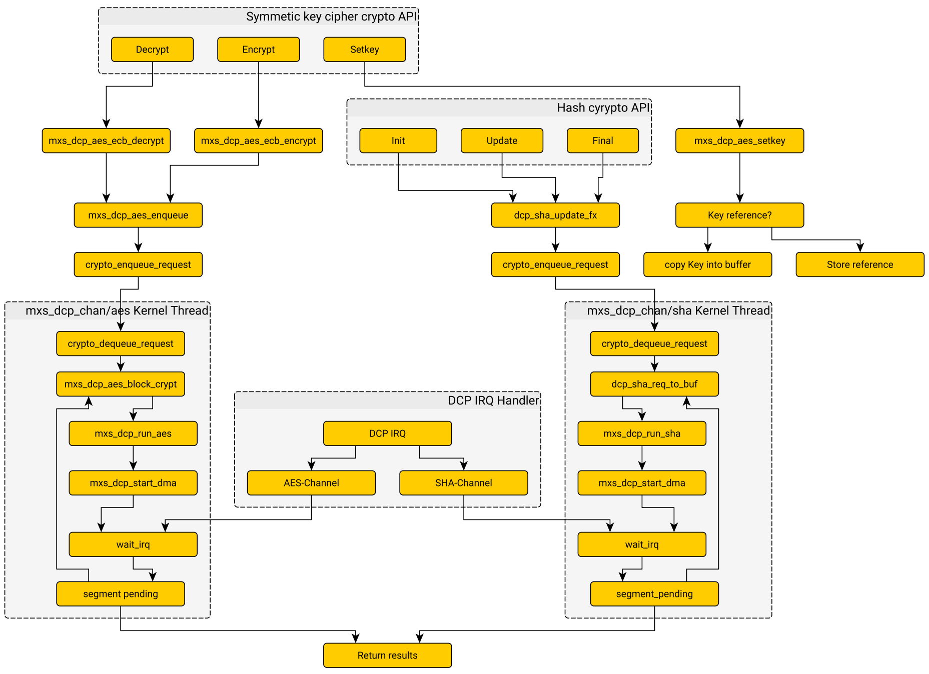

Figure 0.1: Block diagram of the driver

Implementing a CAAM-alike blob mechanism

A DCP blob has the following format:

- Format, *1* byte: Denotes the layout version of this blob. Currently always *0x01*.
- Blob key, *16* bytes: A random AES key used to encrypt the payload. The blob key itself is encrypted using the device one time programmable key.
- Nonce, *16* bytes: A random 16 byte value used for the payload encryption as initialization vector (IV).
- Payload length, *32* bytes: Length of the payload.
- Payload, variable length: The payload of the blob, encrypted in AES-128-GCM mode, usually the key that is covered.

That way an AES key used by Linux for disk encryption can be covered and placed on an untrusted medium such as an SPI-NOR flash. By using AES in GCM it is even possible to detect tampered blobs. Random bytes are fetched from Linux's cryptographic RNG.

Sumary

DCP offers SHA1 and AES in hardware. Since it is possible to use a secret one time programmable key it is possible to implement a way to cover keys which can be used by Linux disk encryption facilities.