Improving dm-crypt performance for XTS-AES mode through extended requests: first results

Levent Demir, Mathieu Thiery, Vincent Roca, Jean-Louis Roch, Jean-Michel Tenkes

Improving *dm-crypt* performance for XTS-AES mode through extended requests: first results

Levent Demir[1 2], Mathieu Thiery[1], Vincent Roca[2], Jean-Louis Roch[3], and Jean-Michel Tenkes[1]

[1] Incas ITSec, France, jm.tenkes@incas-itsec.com
[2] Inria, France, {levent.demir||vincent.roca@inria.fr}
[3] Grenoble Université, Grenoble INP, LIG, France, jean-louis.roch@imag.fr }

Abstract. Using dedicated hardware is common practice in order to accelerate cryptographic operations: complex operations are managed by a dedicated co-processor and RAM/crypto-engine data transfers are fully managed by DMA operations. The CPU is therefore free for other tasks, which is vital in embedded environments with limited CPU power. In this work we discuss and benchmark XTS-AES, using either software or mixed approaches, using Linux and *dm-crypt*, and a low-power Atmel(tm) board. This board featurs an AES crypto-engine that supports ECB-AES but not the XTS-AES mode. We show that the *dm-crypt* module used in Linux for full disk encryption has limitations that can be relaxed when considering larger block sizes. In particular we demonstrate that performance gains almost by a factor two are possible, which opens new opportunities for future use-cases.

Keywords: Full disk encryption, XTS-AES, encryption, decryption, atmel board, linux kernel crypto API, scatterlist, DMA.

1 Introduction

Data confidentiality has become an essential aspect in our society, and in particular Full Disk Encryption (FDE). Smartphones use FDE to protect users' data, and all modern operating systems can offer encrypted partitions, as an option during installation. Since the whole device is encrypted, nobody is able to distinguish encrypted data from random data.

The FDE of hard drives and USB sticks can be done using different tools. Linux, since kernel version 2.6, includes a *dm-crypt* module which is a device mapper for transparent encryption and decryption and is therefore a key component for FDE. In that context, the most recent and suitable cipher for block-oriented storage devices is AES in XTS mode, which is recommended by NIST [4]. However not every crypto-engine, even when it features a specific AES engine, natively supports the XTS-AES mode since this is a relatively recent as well as a complex mode. In our case we focussed on an embedded Atmel board, the SAMA5D3, featuring a cortex A5@536 MHz CPU, 256 MB of RAM, and a

crypto-engine that supports the ECB-AES mode (among other modes) but not XTS-AES mode.

Our first contribution consists in adding XTS-AES support through a mixed hard/soft implementation that leverages the ECB-AES crypto-engine.

Then benchmarks demonstrated that the performance achieved under these conditions were far from good and in particular did not match our own objective of on-the-fly encryption of large amounts of data within the Atmel board. We investigated and found that *dm-crypt* is limited to block sizes that are hard-coded to 512 bytes because it is also the common logical sector size on most devices. For historical reasons and backward compatibility this limit has never been changed. We therefore explored the possibility of having requests significantly larger than 512 bytes, in particular 4 KB long requests.

Our second contribution consists in exploring the modifications of dm-crypt and the underlying Atmel AES drivers required in order to support longer encryption/decryption requests.

Finally our tests turned out to be extremely efficient for the mixed XTS-AES mode implementation and full hardware implementation. A performance gain almost of a factor two was observed by moving to 4 KByte compared to the original 512 byte version.

Our third contribution consists in a benchmark of the various XTS-AES implementations with extended requests, showing major performance gains that open the opportunity for new use-cases with low-power, embedded boards.

2 Related works

FDE and XTS-AES have been considered in several previous works. However almost none discuss *dm-crypt* performance. Several papers analysed the security aspects of FDE like Casey et al. in 2011[3] or Henson et al. in 2013[6]. Gotzfried et al.[5] focused on Android FDE and developed a tool to show that with physical access to an encrypted smartphone only (i.e., without user level privileges), the Android system partition can be subverted with keylogging. Other works from Müller et al.[8] in the same way have developed a tool to perform cold boot attack and retrieve sensitive data from RAM.

XTS mode is approved by NIST for block oriented storage devices[4]. A detailed presentation is described in[7]. Alomari et al.[2] give details of a parallel XTS implementation in multiple cores which enhances performance by 90 %. Shakil [1] presents an implementation on FPGA.

3 Background: Full Disk Encryption, *dm-crypt*, XTS-AES and crypto-engines

3.1 Block Devices and Full Disk Encryption (FDE)

A block device is an abstraction required to access such hardware devices as hard drives that provide buffered access to the device. However, the notion of "block" is sometimes misleading and two types of "blocks" exist:

– the physical sector on a disk, which is the fundamental unit of all block devices. The size of those blocks, on most devices, is 512 bytes;
– the logical block, which is an abstraction of the file system and corresponds to the smallest logically addressable unit. The usual values are 512, 1024 and 4096 bytes.

In this work, we also consider Full Disk Encryption (or FDE), i.e., an encryption of the underlying block device. FDE is now popular, including with smartphones that often natively encrypt their user/data partitions (e.g., Android relies on a *dm-crypt* module). Initially, the encryption of block devices was based on the CBC-AES mode. However NIST proposed to move to a new standard, XTS-AES mode in [4] and new FDE (e.g., Filevault2 for MacOS) all moved to this new cipher. XTS-AES is significantly more complex than CBC-AES, but it has two key advantages: it can be parallelized and it works natively on such fixed data units as blocks. These features make XTS-AES well suited to block device ciphering.

3.2 LUKS/*dm-crypt*

In Linux, the *dm-crypt* kernel module is used to create encrypted containers and is available since version 2.6. *dm-crypt* is implemented as a device mapper target, i.e., it provides transparent encryption of block devices using the kernel crypto API. Each data read from the device is decrypted and conversely each data written to the device is encrypted before being stored on the disk. The kernel crypto API offers a rich set of cryptographic ciphers as well as other data transformation mechanisms and methods to invoke them. For the sake of universality, the kernel crypto ciphers are software based (i.e., the CPU performs all the encryption/decryption operations). An existing hardware crypto-engine will not be used by default, unless a dedicated low level driver for that module exists and replaces the software methods with its own hardware accelerated methods. Figure 1 presents a high-level overview of the global architecture.

Two tools exist to create the device mapper target. The first one, `dmsetup`, offers a low-level interface and must be used by experienced users only. Data is directly encrypted and managed by the user, all the parameters (key included) being provided as command line arguments.

The second tool is more user-friendly and is based on LUKS (Linux Unified Key Setup). LUKS creates a header on the disk with all the required information such as the cipher mode, the salt and the hash of the master key. Moreover LUKS allows to have up to 8 users per container. A key is then derived from the user passphrase with PBKDF2 and is used to encrypt or decrypt the master key. *dm-crypt* supports different ciphers but the most recent default mode is XTS-AES.

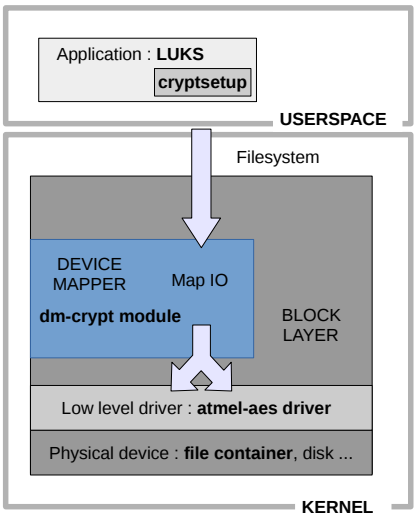

Fig. 1: High-level overview of the global architecture.

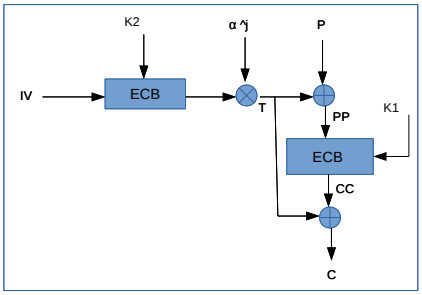

Fig. 2: XTS-AES encryption process

3.3 About the XTS-AES mode

The XTS-AES mode is not included in all embedded boards, essentially because this is a relatively recent mode of operation that is a relatively complex (it involves two ECB-AES encryptions, a multiplication in a Galois Field, and two XOR operations). This mode has been designed to work with fixed-size data units, for instance logical disk blocks, and each data unit must be processed separately and independently of other data units. When applied to a block device, this feature implies that the computations required to modify the content of a block of a fully encrypted device will be localized to this block, without any impact on the adjacent blocks. This is a key asset when compared to modes that rely on a chaining approach.

Three components are necessary to implement XTS-AES:

K: a key that is divided into two equal-sized sub-keys, K_1 and K_2. K_1 is used to encrypt/decrypt data while K_2 is used for IV encryption;
IV: a 128 bit/16 byte value that represents the logical position of the data unit. This IV, once encrypted, is called *tweak* in XTS-AES;
P: a 512 byte data unit that is used as the payload to encrypt or decrypt.

Let us consider a 512 bytes block. It is composed of 32 data units of 128 bits/16 bytes each. Let j denote the sequential number of the 128 bit data unit inside the block. Fig 2 shows the encryption process for such a data unit. The first step consists in encrypting the IV with K_2 using the AES-ECB mode. The result is multiplied (in the Galois field) with the j^{th} power of α to produce T,

where α is a primitive element of GF(2^{128}). Then the 128 bit data unit (plaintext) is XORed with T and encrypted with K_1 using AES-ECB mode, resulting in CC. The last step consists in XORing the output CC with T, producing the encrypted result for this 128 bit data unit. The same operation is performed for all the 128 bit data units, successively.

A pseudo code is shown in Algo. 1. It relies on two functions:

computeTweak(IV): computes 32 tweaks from an encrypted IV (eIV), using the multiplication in the Galois Field;

AESEncECB(P, K): encryption of plaintext P with key K to produce ciphertext C. This function uses the ECB-AES cryto engine.

4 Implementation of extended request for XTS-AES

We designed and benchmarked two XTS-AES drivers:

- a version that leverages the hardware assisted ECB-AES, mixed with software multiplications on the Galois field and software XOR operations. This version operates on data chunks limited to **512 bytes**;
- an optimized version of this driver and the *dm-crypt* component, where the data chunks are extended to **4 kilo-bytes**.

These XTS-AES drivers will be compared to the original full-software XTS-AES function of the Linux kernel crypto API, used as the **reference**.

4.1 Support of XTS-AES mode with hardware assisted ECB-AES operations

The Atmel driver does not natively support the XTS-AES mode. However, as seen in Section 3.3, XTS-AES is composed of two ECB-AES operations that are supported by the crypto-engine of our Atmel SAMA5D3 board. Therefore, we designed a first version of the XTS-AES encryption function, atmel_aes_xts_encrypt(), making use of the ECB-AES hardware crypto-module, as illustrated in Algo.1. Since *dm-crypt* is left unchanged in this version, the data chunks managed by the XTS-AES Atmel driver are still limited to 512 bytes.

4.2 Addition of the extended request mode to the Atmel XTS-AES driver

Then, we modified the Atmel XTS-AES driver and *dm-crypt* in order to relax the 512 byte limit. Because the extended mode handles larger requests, we added the following function to the Atmel XTS-AES driver:

generateIVs(IV, *nblocks*): generates *nblocks* IVs by incrementing each previous IV by one. This is required since we receive a single IV, corresponding to the first block. The maximum size of IVs table is 128 bytes (i.e., 8×16 bytes).

The new algorithm, implemented in the new atmel_aes_xts_encrypt() function, is described in Algo.2. We see that:

- the first encryption operation, $AESEncECB(IVs, K_1)$, is performed on a single chunk of up to 128 bytes of data;
- the second encryption operation, $AESEncECB(PP, K_2)$, is performed on a single chunk of 4 KB.

This has to be compared to applying Algo. 1 8 times, for each chunk of 512 bytes contained in the 4 KB block. We see that significant improvements should be achieved from this optimization.

Algorithm 1: XTS-AES encryption, atmel_aes_xts_encrypt().

input : key split in ($K_1 \parallel K_2$);
IV (16B);
plaintext P (512B);
output: ciphertext C (512B)

```
1 eIV ← AESEncECB (IV, K1)
  // tw_buf is a 512-byte buffer
  // that contains 32 16-byte
  // tweaks
2 tw_buf ← computeTweak (eIV)
3 PP ← tw_buf ⊕ P
4 CC ← AESEncECB (PP, K2)
5 C ← tw_buf ⊕ CC
```

Algorithm 2: Modified atmel_aes_xts_encrypt() with extended requests.

input : key split in ($K_1 \parallel K_2$);
IV (16B);
plaintext P (multiple of 512B);
output: ciphertext C

```
1 nblocks ← sizeof (P)/ 512
2 IVs ← generateIVs (IV, nblocks)
3 eIVs ← AESEncECB (IVs, K1)
  // tw_buf is a buffer that
  // contains 32 × nblocks
  // 16-byte tweaks
4 for i ← 0 to nblocks do
5 |   tw_buf [i × 512] ←
  |   computeTweak (eIVs [i × 16])
6 PP ← tw_buf ⊕ P
7 CC ← AESEncECB (PP, K2)
8 C ← tw_buf ⊕ CC
```

4.3 Addition of extended requests to *dm-crypt*

When the kernel decides that a set of blocks must be transferred to or from a block I/O device, it uses a *bio* structure to describe this operation. Then data is transferred to the low level driver (atmel-aes) with a *scatterlist*. In the following we will briefly explain each structure in order to understand how we improved the original performance.

Data is first gathered in a *bio* structure as shown in Figure 3. This is as a list of segments, where each segment is a chunk of a buffer that is contiguous in memory. However, individual buffers need not be contiguous in memory. This data structure allows to perform block I/O operations as a single buffer from multiple locations in memory. The first segment is pointed by the *bio_io_vec* field. Structures are then stored as an array of *bio_vec* structures. Each *bio_vec*

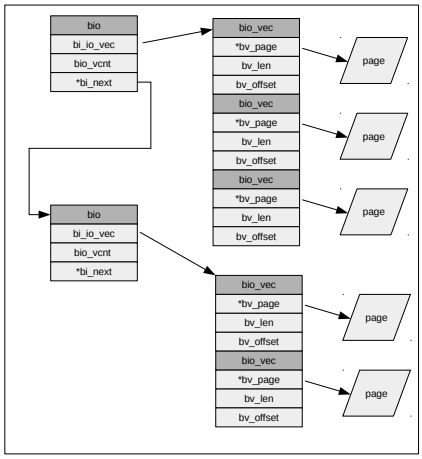

Fig. 3: *bio* structure details

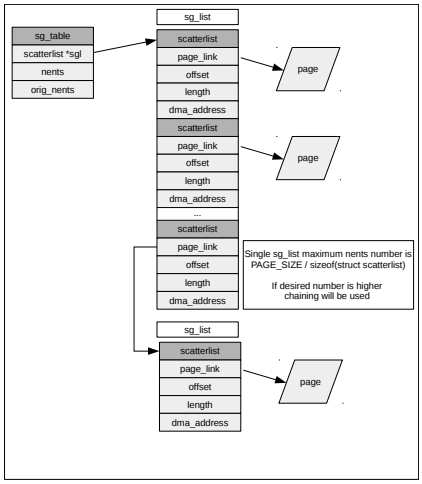

Fig. 4: *scatterlist* structure details

is treated as a vector of the form <page, offset, len> where page denotes the associated physical page, length denotes the data size starting from the offset in the page. Another field in the *bio* structure is *bio_next* which is a pointer to a new *bio* (block I/O could be stored as a chained list of bio structures).

To achieve high performance I/O, the use of Direct Memory Access (DMA) is needed. With DMA the I/O device transfers data to or from memory without intervention of CPU. However, in order to perform as a single operation data needs to be stored contiguously in memory. This is done thanks to an array of *scatterlist* structures. Each structure (called segment) is very similar to a *bio_vec* and is composed of three fields <page, offset, len>. An additional field is the *dma_address* which is an address given to the peripheral. We call *sg_list* the array of *scatterlists* as in the Figure 4. One can create *sg_list* directly but the allocation should be done by hand or through a *sg_table*. The *sg_list* must fit within a single page, which limits the number of segments to PAGE_SIZE/sizeof(struct *scatterlist*). As a result an *sg_table* could be composed of a chained list of *sg_list* for large I/O.

The last step is the relation between both structures: *bio* and scatterlist. The link between both is made in the *dm-crypt* module. Figure 5 illustrates this transfer. As we have seen, the main data size in memory is the page size. Each page has been described as segments. Segments are involved within *bio* and *scatterlist*. The first step is to retrieve a single page of a *bio* through *bio_vec*. Then (step 2) two *scatterlists* (only one is represented) are initialized for source and destination with each one a single segment (one page). The *set_page* function fills the *scatterlist* with the page pointed by the *bio_vec* starting from offset 0 and with a data size of 512 bytes. The third step is to create a *dm-crypt* request

with some information about data to encrypt or decrypt and the recently created *scatterlists*. The fourth step is to send the request to atmel-aes driver in order to process it.

Thus, to process a single 4096 bytes page, steps 2, 3 and 4 are executed 8 times (offset is incremented by 512 each time). After the completion of the current page, the next page is retrieved from the *bio* (step 1) and is treated in the same manner.

Algorithm 3: Legacy crypt_convert().

```
   input :
   # bytes to transfer, nbytes;
   initial sector number, sec;
   sector shift size, sec_shift;
   (sec_shift is hard coded as 512B)
   plaintext, in;
   ciphertext, out;
1  rem ← nbytes
2  while rem > 0 do
       // sec is used as IV below
3      cryptConvertBlock (sec,
       sec_shift, in, out)
4      sec ← sec + 1
5      rem ← rem − sec_shift
```

Algorithm 4: Modified crypt_convert() with extended requests.

```
   input :
   # bytes to transfer, nbytes;
   initial sector number, sec;
   sector shift size, sec_shift;
   plaintext, in;
   ciphertext, out;
1  rem ← nbytes
2  while rem > 0 do
3      if rem > 4096 then
4          sec_shift ← 4096
5      else
6          sec_shift ← rem
       // sec is used as IV below
7      cryptConvertBlock (sec,
       sec_shift, in, out)
8      sec ← sec + sec_shift/512
9      rem ← rem − sec_shift
```

This transfer is done in particular thanks to two functions: *crypt_convert* and *crypt_convert_block*. Algo. 3 shows an extremely simplified version of the first function.

When a I/O block request is handled by *dm-crypt*, it eventually goes to *crypt_convert* which iterates through the *bio* structure and runs a block encryption on the associated plaintext. In this method, *remaining* is the remaining size of data to encrypt from *in* to *out*, *sector_shift* is the offset for the next block, and *req* is the request generated from the variables mentioned before. This encryption is handled by the *crypt_convert_block* method which first of all transfers data pointer (page) into a single *scatterlist* of 512 bytes, and this scatterlist is then encrypted/decrypted by the appropriate cipher (XTS-AES in our case).

As we see, the main problem is that the actual solution is based on a specific *sector_shift* which is used to iterate by 512 bytes blocks. What we need to do then is changing this behavior by adding the ability to use 4096 bytes blocks, which is supposed to be the optimal block size due to *scatterlists* page size which is 4096 bytes as well. However to ensure backward compatibility and for other

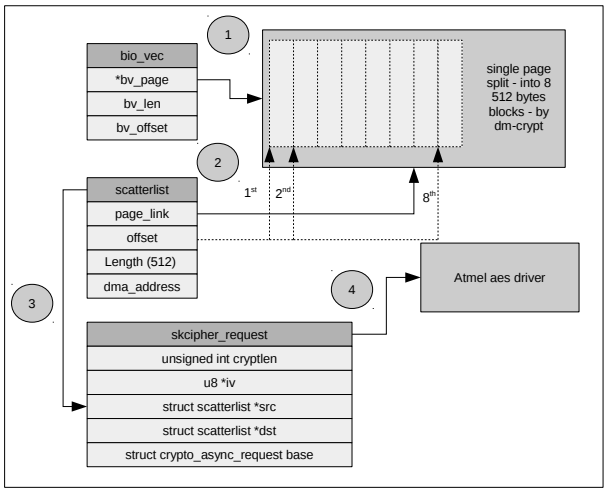

Fig. 5: From bio to scatterlist

specific reasons, like handling the last block of a request if it does not have a size multiple of 4096, we still need to handle 512 bytes blocks.

To do so, we change the sector shift that should be equal to 4096 if the remaining data to encrypt is greater or equal to 4096, and should be equal to the remaining otherwise. Therefore, we replace:
sector_shift = 512
by:
sector_shift $\leftarrow$ *(remaining > 4096) ? 4096 : remaining*
(see Algo. 4).

Therefore the previous way of splitting data into *scatterlists* is totally changed. For instance, if *dm-crypt* receives a 8192 bytes plaintext to encrypt, it will be split into two completely filled *scatterlists* instead of sixteen *scatterlists* filled only with 512 bytes of actual data. As mentioned before, we have chosen a block size of 4096 bytes because it natively corresponds to the page size.

5 Experiments

5.1 Test Procedure

The Atmel SAMA5D3-XPlained board is equipped with a cortex A5 @536 MHz, 256 MB of RAM, a 16 GB class 10 SD card, and runs the Linux 4.6.0-sama5-armv7-r1. It also uses cryptsetup version v1.6.6. For conveniency purposes, the *dm-crypt* module and atmel-aes driver are both compiled as modules. The test procedure is the following:

1. create a file container of 1.5 GB with the `dd` command;

2. create a LUKS container with the following parameters:
   - cipher is aes-xts-plain;
   - key is 256 bits/32 bytes long.
3. create an EXT4 file system in this container and fill it with test files of different sizes;
4. close then reopen the container;
5. measure the time used to compute the MD5sum of the target file;
6. close the container;
7. perform 10 times this open/compute MD5sum/close test.

This test protocol has been chosen to avoid cache problems and to be sure to measure the right time.

5.2 ECB-AES Raw Results

The results in Figure 6 highlight the impact of block size on performance of ECB-AES mode (rather than XTS-AES). ECB-AES is the simplest mode (without IV), it allows us to point out the potential improvements. We clearly notice that the maximum performance is reached when using a 4 KB block size. Figure 7 shows the time required to encrypt a block. To retrieve the time speed in the encryption or decryption step, we added two high-precision `getnstimeofday()` functions. The first one is located in the mapping function before the DMA transfer, the second one is after the DMA interruption when data is retrieved from the DMA.

The results show that the time needed to compute a single 4 KB block is significantly lower than the time required to compute eight blocks of 512 bytes.

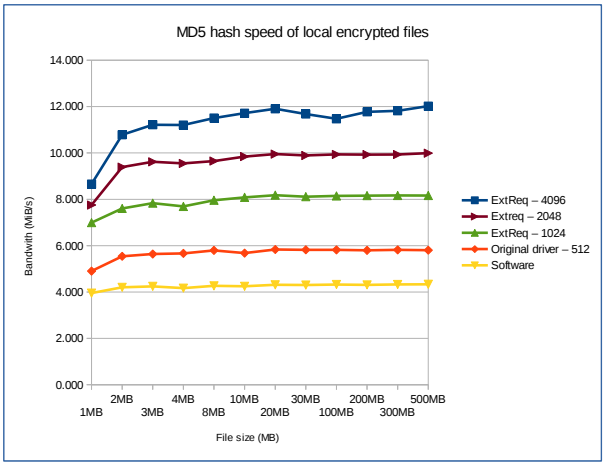

Fig. 6: Performance evolution as a function of the request size for ECB-AES.

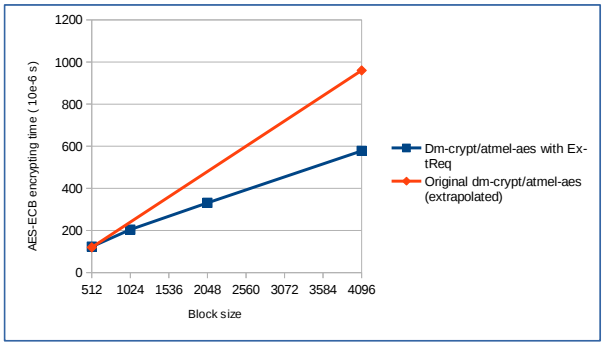

Fig. 7: Time to compute a request with different block sizes for ECB-AES.

5.3 XTS-AES Results

We now compare the performance of the three versions considered in this work:

- the software kernel crypto implementation;
- the atmel-aes driver with our hardware assisted XTS-AES mode (Section 4.1);
- the atmel-aes driver with our hardware assisted XTS-AES mode and extended requests (Section 4.2).

We focus on the decryption time, measured through the MD5 hash. Using a hash of the file allows us to be sure that the whole file content has been decrypted and read, while verifying its integrity.

Figure 8 confirms with XTS-AES mode that the extended request approach is highly efficient. This optimization is in fact mandatory to unleash the potential of crypto-engines. The improvement factor is close to 2 between the original atmel-aes driver/*dm-crypt* and the new one. This result confirms those achieved with ECB-AES.

6 Conclusion

In this work we proposed an approach to take full advantage of the cryptographic engine of an Atmel board running Linux. The original *dm-crypt* module has been patched to increase the size of the request sent to the atmel-aes driver. It enabled our implementation to enhance performance almost by a factor of 2 compared to the original version. These improvements are valid with the XTS-AES mode as well as the ECB-AES mode, as well as other hardware crypto-engines using *dm-crypt* with limited changes in their native drivers.

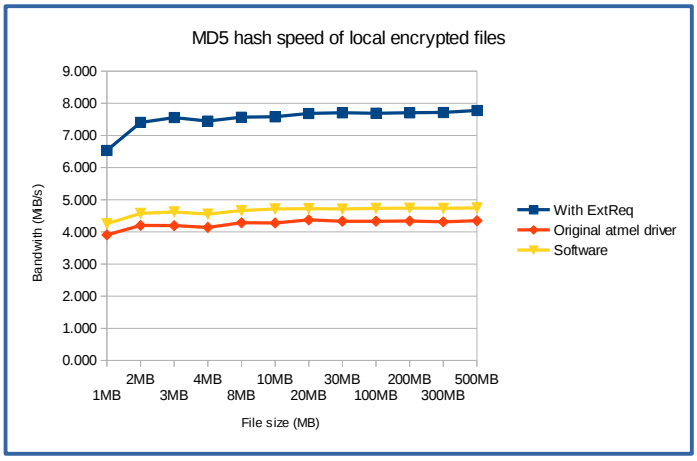

Fig. 8: XTS-AES decryption performance W.R.T. the file size, for the 3 versions.

References

1. S. Ahmed, K. Samsudin, A. R. Ramli, and F. Z. Rokhani. Effective implementation of aes-xts on fpga. In *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 184–186, Nov 2011.
2. M. A. Alomari, K. Samsudin, and A. R. Ramli. A parallel xts encryption mode of operation. In *Research and Development (SCOReD), 2009 IEEE Student Conference on*, pages 172–175, Nov 2009.
3. Eoghan Casey, Geoff Fellows, Matthew Geiger, and Gerasimos Stellatos. The growing impact of full disk encryption on digital forensics. *Digital Investigation*, 8(2):129 – 134, 2011.
4. M Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices," national institute of standards and technology. Technical report, Tech. Rep. 800-38E, 2010.[Online]. Available: http://csrc. nist. gov/publications/nistpubs/800-38E/nist-sp-800-38E. pdf.
5. Johannes Götzfried and Tilo Müller. Analysing android's full disk encryption feature. *JoWUA*, 5(1):84–100, 2014.
6. Michael Henson and Stephen Taylor. *Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors*, pages 307–321. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
7. Luther Martin. Xts: A mode of aes for encrypting hard disks. *IEEE Security AND Privacy*, 8(3):68 – 69, 2010.
8. Tilo Müller and Michael Spreitzenbarth. Frost. In *International Conference on Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.