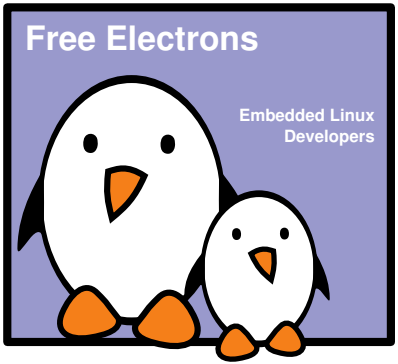
An overview of the crypto subsystem

Boris Brezillon
Free Electrons
boris@free-electrons.com



Boris Brezillon

- Embedded Linux engineer and trainer at Free Electrons
  - Embedded Linux **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux, Linux driver development, Yocto Project / OpenEmbedded and Buildroot **training courses**, with materials freely available under a Creative Commons license.
  - `http://free-electrons.com`
- Contributions
  - **Maintainer of the NAND subsystem**
  - **Kernel support for various ARM SoCs**
  - **Contributed Marvell's crypto engine driver**
- Living in **Toulouse**, south west of France

What is this talk about?

- Short introduction to some cryptographic concepts
- Overview of services provided by the crypto subsystem and how to use it
- Overview of the driver side of the crypto framework (how to implement a driver for a simple crypto engine)
- Random thoughts about the crypto framework

Introduction to generic cryptographic concepts

Cryptography? What is this?

- Do you know Alice and Bob? If you do, take a break and come back in five minutes
- Cryptography is about protecting communications between two or more participants
- Covers three main concepts:
  - **Confidentiality**: content of the communication cannot be spied on
  - **Data integrity**: content of the communication cannot be altered without participants noticing
  - **Authentication**: message origin can be checked
- Achieved by manipulating input/output messages and adding meta-data to them

Cryptography: Hash (or Digest)

- Used to guarantee **Data Integrity**
- Operates on a random number of input data
- Generates a 'unique' fixed-size signature for a specific input
- Examples: `SHA1`, `MD5`, ...
- Main criteria for a hash algorithm:
  - Keep the probability of collision as low as possible (ideally null)
  - Make it impossible to re-generate data from its hash
  - A small modification in the data should generate a completely different hash

Cryptography: Cipher

- Used to guarantee **Confidentiality**
- Transform the data so that someone external to the group can't read it
- Requires one or several **key(s)** to encrypt/decrypt data
- Ciphers can be **stream** or **block** oriented
  - **Stream Ciphers**: operate on a stream of data
  - **Block Ciphers**: operate on fixed-size blocks
- Ciphers can be **symmetric** or **asymmetric**
  - **Symmetric Ciphers**: the same key (called private key) is shared among all participants and is used to both encrypt and decrypt data
  - **Asymmetric Ciphers**: a pair of public/private key is used. The public key can be shared with anyone and is used to encrypt messages sent to the owner of the private key. The private key is then used to decrypt messages.
- Examples: `AES`, `RSA`, `...`

Cryptography: Block Cipher Mode

- Block ciphers can only be used to encrypt/decrypt a single block of data
- We need a solution to handle an arbitrary number of blocks
- Block cipher mode is just a protocol describing how to do that
- Most modes require an **Initialization Vector** (**IV**) to obfuscate the transmitted data
- Examples: `ECB (Electronic Codebook)`, `CBC (Cipher Chaining Block)`, `...`

Cryptography: MAC and HMAC

- MAC stands for Message Authentication Codes
- Mechanism used to authenticate the sender of a message
- Uses a key and a transformation algorithm to generate authentication data
- MAC can be based on Hash algorithms => HMAC

Cryptography: AEAD

- AEAD stands for Authenticated Encryption with Associated Data
- Combines everything in a single step (Authentication, Confidentiality and Data Integrity)
- Is usually based on existing Cipher, Hash/HMAC algorithms

Cryptography: Better introduction than mine

- Interested in more advanced description of these concepts?
- Watch Gilad Ben Yossef's talk: `https://youtu.be/dnGbhvweNb8`, `https://goo.gl/x5ikkv`

Linux Crypto Framework: Common Principles

Linux Crypto Framework: Basic Concepts

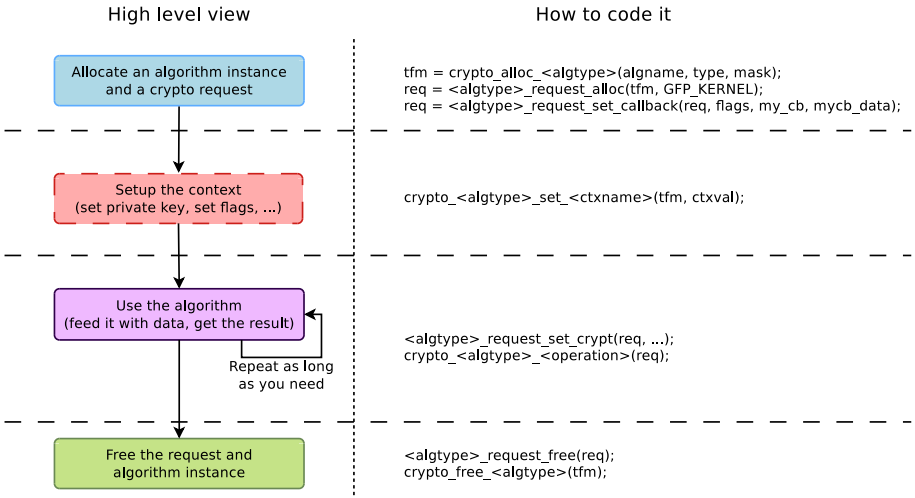
- Every crypto algorithm is about transforming input data into something else
  - **Transformation implementation**: represents an implementation of a specific algorithm (`struct crypto_alg`)
  - **Transformation object**: an instance of a specific algorithm (`struct crypto_tfm`)
- Everything inherits from `struct crypto_alg` and `struct crypto_tfm` either directly on indirectly

Linux Crypto Framework: Basic Concepts

- Supports a whole bunch of algorithms
- Here are some of them, spot the odd one
  - **Cipher**
  - **Hash**
  - **AEAD**
  - **HMAC**
  - **Compression**
- A transformation algorithm can be a template using basic building blocks
- Examples:
  - **hmac(sha1)**: HMAC using SHA1 hash
  - **cbc(aes)**: CBC using AES
  - **authenc(hmac(sha1),cbc(aes))**: AEAD using HMAC based on SHA1 and CBC based on AES

Linux Crypto Framework: How to use it

Using the crypto framework



Dummy user example

- Disclaimer: a lot of things have been deliberately omitted to keep the example simple
  - No error checking
  - Code is not separated in sub-functions
  - We only provide a simple example for a basic cipher: ecb(aes)
  - Headers inclusion has been omitted
  - There's no input/output parameter checking
  - ...
- To sum-up: do not use this example as a base for your developments, it's just here to show the different steps

Dummy user implementation

```
struct encrypt_ctx {
        struct crypto_skcipher *tfm;
        struct skcipher_request *req;
        struct completion complete;
        int err;
};

static void encrypt_cb(struct crypto_async_request *req,
                       int error)
{
        struct crypto_ctx *ctx = req->data;

        if (error == -EINPROGRESS)
             return;

        ctx->err = error;
        complete(&ctx->completion);
}

static void init(struct encrypt_ctx *ctx)
{
        /* Create a CBC(AES) algorithm instance: */
        ctx->tfm = crypto_alloc_skcipher("ecb(aes)", 0, 0);
        /* Create a request and assign it a callback: */
        ctx->req = skcipher_request_alloc(ctx->tfm,
                                          GFP_KERNEL);
        skcipher_request_set_callback(req,
                                      CRYPTO_TFM_REQ_MAY_BACKLOG,
                                      encrypt_cb, ctx);
        init_completion(&ctx.completion);
}
```

```
static void cleanup(struct encrypt_ctx *ctx)
{
        skcipher_request_free(ctx->req);
        crypto_free_skcipher(ctx->tfm);
}

int encrypt(void *key, void *data, unsigned int size)
{
        struct encrypt_ctx ctx;
        struct scatterlist sg;
        int ret;

        init(&ctx);

        /* Set the private key: */
        crypto_skcipher_setkey(ctx.tfm, key, 32);

        /*Now assign the src/dst buffer and encrypt data: */
        sg_init_one(&sg, data, size);
        skcipher_request_set_crypt(ctx.req, &sg, &sg, len,
                                   NULL);
        ret = crypto_skcipher_encrypt(ctx.req);
        if (ret == -EINPROGRESS || ret == -EBUSY) {
                wait_for_completion(&ctx.completion);
                ret = ctx.err;
        }

        cleanup(&ctx);

        return ctx.err;
}
```

In-kernel crypto users

- Disk encryption: **dm-crypt**
- Network protocols:
  - **IPSec**
  - **802.11**
  - **802.15.4**
  - **Bluetooth**
  - **...**
- File systems
- Device drivers
- `git grep "crypto_alloc_"` and you'll find a lot more

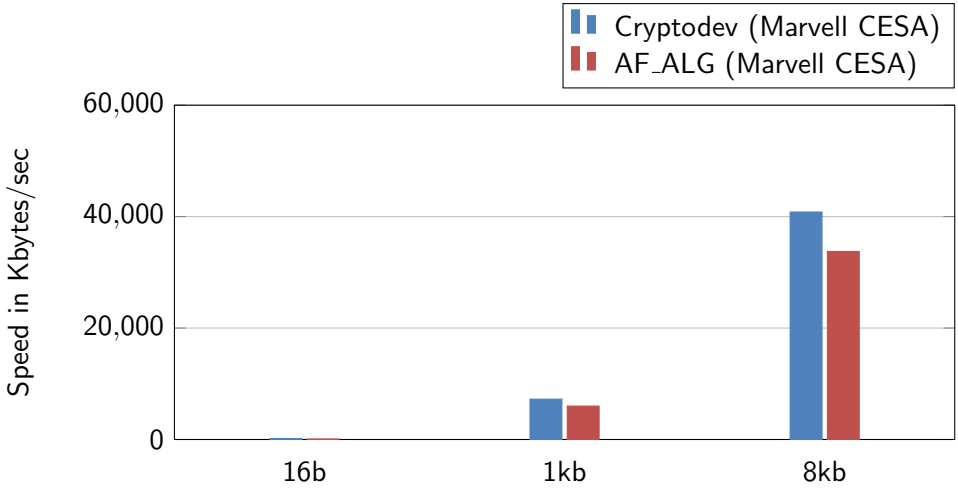
Using kernel crypto features from userspace

- People have pushed for this for quite some time
- Motivation for this:
  - Have a single base of code instead of duplicating it in userspace
  - Use hardware crypto engines that are only exposed to the kernel world
- Two competing solutions:
  - `cryptodev`: is an out-of-tree solution, ported from the BSD world
  - `AF_ALG`: the in-tree/official solution

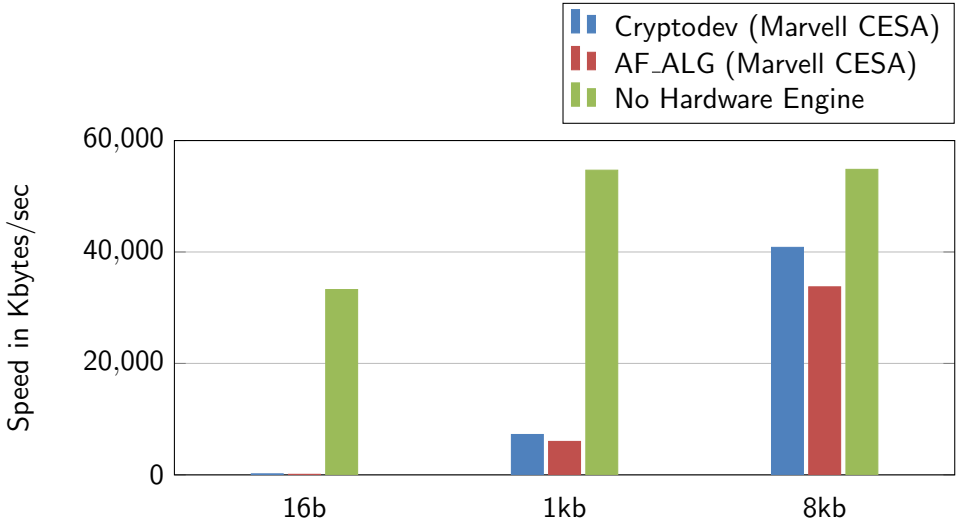
cryptodev vs AF_ALG

- cryptodev:
  - The first solution to have emerged
  - Is said to have more performance than AF_ALG
  - Still maintained as an out-of-tree kernel module
  - Interfaces with the in-kernel crypto framework
  - Exposes a device under /dev/crypto
  - Uses ioctls to setup the crypto context
  - Natively supported in OpenSSL
- AF_ALG:
  - Supported in mainline (appeared in Linux 2.6.38)
  - Manipulated through a netlink socket
  - Supported in OpenSSL as an out-of-tree module
- **Which one should I use???**

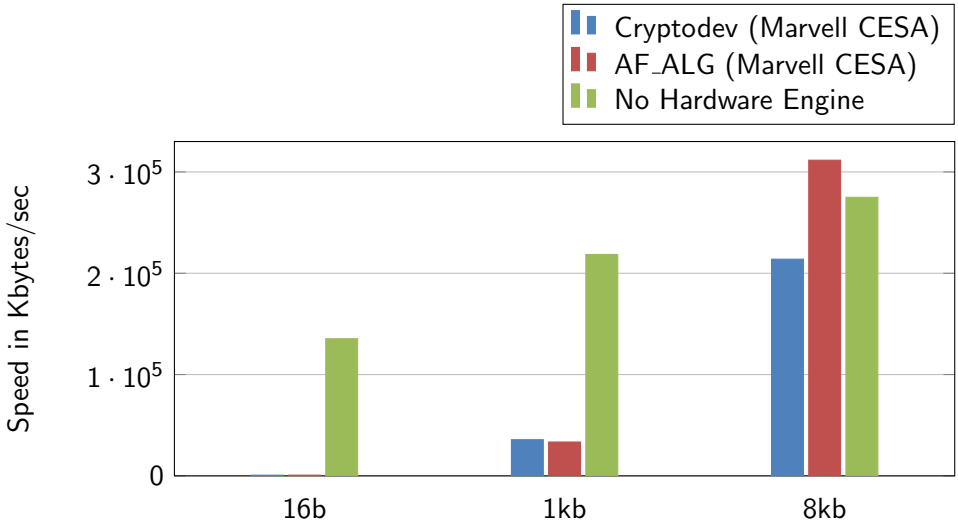
OpenSSL Speed Test: Sequential



OpenSSL Speed Test: Sequential



OpenSSL Speed Test: Parallelization (128 Threads)



cryptodev vs AF_ALG: Conclusion

- **Which one should I use???**
- cryptodev shows better results, but...
- ... both cryptodev and AF_ALG are outperformed by the software userspace solution
- And we don't gain that much in term of CPU load when offloading to the hardware engine (60% vs 100%)
- Of course those results are likely to be dependent on the crypto engine and its driver
- In any case
  - Think twice before using a hardware crypto engine from userspace
  - Do the test before taking a decision

Linux Crypto Framework: How to develop a crypto engine driver

How to develop a crypto engine driver

- The crypto framework does not distinguish hardware engines from software implementation
- Developing a crypto engine driver is just about registering a `crypto_alg` to the crypto subsystem
- Identify the type of algorithm you want to add support for and the associated `crypto_alg` interface (`skcipher_alg`, `ahash_alg`, `...`)
- Implement the `xxx_alg` interface and call `crypto_register_xxx()` to register it to the crypto framework
- We will study a simple algorithm type: `cbc(aes)`
- Go look at other drivers or the crypto framework doc if you need information on other alg type interfaces

Important crypto_alg Fields

```
struct skcipher_alg xxx_cbc_aes_alg = {
        ...
        .base = {
                /* Name used by the framework to find who is implementing what. */
                .cra_name = "cbc(aes)",

                /* Driver name. Can be used to request a specific implementation of an algorithm. */
                .cra_driver_name = "xxx-cbc-aes",

                /* Priority is used when implementation auto-selection takes place:
                 * if there are several implementers, the one with the highest priority is chosen.
                 * By convention: HW engine > ASM/arch-optimized > plain C
                 */
                .cra_priority = 300,

                /* CRYPTO_ALG_TYPE_XX: describes the type algorithm implemented here
                 * CRYPTO_ALG_ASYNC: the engine is operating in an asynchronous manner
                 * CRYPTO_ALG_KERN_DRIVER_ONLY: engine is not directly accessible to userspace
                 */
                .cra_flags = CRYPTO_ALG_TYPE_SKCIPHER | CRYPTO_ALG_KERN_DRIVER_ONLY | CRYPTO_ALG_ASYNC,

                /* Size of the data blocks this algo operates on. */
                .cra_blocksize = AES_BLOCK_SIZE,

                /* Size of the context attached to an algorithm instance. */
                .cra_ctxsize = sizeof(struct xxx_aes_ctx),

                /* constructor/destructor methods called every time an alg instance is created/destroyed. */
                .cra_init = xxx_skcipher_cra_init,
                .cra_exit = xxx_skcipher_cra_exit,
        },
};
```

skcipher_alg Fields

```
struct skcipher_alg mv_cesa_cbc_aes_alg = {
        /* Set key implementation. */
        .setkey = xxx_aes_setkey,

        /* Encrypt/decrypt implementation. */
        .encrypt = xxx_cbc_aes_encrypt,
        .decrypt = xxx_cbc_aes_decrypt,

        /* Symmetric key size. */
        .min_keysize = AES_MIN_KEY_SIZE,
        .max_keysize = AES_MAX_KEY_SIZE,
        /* IV size */
        .ivsize = AES_BLOCK_SIZE,
        .base = {
        ....
        },
};
```

```
static int xxx_encrypt(struct skcipher_request *req)
{
        struct my_ctx *ctx = crypto_tfm_ctx(req->base.tfm);

        /* Prepare and queue the request here. Return 0 if the request has
         * been executed, -EINPROGRESS if it's been queued, -EBUSY if it's
         * been backlogged or a different error code for other kind of
         * errors.
         */

        return ret;
}

static int xxx_decrypt(struct skcipher_request *req)
{
        struct my_ctx *ctx = crypto_tfm_ctx(req->base.tfm);

        /* Similar to xxx_encrypt() except this time we prepare and queue
         * a decrypt operation.
         */

        return ret;
}

static int xxx_setkey(struct crypto_skcipher *cipher, const u8 *key,
                      unsigned int len)
{
        struct my_ctx *ctx = crypto_tfm_ctx(req->base.tfm);

        /* Expand key and assing store the result in the ctx. */
        return ret;
}
```

Feedback on my experience with the crypto framework

The framework is a complex beast

- Good aspects:
  - Can easily be extended to support new algorithms (even non crypto related ones)
  - It comes with an extensive testsuite to detect bad implementation or regressions
- Bad aspects:
  - The crypto framework is so open that you sometime have several ways to expose the same thing (example: `crypto_alg+ablkcipher` or `skcipher`)
  - The object model is not consistent (how to inherit from a base interface is not enforced even in the core)
  - Hard to tell what the good practices are (old/existing drivers are usually not converted to the new way of doing things)
  - Important details can be discovered the hard way (example: completion callback should be called with `softirqs` disabled)
- Who am I to complain about these problems, the NAND framework is probably worse in this regard...
- Getting these things addressed requires a non-negligible effort and some help from drivers contributors

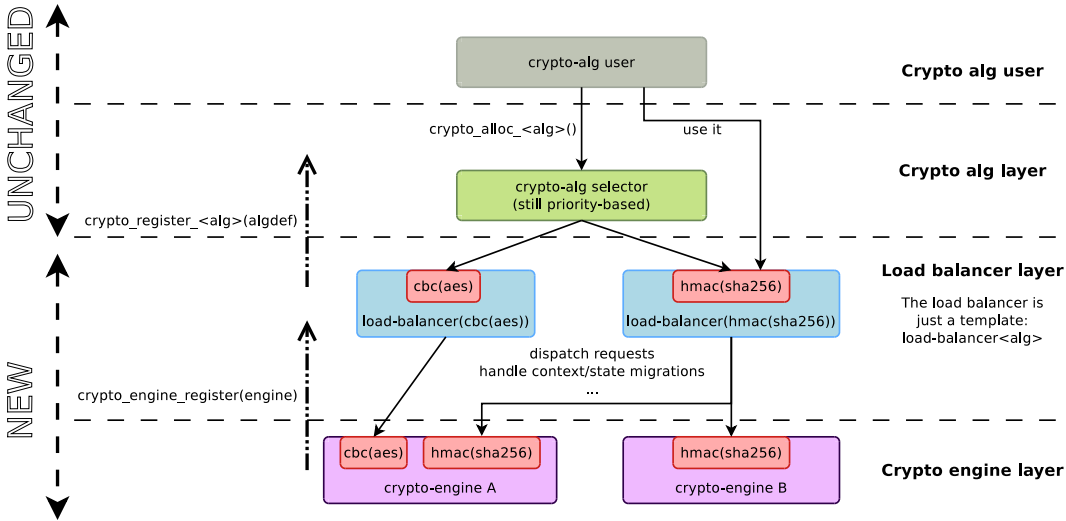
Limiting the number of interrupts: NAPI for crypto?

- Crypto engines may generate a lot of interrupts (potentially one per crypto request, maybe more if the engine has a limited FIFO size)
- Crypto can be used by the net stack which may decide to switch in polling mode under heavy net load
- The lack of NAPI-awareness at the crypto level defeats NAPI mode: you'll end up with a bunch of crypto interrupts which will prevent your system from running smoothly while under heavy net+crypto load
- CESA driver approach to limit the number of interrupts: try to queue requests at the DMA level and use a threaded IRQ in `IRQF_ONESHOT` mode to do some polling instead of re-activating interrupts right away
- Not a perfect solution: bigger latency than the `irq+softirq` approach
- **Question**: should we add a NAPI-like interface?

Load balancing: random thoughts

- Current priority-based algorithm selection shows its limits:
  - Some systems come with several instances of the same engine but only one of them can be used/exposed, unless the driver implements its own load balancing logic
  - We can't distribute the load over heterogeneous engines in the system: the engine with the highest priority gets all the requests
- Should we introduce a generic load-balancing mechanism?
- Here is a list of things to address if we do:
  - Introduce the concept of crypto engine, because some engines expose several crypto algs, but can't process things in parallel
  - Calculate a per-request load based on the request type, the request length and the engine it is queued to (or something simpler `load = length`?)
  - Keep track of the total load of each engine registered to the system in order to decide where the next request will go
  - Algorithm contexts/states are driver dependent: we need an intermediate/driver-agnostic representation to allow moving the context/state from one engine to another

Load balancing: random thoughts



Questions? Suggestions? Comments?

Boris Brezillon

boris.brezillon@free-electrons.com

http://free-electrons.com/pub/conferences/2017/elce/brezillon-crypto-framework/