Optimizing *dm-crypt* for XTS-AES: Getting the Best of Atmel Cryptographic Co-processors

Levent Demir[1,2], Mathieu Thiery[1,2], Vincent Roca[1], Jean-Michel Tenkes[2] and Jean-Louis Roch[3]
[1]*Incas ITSec, France*
[2]*Univ. Grenoble Alpes, Inria, France*
[3]*Univ. Grenoble Alpes, Grenoble INP, LIG, France*

Keywords: Full Disk Encryption, XTS-AES, Linux dm-crypt Module, Cryptographic Co-processor, Atmel Board.

Abstract: Linux implementation of Full Disk Encryption (FDE) relies on the *dm-crypt* kernel module, and is based on the XTS-AES encryption mode. However, XTS-AES is complex and can quickly become a performance bottleneck. Therefore we explore the use of cryptographic co-processors to efficiently implement the XTS-AES mode in Linux. We consider two Atmel boards that feature different cryptographic co-processors: the XTS-AES mode is completely integrated on the recent SAMA5D2 board but not on the SAMA5D3 board. We first analyze three XTS-AES implementations: a pure software implementation, an implementation that leverages the XTS-AES co-processor, and an intermediate solution. This work leads us to propose an optimization of *dm-crypt*, the extended request mode, that enables to encrypt/decrypt a full 4kB page at once instead of issuing eight consecutive 512 bytes requests as in the current implementation. We show that major performance gains are possible with this optimization, a SAMA5D3 board reaching the performance of a SAMA5D2 board where XTS-AES operations are totally offloaded to the dedicated cryptographic co-processor, while remaining fully compatible with the standard. Finally, we explain why bad design choices prevent this optimization to be applied to the new SAMA5D2 board and derive recommendations for future co-processor designs.

1 INTRODUCTION

Data protection is a necessity: large amounts of sensitive information are stored in many different devices, smartphones, tablets and computers. If such devices are lost or stolen, unauthorized access to information could have disastrous consequences (e.g., psychological or economic (LLC, 2010)). We also have to pay attention not only to data at rest, but also to data in different memories like RAM and swap spaces.

One possible approach is to use Full Disk Encryption (FDE), which consists of encrypting an entire disk, content as well as associated metadata, all information being encrypted/decrypted on-the-fly transparently. At the system level, data is stored either in a logical partition or in a file container. Different tools exist for FDE. With Linux, the native solution is based on cryptsetup/LUKS application (Fruhwirth, 2005), within user-space, and the *dm-crypt* module (Brož et al., 2020) within kernel-space, which allows transparent encryption and decryption of blocks.

A crucial aspect for FDE is the cipher mode of operation, AES being the main cipher choice. Until 2007, the standard for data encryption in FDE was the CBC-AES mode. But this mode has several drawbacks. For instance, as explained in (IEEE Computer Society, 2008): "an attacker can flip any bit of the plaintext by flipping the corresponding ciphertext bit of the previous block" which can be dangerous. Furthermore, encryption is not parallelizable which is an issue for certain use cases.

A new mode has been introduced in 2008, XTS-AES (IEEE Computer Society, 2008) that solved the two previous limitations as 16-bytes block encryption/decryption is now performed independantly of any previous 16-byte block. Each 16-byte block can be accessed in any order and parallelization is possible during both encryption and decryption. In spite of that, XTS-AES encryption/decryption operations are complex and the use of this mode in lightweight environments over huge amounts of data is challenging.

The motivation for this work is to offload all XTS-AES cryptographic operations to a dedicated board in charge of FDE. This feature can be useful to design a security board that would handle all cryptographic operations required to outsource user's data in external, untrusted storage facilities (e.g., a Cloud). This architecture, with a security board between the client and the storage facility, was our initial goal that triggered the present work. The question of XTS-AES mode

Demir, L., Thiery, M., Roca, V., Tenkes, J. and Roch, J.
Optimizing dm-crypt for XTS-AES: Getting the Best of Atmel Cryptographic Co-processors.
DOI: 10.5220/0009767802630270
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 263-270
ISBN: 978-989-758-446-6

performances improvement in embedded, lightweight environments, is therefore critical.

Choice of Atmel Boards and Importance of Detailed Technical Specifications: We considered two Atmel boards, both equipped with a cryptographic co-processor, the (old) SAMA5D3 board (ATMEL, 2017b) and the (new) SAMA5D2 board (ATMEL, 2017a). We chose these because of their low price and wide acceptance in industrial systems, and because the cryptographic co-processor documentation is publicly available, a requirement for advanced developments. This is not always the case as we discovered after buying another more powerfull board: the provided information turned out to be too limited for our needs and our academic status did not enable us to obtain the technical documentation from the manufacturer, even after asking their support.

A major difference exists between these two Atmel boards, which justifies that we consider both of them: the cryptographic co-processor of the first board supports common AES modes but not XTS-AES, while the second one also supports XTS-AES. Those constraints led us to consider different implementation options that are the subject of this work.

Scientific Approach Followed in this Work: The first step of our work was the experimental analysis of three XTS-AES implementations: a pure software implementation (the legacy baseline), an implementation that leverages the dedicated cryptographic co-processor with XTS-AES support of the SAMA5D2 board (the most favourable case), and in between an implementation that leverages the cryptographic co-processor with ECB-AES support only of the old SAMA5D3 board. Our benchmarks demonstrated that the performance in all cases was still behind expectations and did not match our objective of efficient on-the-fly encryption/decryption of large amounts of data within the Atmel boards.

An analysis of in-kernel data paths highlighted a limitation of plaintext sizes to a hard-coded 512 bytes value, in particular because this is the common sector size on most devices, and also because test vectors are limited to a maximum of 512 bytes in the official XTS-AES standard (IEEE Computer Society, 2008). We therefore explored the possibility of having 4 KB long requests (i.e., a page size), a rational choice and a pretty natural idea for kernel operations. We called this optimization "extended request mode", or *extReq*.

We therefore modified *dm-crypt* as well as the underlying *atmel-aes* driver, two highly complex tasks, in order to support extended encryption/decryption requests. We then analyzed the performance impacts. With this optimization, a mixed implementation with the (old) SAMA5D3 ECB-AES co-processor features roughly the same performance as that of the (new) SAMA5D2 XTS-AES co-processor.

Finally we analyzed the existing XTS-AES cryptographic co-processor of the SAMA5D2 board in order to apply the *extReq* optimization to it directly. Unfortunately, because of bad design choices by Atmel, this new cryptographic co-processor is not compatible with this optimization, therefore limiting the opportunities for major performance improvements. We explain why it is so and conclude this work with recommendations for future co-processor designs (the interested reader is invited to refer to the full paper: https://hal.archives-ouvertes.fr/hal-02555457).

Note that this work only considers cryptographic operations over large data chunks, which is pretty common with FDE use-cases. It does not consider the opposite case, i.e., large numbers of small data chuncks, which is not the target of our optimisation.

Contributions of this Work:

- this works explores the implementation of cryptographic primitives in Linux systems, detailing the complex interactions between software and hardware components, and the *dm-crypt* kernel module internals. Note that this work implied major in-kernel low-level software developments and complex performance evaluation campaigns.
- this work shows that significant performance gains are possible thanks to the "extended request mode", *extReq*, optimization, even with boards that do not feature cryptographic co-processors supporting XTS-AES. Although the idea behind this optimisation is pretty natural, we describe the architectural implications, we apply it to several XTS-AES implementations, depending on the available hardware, and provide performance evaluation results. Note that even if this work only considers embedded boards, it will be useful to other execution environments.
- when we tried to apply the *extReq* optimization to the XTS-AES facility of the new cryptographic co-processor, we discovered an uncompatible design. We explain why it is so, we provide likely explanations for this situtation, as well as recommendations for future co-processor designs. This is an important outcome of this work if we want to boost FDE cryptographic performance.

2 FULL DISK ENCRYPTION (FDE) IMPLEMENTATIONS ON LINUX

2.1 About FDE in Linux

Since Linux kernel v.2.6, FDE relies on the *dm-crypt* kernel module. It provides transparent encryption/decryption of a virtual block device using the kernel crypto API, in which the block device can be a logical partition, an external disk (HDD or USB stick) or a file container. Data written/read to/from the device is automatically encrypted/decrypted.

The kernel crypto API offers a rich set of cryptographic ciphers, modes, and data transform mechanisms. Natively the crypto API offers its own generic software ciphers: since the cryptographic operations are performed by the CPU, this cipher is portable, without any assumption on available hardware. When another implementation exists for the same cipher (see section 2.3), it is used instead of the generic one.

On top of the kernel module, FDE relies on *cryptsetup*, which in turn is based on Linux Unified Key Setup (LUKS). LUKS provides a standard on-disk header with all required information like cipher mode, salt and hash of the master key (Fruhwirth, 2018). It also provides a secure user management system that allows up to eight users to share a single container.

Figure 1 presents a high-level overview of the global architecture and summarizes the various operations on data between user-space, kernel space, and physical device.

2.2 About XTS-AES

XTS-AES is the standard cipher mode for block-oriented storage devices since 2007 (IEEE Computer Society, 2008). The block size of this mode matches the block size of the storage device: 512 bytes[1].

The sector number corresponding to a 512-byte block is used as IV, which means that the encryption/decryption operations can be done independently for each block, and in parallel if needed.

Let us consider XTS-AES encryption (refer to (Martin, 2010; IEEE Computer Society, 2008) for decryption). There are three input parameters:

- **The key,** K**,** is 256 or 512 bits long and 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.
- **The initialization vector,** IV**,** is 128 bits/16 bytes long and represents the sector number (i.e., the logical position of the data unit). This IV, once encrypted, is called eIV. After multiplication, it forms the *tweak*, denoted as T.
- **The plaintext** P is 512 bytes long block and constitutes the payload to encrypt.

[1] We discuss in the long version of this paper that this block size value significantly impacted the design of the Atmel XTS-AES co-processor, thereby preventing us from applying our optimization on this board.

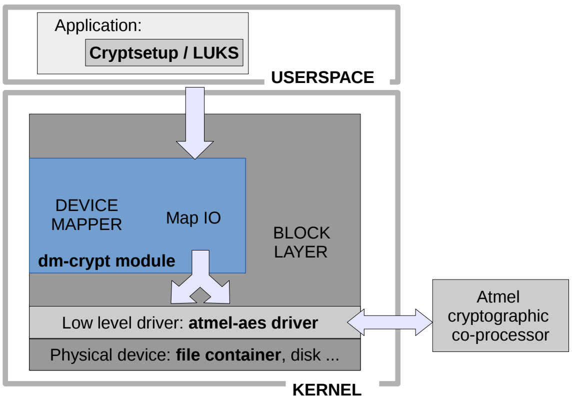

Figure 1: High-level overview of the FDE architecture in Linux and the data paths during cryptographic operations. *Within the user-space, cryptsetup allows to create an encrypted disk following LUKS format. When plaintext needs to be encrypted, data is sent to the dm-crypt module, within the kernel space. Depending on what is available, a software or cryptographic co-processor based implementation is chosen. When a co-processor is available, as is the case here, data is transferred to the specific low level driver. Finally the ciphertext is written to the physical device.*

A 512-byte block is composed of 32 data units of 128 bits/16 bytes each. Let j denote the sequential number of the 128 bits data unit inside this block. Figure 2 shows the encryption process for this data unit. The first step consists in encrypting the IV with K_2 using AES-ECB to produce the eIV. The result is multiplied in the Galois Field with the j^{th} power of α to produce the tweak, T, where α is a primitive element of $GF(2^{128})$. Then the 128 bits data unit (plaintext) is `XOR`ed with T and encrypted with K_1 using AES-ECB, resulting in CC. The last step consists in `XOR`ing CC with T, producing the encrypted result C for this 128 bits data unit. The same operation is performed for all the 128 bits data units, successively.

2.3 XTS-AES Implementations

Cipher implementations are available at different levels including from userspace, through libraries such as OpenSSL, GnuTLS or Gcrypt. Within the Linux kernel, other ciphers are used:

- some of them are pure software implementations;

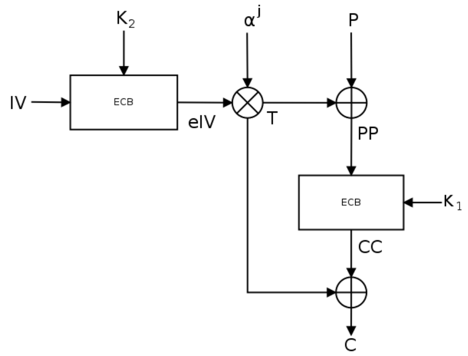

Figure 2: XTS-AES encryption of the j^{th} 128 bits data unit of a 512-byte block.

- other ciphers use specific CPU instructions like AES-NI (Gueron, 2012) for Intel CPU, or ARMv8 Crypto Extensions for ARM processor. They offer a clear performance benefit compared to generic software implementations;
- finally some implementations rely on a dedicated cryptographic co-processor. This approach usually features better performance than generic software, but on the downside, the co-processor acts as an unmodifiable black box.

In the next section we introduce a fourth solution which leverages on the SAMA5D3 cryptographic co-processor ECB-AES support.

3 OPTIMIZING *DM-CRYPT* FOR XTS-AES

3.1 Accelerating XTS-AES with an ECB-AES Co-processor

For situations where a cryptographic co-processor is available and supports ECB-AES but not XTS-AES (e.g., the Atmel SAMA5D3 board, section 4.1), a mixed approach is possible. XTS-AES is composed of five operations: two `XOR` operations, a multiplication in GF(2^{128}), two ECB-AES encryptions (or decryptions). The idea is to offload the two ECB-AES operations onto the cryptographic co-processor while other operations are performed by the CPU. Doing so requires to modify the *atmel-aes* driver. The main difficulty is to accommodate the asynchronous nature of the cryptographic co-processor operations: the interruption generated at the end of the ECB-AES encryption (or decryption) by the co-processor is intercepted and triggers the remaining CPU operations.

As we will see later on, the performance gain achieved was not as high as expected and we looked at another possible optimization.

3.2 Extended Requests to the *atmel-aes* Driver

We also analyzed the mapping between 4kB pages managed by the *dm-crpyt* module and the low level cryptographic operations within the *atmel-aes* driver. Let us consider the encryption operation in Figure 3 (decryption is similar):

- The *dm-crypt* module gets a description in a *bio* structure of the plaintext file (this *bio* is not represented in Figure 3). This *bio* structure consists of a list of *bio_vec* structures, one per 4kB page.
- For each 4kB page of the list, the *dm-crypt* module splits this page into eight 512-byte blocks and initializes two *scatterlist* structures for each block, respectively for source (where the plaintext is) and destination (where to write the ciphertext). The *offset* in the page is incremented for each *scatterlist* to point to the right 512-byte block.
- Then an encryption request is generated for each block, with complementary information (like the IV) and sent to the *atmel-aes* driver.
- Finally the *atmel-aes* driver encrypts each 512-byte block, writing the result to the destination.

It appears that a natural optimization would consist in working with larger requests to the *atmel-aes* driver, a full 4kB page at a time. Doing so reduces by a factor eight the number of requests and reduces the impact of fixed overheads within the cryptographic co-processor (e.g., when programming a DMA to move data from a kernel buffer to the internal co-processor memory, and vice-versa).

We also limit ourselves to 4kB pages (rather than a list of pages) because the page size is the common size for file processing. It follows that the various pages are not necessarily contiguous on disk which limits the benefit of having a single request larger than a page.

This optimization requires modifying both *dm-crypt* and the *atmel-aes* driver. We increased the *dm-crypt* 512 bytes limit to 4kB. Of course, the original *dm-crypt* behavior is preserved and used if less than a full page is concerned.

The *atmel-aes* driver is also modified. Again, any request size from 512 bytes to 4kB (with a 512 bytes step) is accepted. For instance, with an extended request for a full page, the driver computes eight IVs, by incrementing the initial IV value for each 512-byte block. This is in line with the way data is stored in the page, since the eight blocks are necessarily stored sequentially. The driver also computes eight times more tweaks from these IVs, and performs XOR and ECB-AES encryption operations 4kB at a time.

This approach *is fully backward compatible*, which we experimentally checked: a plaintext encrypted using this optimized extended request mode can be decrypted with a classic XTS-AES mode implementation, and vice versa.

4 EXPERIMENTS

4.1 Experimental Platform

We implemented the proposals of section 3. In order to assess the performance of the various options, we considered two Atmel boards: the SAMA5D3 (ATMEL, 2017b) and the SAMA5D2 (ATMEL, 2017a) boards. Both cards feature the same single core Cortex A5 ARM processor, 500 MHz, and a specific cryptographic co-processor. The SAMA5D3 co-processor supports five common AES modes, but not XTS-AES. On the opposite, the SAMA5D2 co-processor, more recent, also supports XTS-AES. Otherwise both cards feature 256 MB of RAM, a Sandisk Class 10 SDHC card, and run the same Linux/Debian operating system with a 4.6 Linux kernel. During all tests, we used the default key size of *dm-crypt*: a 256-bit XTS-AES key, divided into two 128-bit sub-keys, which means that ECB-AES-128 mode is always used.

Here are the various configurations tested:

SAMA5D3 Board:

- **software:** existing `xts.ko` linux kernel module;
- **mixed, with ECB-AES co-proc. but not *extReq*:** *atmel-aes* driver modified to use the co-processor for ECB-AES operations and CPU for other operations, with 512-byte request sizes;
- **mixed, with ECB-AES co-proc. and *extReq*:** same as above, with 4kB request sizes (full page).

SAMA5D2 Board:

- **software:** existing `xts.ko` linux kernel module;
- **with XTS-AES co-proc.:** cryptographic co-processor for the full XTS-AES processing, with 512-byte request sizes.

In these tests the two ”full software” configurations enable us to calibrate the two Atmel boards. As anticipated from the specifications, we show in section 4.3 that these ”full software” configurations exhibit similar performances. Therefore the results obtained on the SAMA5D2 board can be safely compared to results obtained on the SAMA5D3 board, the main difference being the cryptographic co-processors, not the remaining of the execution environment.

4.2 Time Breakdown with or without Extended Requests

Let us first focus on our mixed implementation using the ECB-AES co-processor, with or without extended requests. We measured the total time spent within the *atmel-aes* driver for each of the five operations of XTS-AES mode on the SAMA5D3 board, during a large 50 MB file encryption and decryption. To that purpose, we instrumented the driver and collected timestamped traces with the `getnstimeofday()` and `printk()` Linux kernel functions. In order to assess the practical precision of `getnstimeofday()` and `printk()`, we ran several consecutive calls and measured a 330 ns overhead per measure, which is an acceptable precision for our experiments.

The breakdown values reported in Tables 1 and 2 are obtained by summing all the elementary times for each of the following categories over the full file encryption or decryption:

- Total time spent in the *atmel-aes* driver;
- Tweak computation time;
- First XOR time;
- Second XOR time;
- DMA (to and from the co-processor) + encryption (resp. decryption) time. Note that unfortunately these operations cannot be isolated from the *atmel-aes* driver;
- Other time computed as the difference between the total time and the previous four categories;

Let us focus on the encryption of this 50 MB file first. From Table 1 we see that the DMA plus encryption process takes more than the half of the total time in the default configuration, with 512-byte requests: $5.31s$ out of $9.09s$, followed by the tweak computation, with a total of $2.62s$. Both of them amount to 87% of the total time.[2]

Using the extended request mode, *extReq*, the total processing time is reduced by a factor 1.74, down to $5.23s$. Looking at the DMA plus encryption process, if it still represents more than half of the total time,

[2] Looking more carefully one can notice that the first XOR is significantly faster than the second one. This difference may come from cache behaviors, the second XOR using a data area initialized by the DMA unlike the first one. Since the impacts are marginal compared to other processing times, we did not investigate the topic more in details.

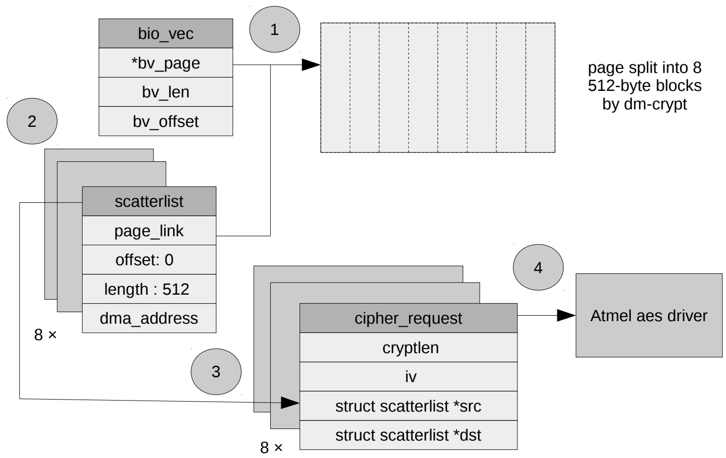

Figure 3: Classic approach for the encryption of a 4kB page with *dm-crypt*.

Table 1: Time breakdown of 50 MB file encryption with the mixed implementation using the ECB-AES co-processor, without or with extReq.

	Without extReq		With extReq	
	Time (s)	%	Time (s)	%
Total time	9.09	100.00	5.23	100.00
Tweak computation time	**2.62**	**28.90**	**1.70**	**32.61**
First XOR time	0.31	3.41	0.31	6.08
Second XOR time	0.63	6.99	0.41	7.86
DMA + encryption time	**5.31**	**58.46**	**2.75**	**52.72**
Other time	0.20	2.24	0.03	0.73

we notice a major improvement by a factor 1.93, now amounting to $2.75s$. The tweak computation is also significantly reduced by a factor 1.54, now amounting to $1.70s$.

The situation is pretty similar during the decryption of this 50 MB file. These results show that the extended request optimization has a considerable impact when we use the dedicated hardware, by reducing the overhead due to the set up of the cryptographic co-processor and the multiple data transfers, which is not surprising.

4.3 Benefits of Extended Requests to the Global Processing Time

We now consider the global processing time with our mixed implementation using the ECB-AES co-processor. This global time now includes *dm-crypt* processing, I/O operations, and all the remaining system call/kernel processing overheads. In particular we want to see to what extent the extended request mode can improve this global time, beyond the benefits it has on the *atmel-aes* driver itself (section 4.2).

However the total time for the encryption and decryption of file is difficult to measure because of asynchronous operations and the presence of caches. In

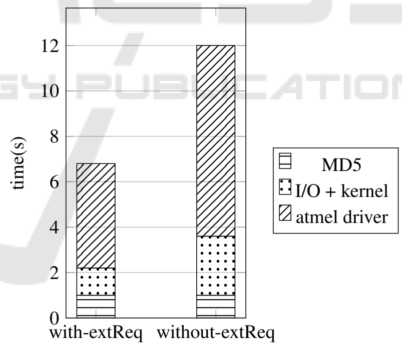

Figure 4: Time breakdown for the MD5 computation of a 50 MB encrypted file, with our mixed implementation using the ECB-AES co-processor.

order to circumvent these difficulties, we measured the time to compute the MD5 digest of an already encrypted file, i.e. the time to decrypt and then compute the MD5 hash. Therefore the total time is composed of the Atmel driver time (line 1 of table 2), the MD5 digest time which is constant, and the I/O and other kernel processing time. We have:

$$t_{total} = t_{md5} + t_{I/0\_and\_kernel\_processing} + t_{atmel\_driver}$$

Here also, we focus on our mixed implementation using the ECB-AES co-processor in order to assess the impacts of the *extReq* optimization.

Table 2: Time breakdown of 50 MB file decryption with the mixed implementation using the ECB-AES co-processor, without or with extReq.

	Without extReq		With extReq	
	Time (s)	%	Time (s)	%
Total time	9.30	100.00	5.13	100.00
Tweak computation time	**3.05**	**32.78**	**1.46**	**28.61**
First XOR time	0.29	3.21	0.27	5.32
Second XOR time	0.55	5.98	0.40	7.92
DMA + decryption time	**5.22**	**56.12**	**2.81**	**54.83**
Other time	0.17	1.90	0.17	3.32

Figure 4 shows the breakdown of the total time. It confirms that the MD5 hash processing is both constant and small with respect to the total time: the method followed is not negatively impacted by the computation of a MD5 digest. Non surprisingly, decryption within *atmel-aes* driver represents the most important time, and is significantly reduced as was shown before. But we also notice that the I/0 and other kernel processing times is divided by a factor of almost 2: this is an additional benefit of the extended request mode.

4.4 Performance Comparison for All Configurations

So far we only focused on our mixed implementation using the ECB-AES co-processor. Let us now compare the various ciphers listed in section 4.1, using either the SAMA5D3 and SAMA5D2 cards. In order to perform this comparison, we considered the IOZONE tool (Norcott and Capps, 2003) that provides encryption and decryption throughputs for large files (256 MB and 512 MB files in our tests).

Figure 5 shows the results. First of all, the two "full software" configurations exhibit similar performance which means the results can be safely compared even if two different Atmel boards have been used. These experiments show that our mixed implementation with ECB-AES co-processor and *extReq* exhibits similar performance to that of the SAMA5D2 XTS-AES co-processor: our solution is slightly slower during encryption, but slightly faster during decryption, no matter the file size. All other solutions are clearly behind.

These experiments outline that *in the absence of native XTS-AES co-processor support, an implementation that can leverage an ECB-AES co-processor and extReq is highly competitive*.

5 CONCLUSION

XTS-AES is complex and can easily become a performance bottleneck when dealing with large amounts of data in the context of Full Disk Encryption (FDE). If this is a perfect target for a hardware cryptographic co-processor, XTS-AES is also relatively recent and not universally supported. For this work we chose two SAMA5 Atmel boards, in parts because of the availability of technical information required by our advanced developments (this is not always the case). If the two boards feature a cryptographic co-processor, only the recent SAMA5D2 supports XTS-AES hardware acceleration.

This work focused on FDE in Linux, where the *dm-crypt* module is in charge of block device, low level, encryption/decryption. We studied three XTS-AES implementations, from a pure software implementation (baseline) to an implementation relying on the SAMA5D2 XTS-AES cryptographic co-processor (most favourable case), and in between an implementation relying on the SAMA5D3 cryptographic co-processor for ECB-AES and CPU for the other operations. We benchmarked them and identified that performance was behind expectations.

Therefore we explored the inner working of the *dm-crypt* module and identified a possible optimization: extended requests. Indeed, sending a single encryption or decryption request to the *atmel-aes* driver for a full 4kB page (instead of eight consecutive requests) enables major performance improvements. Although this idea is pretty natural, we describe the architectural implications, and provide detailed performance evaluations achieved with modified the low level drivers. With this optimization, a mixed implementation limited to the old SAMA5D3 ECB-AES co-processor features roughly the same performance as that of the new SAMA5D2 board with an XTS-AES co-processor. It therefore opens news perspectives to accelerate FDE on Linux: old systems without XTS-AES co-processor support will be greatly accelerated for intensive encryption/decryption tasks.

This work also discusses the possibility of having an extended request mode support in the SAMA5D2 XTS-AES cryptographic co-processor. If the current cryptographic co-processor design, limited to 512-byte blocks maximum, prevents this optimization, we explain how to solve the problem. We hope that the

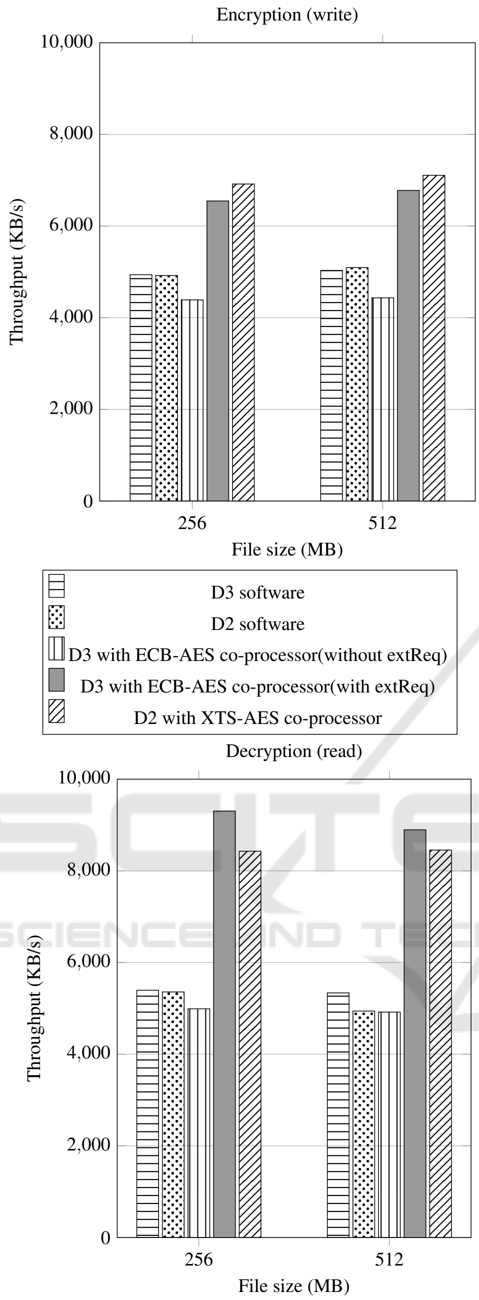

Figure 5: IOZONE benchmark, encryption and decryption throughputs for all configurations.

design of future boards will be updated accordingly to enable faster XTS-AES and FDE operations with the proposed *extReq* optimization.

The interested reader is invited to refer to the full paper: https://hal.archives-ouvertes.fr/hal-02555457

REFERENCES

ATMEL (2017a). SAMA5D2 board. http://www.atmel.com/tools/ATSAMA5D2-XULT.aspx.

ATMEL (2017b). SAMA5D3 board. http://www.atmel.com/tools/ATSAMA5D3-XPLD.aspx.

Brož, M., Kozina, O., Wagner, A., Meurer, J., and Virgovic, M. (2020). dm-crypt: Linux kernel device-mapper crypto target. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt.

Fruhwirth, C. (2005). Hard disk encryption with dm-crypt, luks, and cryptsetup. *Linux Magazine*, 61:65–71.

Fruhwirth, C. (2018). Luks on-disk format specification version 1.2.3. Technical report. https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf.

Gueron, S. (2012). Intel advanced encryption standard instructions set - rev 3.01. Technical report. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf.

IEEE Computer Society (2008). IEEE standard for cryptographic protection of data on block-oriented storage devices. *IEEE std 1619-2007*.

LLC, P. I. (2010). The billion dollar lost laptop problem: benchmark study of U.S. organizations. http://www.intel.com/content/dam/doc/white-paper/enterprise-security-the-billion-dollar-lost-laptop-problem-paper.pdf.

Martin, L. (2010). XTS: A mode of AES for encrypting hard disks. *IEEE Security & Privacy*, 8(3):68–69.

Norcott, W. D. and Capps, D. (2003). Iozone filesystem benchmark. http://www.iozone.org/.