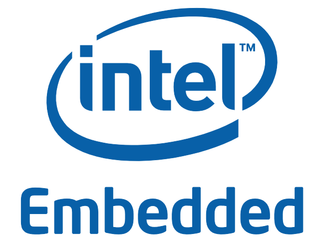


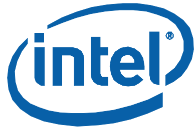
White Paper

Adrian Hoban
Pierre Laurent
Ian Betts
Maryam Tahhan
Intel Corporation

Unleashing Linux*-Based Secure Storage Performance with Intel® AES New Instructions

February 2013

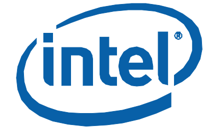
328679-001



Executive Summary

AES-XTS [1] is a popular cipher for use in secure storage. In Linux*, the storage subsystem leverages the crypto subsystem to access the cipher suites. Such an implementation limits the incredible performance in Intel® Architecture processors equipped with Intel® AES New Instructions (Intel® AES-NI) [6].

This paper introduces some of the innate characteristics of Intel AES-NI, and demonstrates how an alternative approach to integrating cipher algorithms optimized for Intel AES-NI can deliver an up to 6x performance gain in the storage subsystem. Furthermore, such gains are demonstrable with standard storage tools with sector sizes configured at commonly deployed levels.



Contents

Introduction 16

What is AES-XTS? 22
- AES 23
- XTS 30

Block Storage Encryption in Linux 41
- *The bio* Structure 50

Inefficiencies using Linux Kernel Crypto Framework with dm-crypt 60
- Async vs. Sync APIs 62

Integrating Optimized AES-XTS with dm-crypt 71

Optimization of AES-XTS 80
- Exploiting Parallelism 81
- Serial vs. Parallel Code 89
- Implementation in C 95

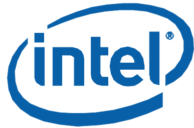
Testing 113
- Accuracy and Performance Testing 114
- Integration Testing 118

Results 127
- Legal Disclaimer 128
- AES-XTS Cipher Throughput 136
- Benchmarking Results - Encrypted RAM Disk 145
- Benchmarking Results - Encrypted SSD 152
- Benchmarking Results - Virtual Machine 159

Challenges, Lessons, and Next Steps 164

Conclusions 178

References 184



Introduction

To address the increasing requirement to protect sensitive personal or financial information, the Advanced Encryption Standard (AES) has been adopted by the U.S. National Institute of Standards as part of Federal Information Processing Standard (FIPS) 197 [2], and is now widely used.

Since 2010, the Intel® Core™ processor family has included a set of Intel® Advanced Encryption Standard (AES) New Instructions (NI)[6] that help to accelerate the execution of the AES algorithms, achieving a performance improvement of up to 10x over earlier software implementations.

A close inspection of the Linux kernel [4] software stack shows that the Linux kernel serializes all requests, splits the data structure relating to ongoing I/O operations and buffers into sectors, allocates and frees descriptors on a per-sector basis, and consequently loses the potential advantages of the flush logic from the cache subsystem where many requests are available at the same time for the same device.

By providing a better hook for parallel encryption of multiple buffers, software encryption on any logical drive can be further optimized.

This paper shows that a highly optimized AES-XTS implementation — able to process up to eight data units in parallel — can be integrated with dm-crypt to unleash significant performance gain.

What is AES-XTS?

AES

AES is a symmetric block cipher that encrypts/decrypts 128-bit data blocks, using 128-, 192-, or 256-bit cipher keys. The cipher performs iterative transformations on the input data block over 10, 12, or 14 rounds, depending on key size.

The 128-, 192-, or 256-bit keys are expanded to yield a schedule of unique 128-bit keys, one for each round (also known as *round keys*). The 128-, 192-, or 256-bit cipher key is diffused across the key schedule.

Each 128-bit data block is treated as a 4x4 byte state. The AES algorithm achieves three fundamental cryptographic objectives: *confusion, diffusion, and secrecy* by applying transformations (Substitute Bytes, Shift Rows, Mix Columns, and Add Round Keys) to the state.

The Intel® AES-NI includes six new instructions that help accelerate the iterative crypt transformations performed during the AES rounds, and also the key expansion.

The AES algorithm is well documented in many published sources, as are details about Intel AES-NI (see [2] and [6] for more information).

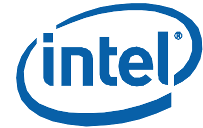
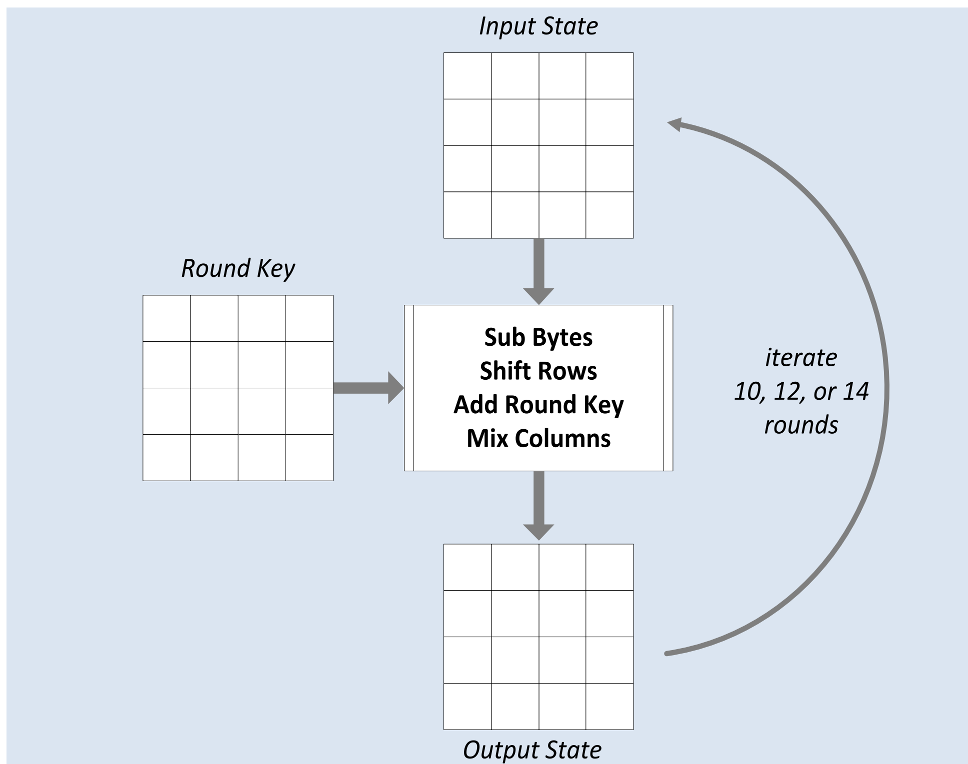


Figure 1. AES Encrypt



XTS

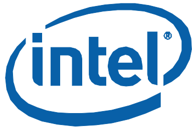
AES-XTS builds on AES to provide for encryption of data units that are arbitrary subdivisions of some storage object; for example, sectors on a disk partition (or parts of a file).

For ease of explanation this discussion focuses on the case of AES-XTS-based whole disk encryption, where a data unit equates to a disk sector.

From a user perspective, storage is protected by a single (256- or 512-bit) key; however, if the same key were to be used for every sector, this would leave the encrypted storage vulnerable to a "known plain text" attack.

AES-XTS solves the problem by "tweaking" the initialization vector for successive data units. The tweak is derived from the data unit sequence, i.e., the sector number. As a consequence, encryption of sectors containing the same plain text results in a different cipher text. Further, because the tweak value can be derived, it is not necessary to store it, thus avoiding storage expansion.

The key is split into two parts denoted key-1 and key-2, and each data unit is encrypted in two steps: The first step is to encrypt the tweak, the second step is to iterate over the data unit, *in 16-byte blocks*, performing an "XOR – Encrypt – XOR" transformation. In the case of whole disk encryption, the data unit size is the disk sector size, typically 512 bytes (32 AES blocks).



Step one
Key-2 is used to encrypt the tweak (i.e., the sector number) using the AES algorithm from Figure 1.

Step two
The result of step one is transformed by a Galois modular multiplication involving the AES block number, denoted "J", to yield an intermediate value "T", which is unique per AES block.

After this, the input plain-text of AES block[j] is XOR'd with T to yield an intermediate result PP.

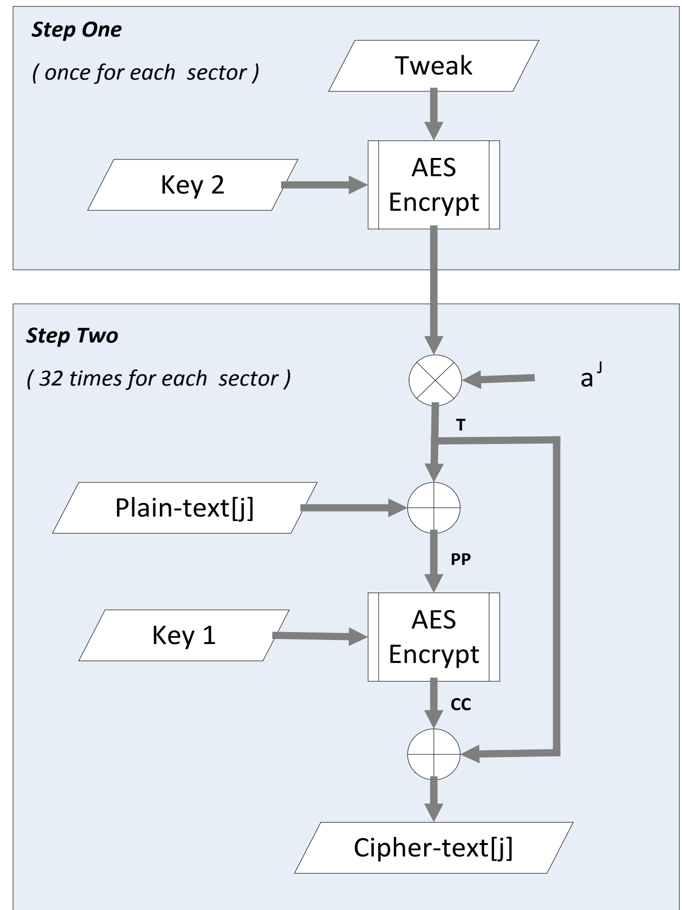
Next, key-1 is used to Encrypt PP, again using the AES algorithm from Figure 1.

Finally, the result of this step is once again XOR'd with T.

Step two is repeated for each of the 32 AES blocks in the 512-byte sector.

Decryption follows the same approach, but the AES decrypt operation is substituted in step two.

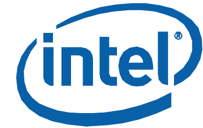
Figure 2. AES-XTS



Block Storage Encryption in Linux

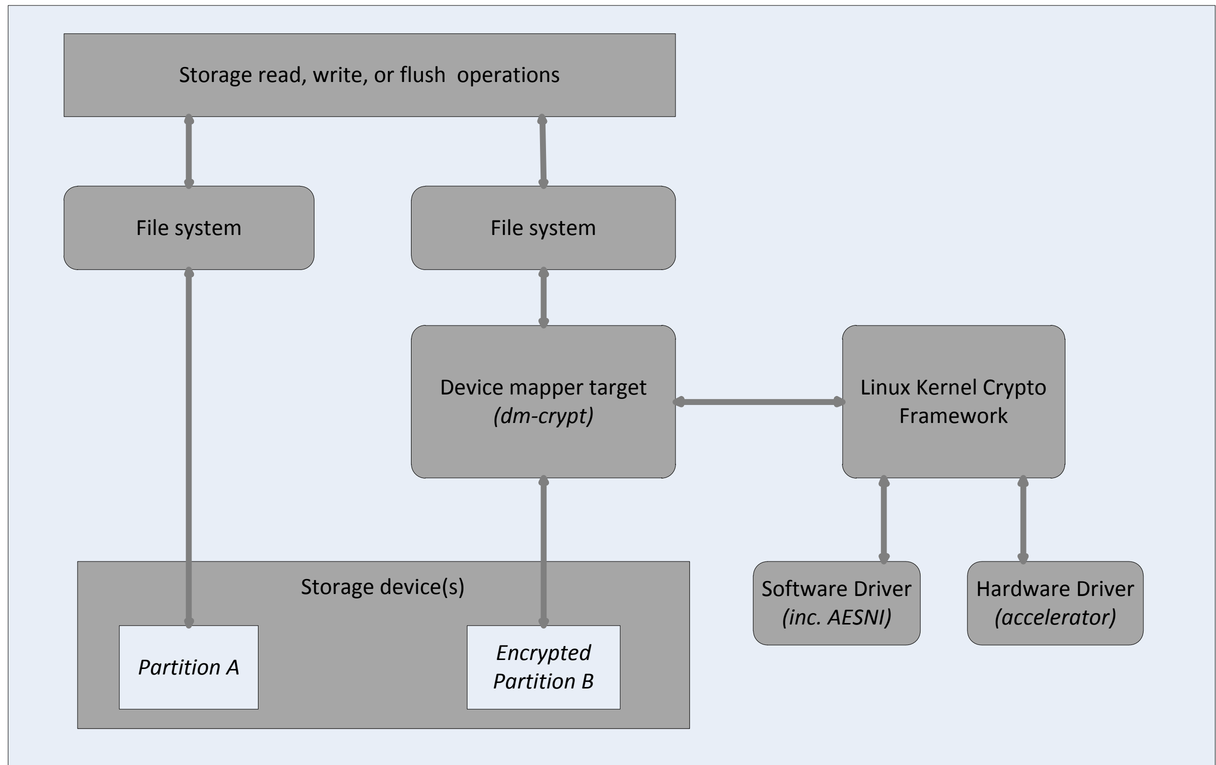
The Linux storage subsystem makes use of a framework for stacking virtual block device filter drivers (also known as targets) in order to realize features like the logical Volume Manager (LVM), software RAID, disk encryption, etc. This framework is called a device mapper (DM) and is a virtual block device driver itself. The device mapper resides at the generic block layer which handles requests for all block devices in the system.

dm-crypt is the DM target responsible for disk encryption. The generic block layer initiates data transfers by starting one or more I/O operations, each of which is packed into a *bio* structure. The *bio* structures relating to any read or write requests that are directed at an encrypted block device are passed to dm-crypt. dm-crypt partitions the data into smaller units and issues multiple crypto requests (one per sector) to the Linux Kernel Crypto Framework (LKCF).



The LKCF has been designed to take advantage of both off-chip hardware accelerators and to provide software implementations when accelerators are not available. The LKCF supports both a legacy software implementation of AES-XTS and a software version that is accelerated using Intel AES-NI.

Figure 3. Linux Device Mapper Target dm-crypt



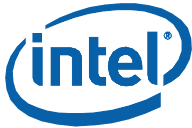
The current implementation of dm-crypt serializes all requests, splits the data referenced by *bio_vec* structures into individual sectors, and allocates and frees descriptors on a per-sector basis. Therefore the current implementation does not maximize the potential advantages of the flush logic from the cache subsystem where many requests are available at the same time for the same device.

The bio Structure

A *bio* is the fundamental data structure for the generic block layer. It conveys the read or write request data (e.g., disk sectors) between DM targets and ultimately to a physical device driver.

In order to support features such as LVM and RAID, the device mapper (DM) manipulates a *bio* by performing the following operations:

- Cloning - a near-identical copy of the *bio* that is sent to the DM device is created. The completion handler for the cloned *bio* points to the DM's completion handler (which differs from that of the original *bio*).

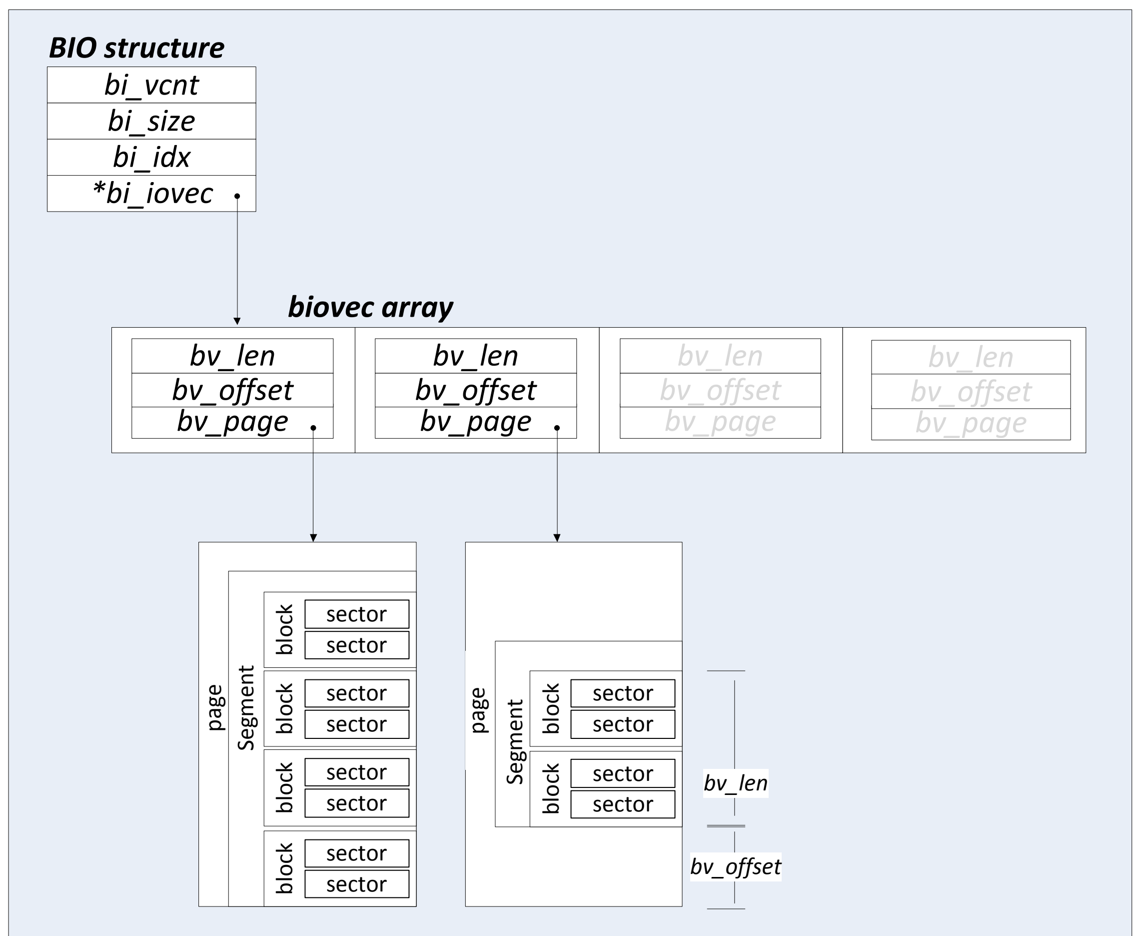


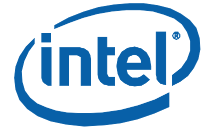
- Mapping - the cloned *bio* is passed to the target device where its fields are modified. The bi_dev field specifies the device to send the *bio* to. The cloned *bio* is submitted to that device.
- Completion – the DM's completion handler calls the target specific completion handler where the *bio* can be remapped or completed. The original *bio* is completed when all clones have completed.

Note: These operations are also exploited by the asynchronous functionality of dm-crypt.

Each bio references the data involved in an I/O operation using an array of bio_vec structures. A bio_vec essentially consists of: a page, offset, length tuple. Each bio_vec points to a 4K page, which can contain up to eight sectors. Sectors are always contiguous within the same page; consecutive vectors are not necessarily contiguous.

Figure 4. Linux *bio* Structure





Inefficiencies using Linux Kernel Crypto Framework with dm-crypt

Async vs. Sync APIs

There are two fundamentally different approaches to leveraging parallel processing resources. Comparing and contrasting them as follows can help explain the opportunity to optimize dm-crypt.

An **Asynchronous** model assumes work is partitioned across multiple CPUs/accelerators. Crypto requests are partitioned and responses are queued and aggregated. The request handling, crypto processing, and aggregation and return of results happen in different execution contexts.

A **Synchronous** model processes requests immediately and achieves parallelization of work by taking advantage of the CPU pipeline.

While an Asynchronous design can bring more resources to bear, the overhead of context switching and scheduling must be amortized and may be disproportionate for small request sizes.

The standard Linux disk encryption is based on an asynchronous model; however, it has been demonstrated that pipelining multiple 512-byte sectors using a synchronous design is a much more efficient approach.

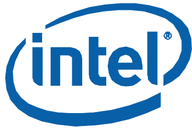
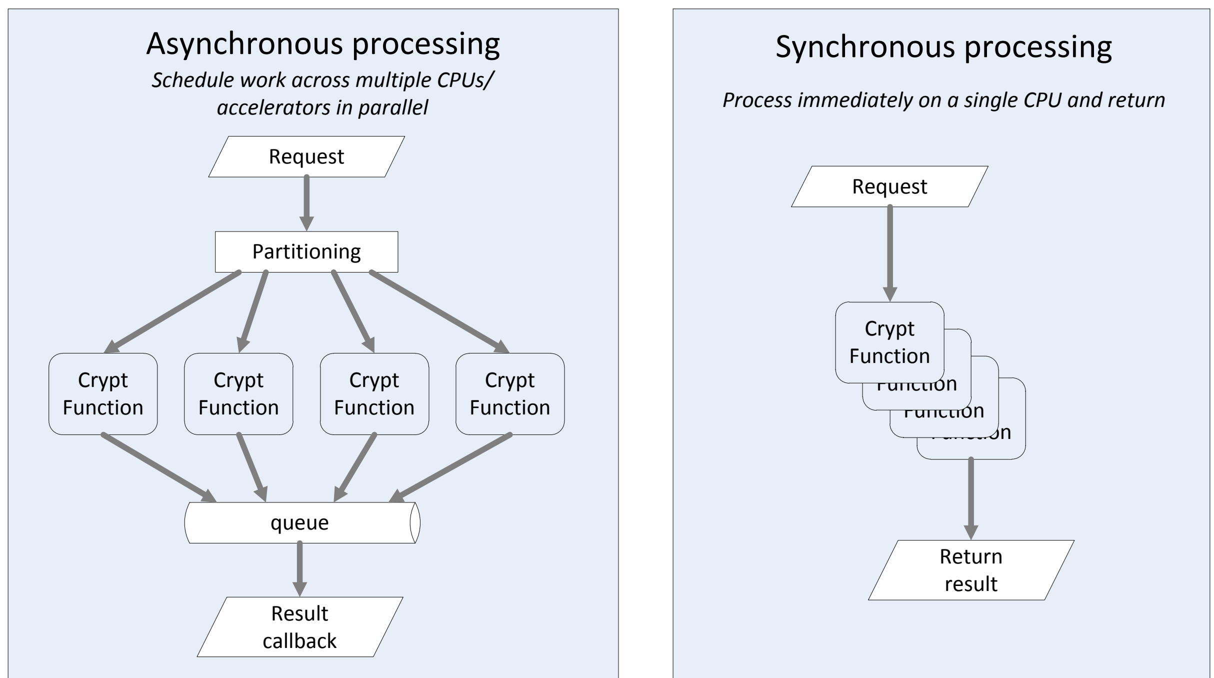


Figure 5. Synchronous vs. Asynchronous APIs



Integrating Optimized AES-XTS with dm-crypt

By introducing a synchronous processing model it is possible to exploit the native crypto instructions of the host CPU and completely eliminate the accelerator driver path.

This is achieved by implementing a new synchronous AES-XTS subsystem within dm-crypt; I/O requests bypass the local work queues and are processed immediately.

This new implementation of dm-crypt simply iterates over the pages referenced by each *bio_vec* within a *bio*, directly processing up to eight sectors at a time.

The experimental implementation used a compile time option ("CONFIG_DM_CRYPT_FORCE_XTS_LKCF_BYPASS") selectable in the kernel configuration menu.

When defined, the cipher configured for ALL storage encryption configurations is assumed to be AES-XTS, and the asynchronous interface to LKCF is bypassed.

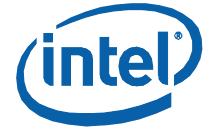
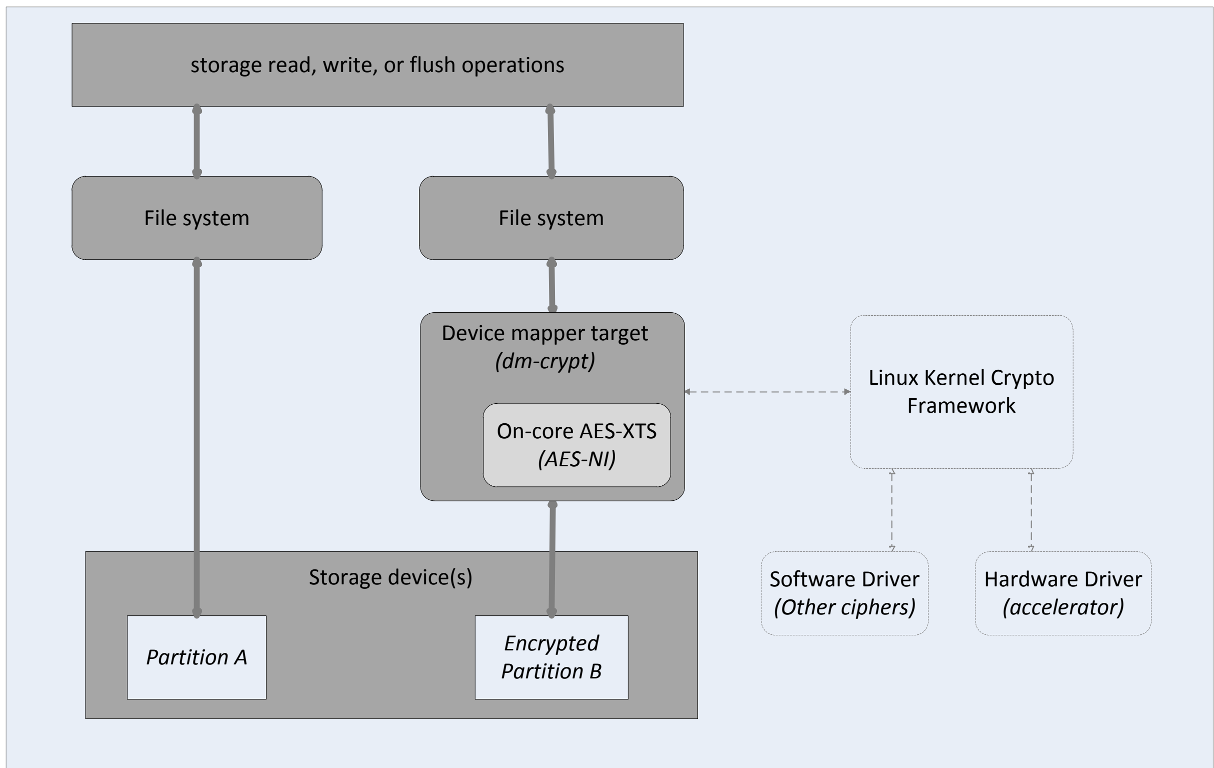


Figure 6. Integration of an Optimized AES-XTS with dm-crypt



Optimization of AES-XTS

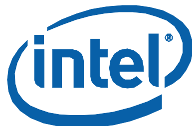
Exploiting Parallelism

The synchronous processing model relies on the possibility of exploiting data- and instruction-level parallelism to maximize use of the processor execution units and pipeline.

To benefit from data parallelism, programs must be explicitly crafted; for example, using Single Instruction Multiple Data (SIMD) instructions to exploit parallelism that may be inherent in a particular algorithm.

Architectural features, including superscalar execution, pipelines, out of order execution, branch prediction and speculative execution, are designed to maximize instruction-level parallelism.

With instruction-level parallelism the CPU hardware strives to detect and exploit opportunities dynamically; however, good program design can further help avoid hazards that reduce utilization.



The most significant penalties are due to data dependencies and branch misprediction. An ideal code stream would exhibit no data dependency stalls, no branch misprediction, and would saturate the different execution units perfectly.

Performance can be significantly improved by applying 'block factoring' techniques to optimize for the memory hierarchy (the register pool, Level 1, and next-level caches, and so on) by maximizing placement and reuse of data to achieve the highest possible hardware throughput.

Optimization thus strives to keep the pipeline full, minimize latencies, and maximize the use of multiple execution units.

Serial vs. Parallel Code

As shown in Figure 7, pseudo code processes multiple flows correctly, but fails to achieve any of the benefits alluded to in the preceding discussion.

The outer loop processes the flows sequentially, and thus misses significant opportunity to benefit from instruction-level parallelism possible with multiple AES units.

The round keys have to be fetched repeatedly from memory, taking no advantage of the memory hierarchy.

```
Serial code example...

For each data flow
        For each block in flow
                AESENC round 0
                For round 1- N
                        AESENC round N
                End
                AESENCLAST last round
        End
End
```

Figure 7. Serial Code

By contrast, the parallel code example in Figure 8 shows AES rounds processed for each flow in turn, and the process is repeated for each AES block.

This generates a stream of independent instructions that can be efficiently pipelined.

The current round key is reused for each flow, taking advantage of the memory hierarchy.

A further potential optimization is to unroll the inner loops to minimize potential stalls due to branching.

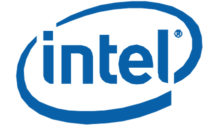
```
Parallel code example ...

For each block in flow
        For each data flow
                AESENC round 0
        End
        For round = 1 to N
                For each data flow
                        AESENC round N
                End
        End
        For each data flow
                AESENCLAST last round
        End
End
```

Figure 8. Parallel Code

Implementation in C

The XTS algorithm can be readily implemented in C programming language using compiler intrinsic functions to invoke the AES-NI operations directly.

In a C language implementation using intrinsic functions, the 128-bit AES state variables and round keys are stored in variables of type __m128i. If possible, both



the GNU Compiler Collection (GCC) and the Intel C/C++ Compiler (ICC) compiler will maintain these as XMM registers.

The machine code fragment in Figure 9 illustrates the level of optimization possible in C. As shown, the compiler was able to detect the advantage of unrolling the inner loops. Further optimization is possible by coding in assembly language to interleave independent instruction sequences at the beginning and/or end of the unrolled round loop.

Figure 9. Implementation in C

Resulting machine code

Loops unrolled to process 8 flows at a time
State in registers xmm0 to xmm7
First round key in xmm8

Example C code fragment for AES loop

```
/* round  1 to N */
for (round = 1; round < AES128_ROUNDS; round += 1) {
   k1 = pkey1[round];
   for (flow = 0; flow < nb_flows; flow++) {
      state[flow]=_mm_aesenc_si128(state[flow], k1);
   }
}

/* last round  */
k1 = pkey1[round];
for (flow = 0; flow < nb_flows; flow +=1 ) {
   state[flow]=_mm_aesenclast_si128(state[flow], k1);
}
```

```
                                        /* first round */
                                        aesenc %xmm8,%xmm7
                                        aesenc %xmm8,%xmm6
                                        aesenc %xmm8,%xmm5
                                        aesenc %xmm8,%xmm4
                                        aesenc %xmm8,%xmm3
                                        aesenc %xmm8,%xmm2
                                        aesenc %xmm8,%xmm1
                                        aesenc %xmm8,%xmm0
                                        /* second round */
                                        movdqa 0x20(%rax),%xmm8
                                        aesenc %xmm8,%xmm7
                                        aesenc %xmm8,%xmm6
                                        aesenc %xmm8,%xmm5
                                        aesenc %xmm8,%xmm4
                                        aesenc %xmm8,%xmm3
                                        aesenc %xmm8,%xmm2
                                        aesenc %xmm8,%xmm1
                                        aesenc %xmm8,%xmm0
                                        /* third round */
21b7:     66 44 0f 6f 40 30             movdqa 0x30(%rax),%xmm8
21bd:     66 41 0f 38 dc f8             aesenc %xmm8,%xmm7
21c3:     66 41 0f 38 dc f0             aesenc %xmm8,%xmm6
21c9:     66 41 0f 38 dc e8             aesenc %xmm8,%xmm5
```

Pattern repeats for N rounds……

```
                                        /* Last round */
2337:     66 44 0f 6f 80 a0 00          movdqa 0xa0(%rax),%xmm8
233e:     00 00
2340:     66 41 0f 38 dd f8             aesenclast %xmm8,%xmm7
2346:     66 41 0f 38 dd f0             aesenclast %xmm8,%xmm6
234c:     66 41 0f 38 dd e8             aesenclast %xmm8,%xmm5
```

And so on…

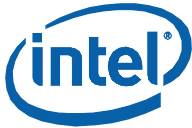
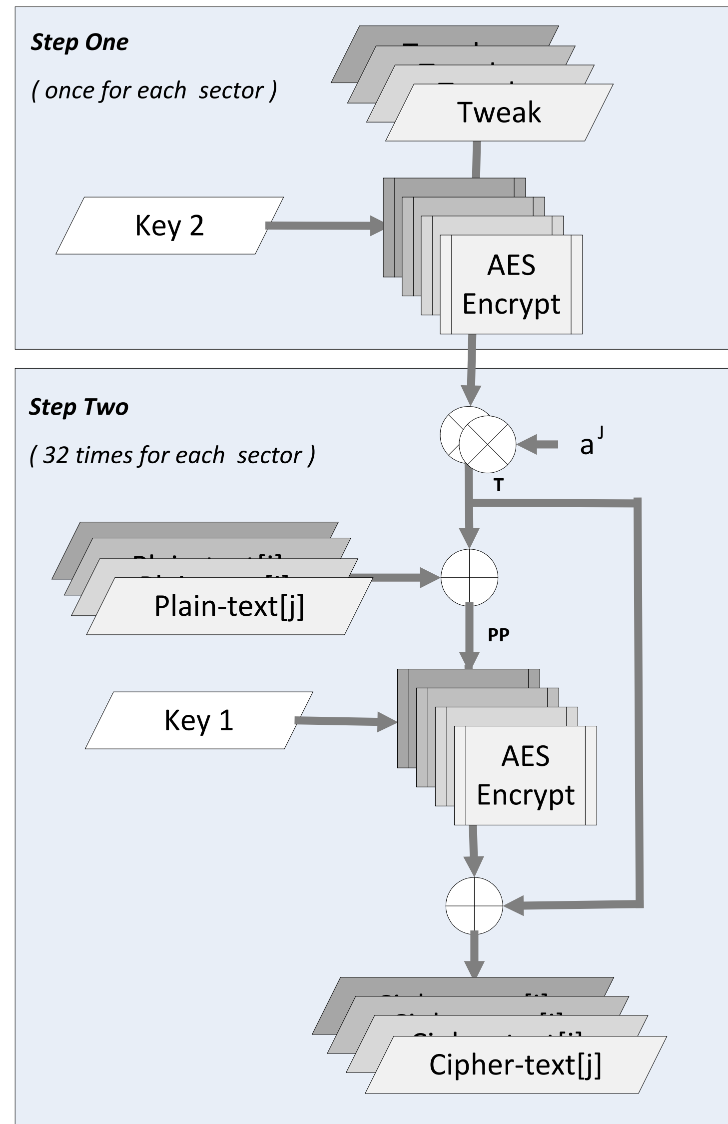


Figure 10. Opportunities for Parallelism in AES-XTS



In summary, the XTS algorithm exhibits opportunities to exploit both data- and instruction-level parallelism.

Instruction-level parallelism

By starting the first (and subsequent) AES rounds for each of multiple disk sectors at the same time, the AES execution units can be saturated.

Because the data flows are independent, there are no data-dependency hazards and the AES encrypt instructions can be pipelined optimally.

This opportunity exists in step 1 when the tweak is encrypted, and in step 2 when iterating over the sector data.

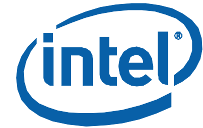
Above a maximum optimal number of flows, results are likely to be poor due to register pressure; below a minimum optimal number it may not be possible to keep the pipeline full, resulting in underutilization of the execution units.

The optimal number of flows is influenced by the pipeline and execution logic design, and by the number of registers available, and is thus architecture-dependent.

Flows can be synthesized if necessary. By dividing a sector into parts, and pre-computing the respective starting values for T, each part can be processed in parallel.

Data-level parallelism

There is also a limited opportunity to gain from data-level parallelism by using SIMD in the calculation of the intermediate variable T. This enables T values to be calculated two at a time.



Testing

Accuracy and Performance Testing

All of the tuning work on the core algorithm was carried out in user space.

Correctness of the implementation was ensured by verification against published test vectors.

Developing and testing initially as a simple user-space application facilitated debug and optimization.

By repeatedly executing hundreds of thousands of crypt operations and time stamping before and after execution it was possible to calculate the performance in terms of CPU clock cycles per byte of encrypted or decrypted data.

The results were calculated and displayed for different numbers of flows. Using these results it was possible to quickly observe the consequences of different optimization strategies.

Integration Testing

The primary objective of integration testing was to confirm the performance benefit predicted by moving to a highly optimized synchronous implementation.

Initially an encrypted RAM disk was used as a sandbox mechanism, which enabled a safe and convenient way to test without risk of destroying critical operating system disk partitions.

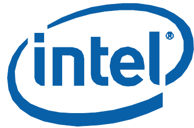
Once stable, the testing was extended to include an encrypted partition on a physical disk drive (for example, a solid state drive (SSD)).

While physical drive technology is increasingly able to integrate encryption functions in hardware, a particularly interesting use case for software-based encryption is a virtualized environment where guests may require complete control over encryption of their own partitions. This use case was validated by testing a guest operating system running on a virtual machine.

For the results to be meaningful, a representative storage workload or benchmark was required. IOzone* [3] is an established and open file system benchmark tool that measures I/O performance for a variety of file operations.

IOzone was used to measure performance when encrypting RAM disk and SSD partitions. Using IOzone, the performance of the stock Linux asynchronous version of dm-crypt (that relies on LKCF) was benchmarked and contrasted with a synchronous version of dm-crypt that integrates a highly optimized AES-XTS implementation using Intel AES-NI..

To show the difference in performance relative to the baseline Linux dm-crypt implementation, the results in the following sections are expressed as a ratio of the IOzone results for the new versus legacy Linux implementations (new result divided by legacy Linux result).



Thus a ratio or "improvement factor" of 1 actually means no improvement; a factor of 2 means 100% improvement and so on. A factor less than 1 means a degradation in performance.

The results are given for a representative subset of IOzone test cases: random read test, and random write test.

Results

Legal Disclaimer

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel® AES-NI requires a computer system with an AES-NI enabled processor, as well as non-Intel software to execute the instructions in the correct sequence. AES-NI is available on select Intel® processors. For availability, consult your reseller or system manufacturer. For more information, see Intel® Advanced Encryption Standard Instructions (AES-NI).

Configurations

Hardware

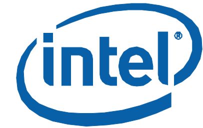
Product name: S2600GZ
Dual Intel® Xeon® CPU E5-2680 @ 2.70GHz
L3 cache 20,480KB
Total CPU cores: 16 (8 x2)
Total system memory: 16GB (8 x 2G 1333MHz synchronous DIMM)
Disk drive: 160GB SSD; Intel SSDSA2CW160G3; firmware revision 4PC10362

Software

BIOS: Intel version SE5C600.86B.99.99.x040.111920110024, release date 11/19/2011
Operating system: Fedora* 16, Linux kernel 3.6.3 (http://www.kernel.org/)
Hypervisor: Xen* version 4.1.3
Compiler: GCC 4.6.3
IOzone*: http://www.iozone.org/, version 3.408, 64-bit.

Tests were performed at Intel Communications and Storage Infrastructure Group labs in Shannon, Ireland.

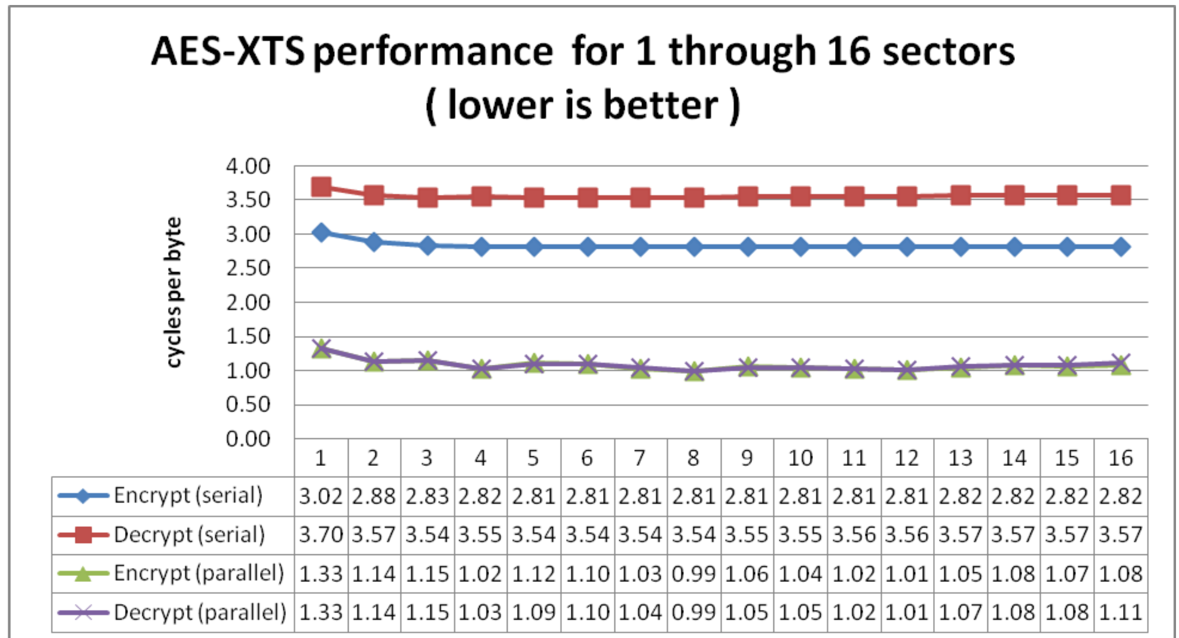
For more information, see http://www.intel.com/performance



AES-XTS Cipher Throughput

The results below contrast the performance of a simple serial implementation and a highly optimized parallel implementation based on the ideas illustrated on page 106.

Figure 11. AES-XTS Performance



The results demonstrate a significant performance improvement — greater than 80% in most cases — for the parallel over the serial implementation.

Figure 12. Relationship Between Flows and Sectors

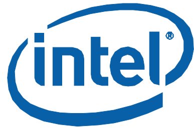
# sectors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# flows	8	8	6	8	10	6	7	8	9	10	11	12	13	14	15	16
# blocks per flow	4	8	16	16	16	32	32	32	32	32	32	32	32	32	32	32

As shown in Figure 12, when the number of sectors per request is five or less, the parallel implementation synthesizes flows by splitting each sector into multiple sub-flows.

This partitioning requires pre-computation of intermediate T values and results in smaller flow sizes, consequently yielding slightly worse performance.

The best results are for 7 sectors and up to and below 16 sectors. In these cases there are free XMM registers, enabling the round key to be maintained in a register.

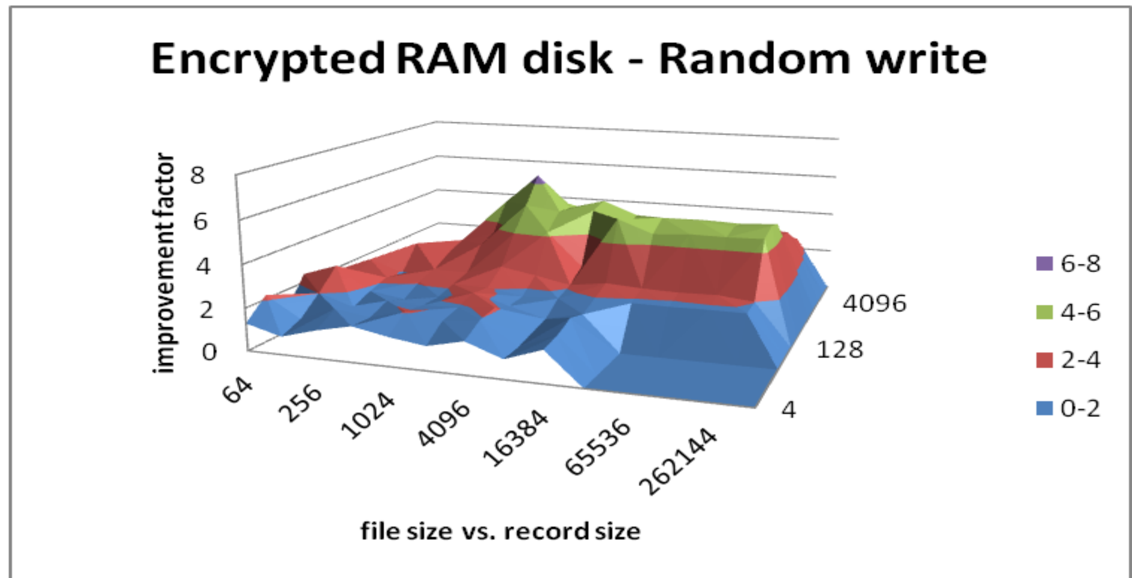
In the case of 16 flows, the compiler is able to maintain the AES state in XMM registers 0 to 15 while indirectly accessing the round keys.



Extending this implementation beyond 16 sectors yields poor results due to register pressure, and an alternative optimization strategy should be sought.

Benchmarking Results - Encrypted RAM Disk

Figure 13. Encrypted RAM Disk – Random Write



The results in Figure 13 indicate a peak performance improvement factor in the case of a random write test of greater than 6x, with almost all measurements exceeding a factor of 2 in this test case. Other write cases exhibit a very similar pattern.

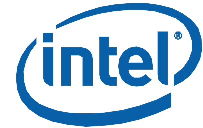
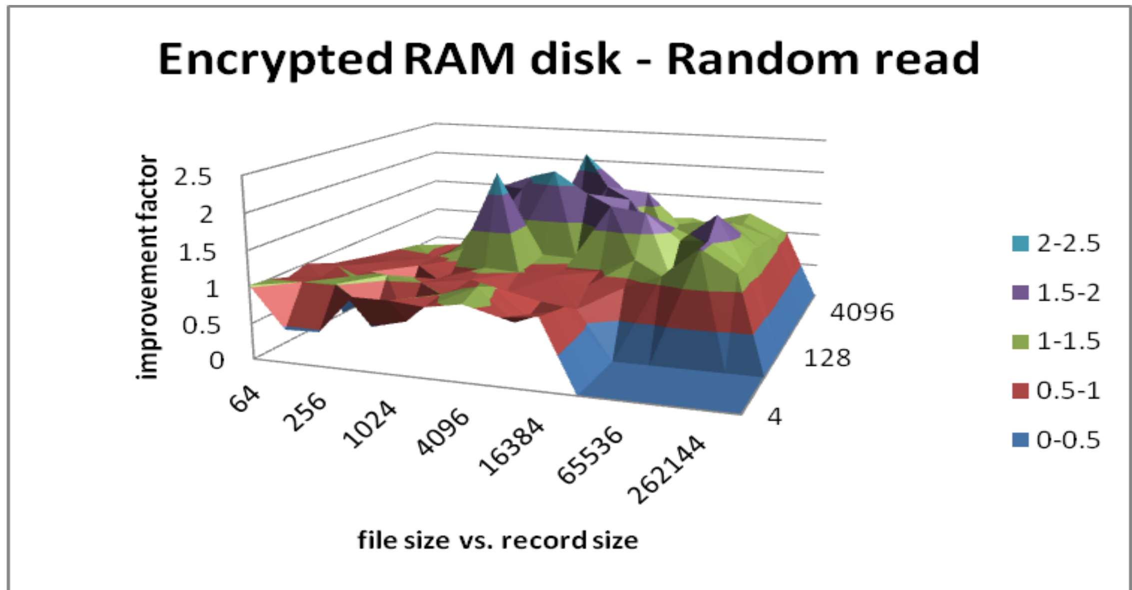
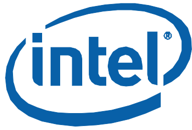


Figure 14. Encrypted RAM Disk – Random Read



The apparently poor results obtained in read test cases are influenced by two factors: the relatively small sizes of read requests made to dm-crypt (a write request *bio* can often contain as many as 16 pages, whereas read requests are never greater than 1 page), and the impact of Linux disk caching masking the relative performance gain (during a benchmark run, there are approximately 1000 times more write requests than read requests, implying that already decrypted read requests are being serviced by the Linux disk cache).



Benchmarking Results - Encrypted SSD

Figure 15. Encrypted SSD – Random Write

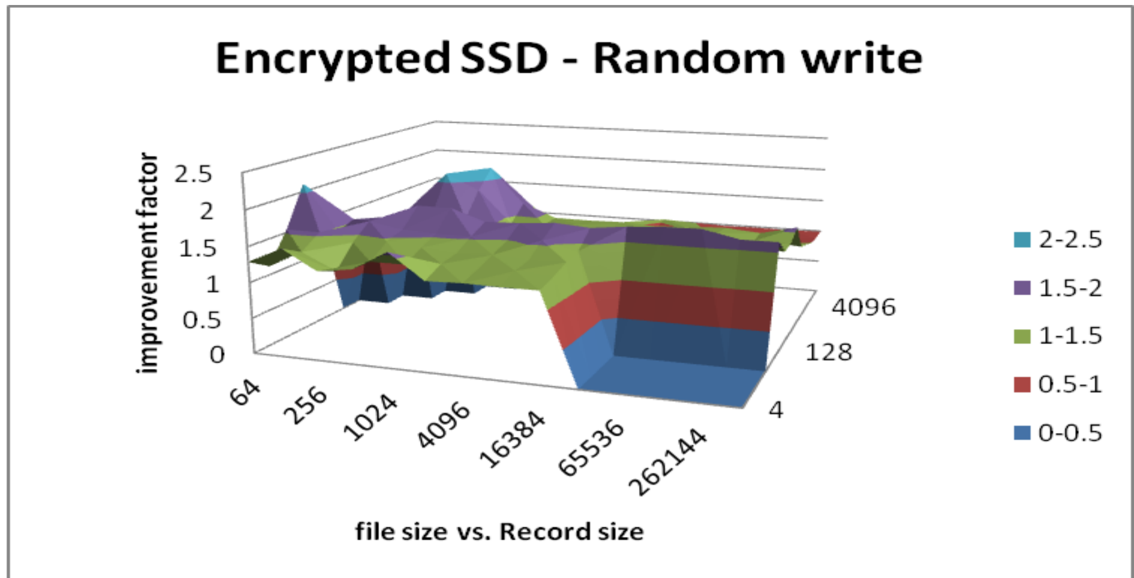
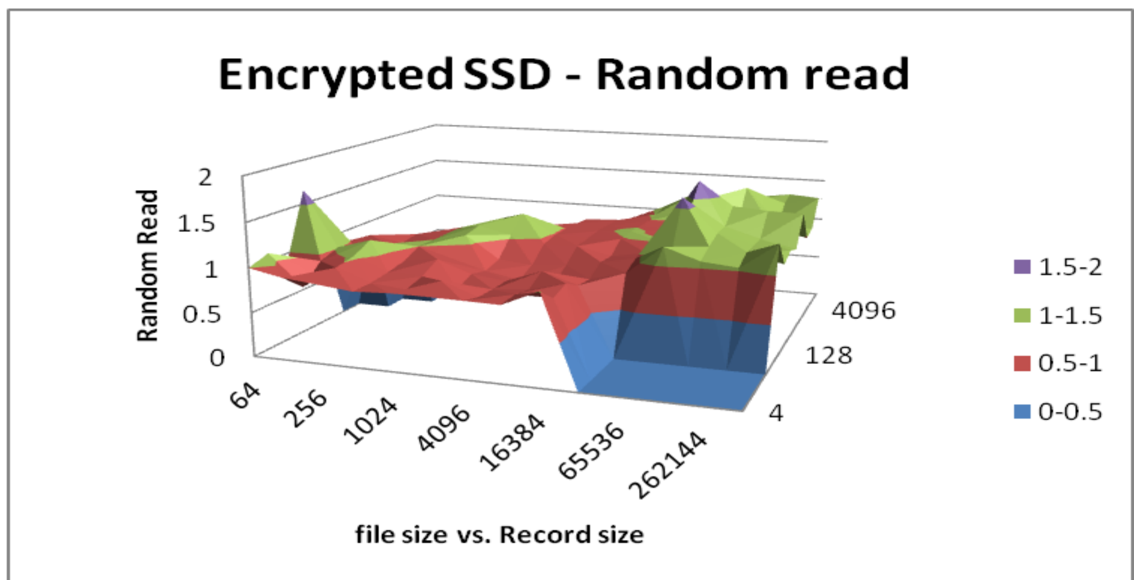
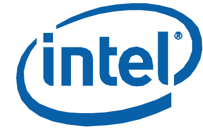


Figure 16. Encrypted SSD – Random Read



The results for encrypted SSD tests indicate a peak performance improvement factor for random write tests of greater than 2x, with the majority of measurements exceeding a factor of 1.5x. Other write cases exhibit a very similar pattern.

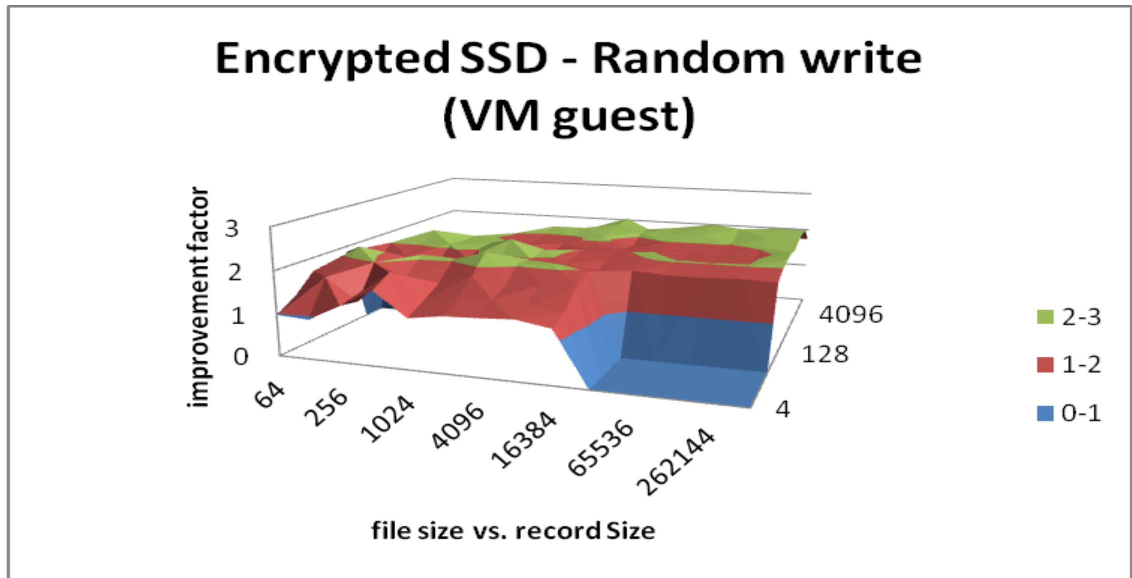


Contrasting the SSD results with the RAM disk results for the write test indicates that much of throughput benefit obtained over the stock implementation is masked by the increase in IO latency when a physical disk device is accessed.

As with the RAM disk results, the poorest results are obtained in read test cases, with random read yielding the lowest improvement factor. The majority of measurements in read test cases yield an improvement factor close to 1, i.e., no improvement.

Benchmarking Results - Virtual Machine

Figure 17. Encrypted SSD – Random Write (VM Guest)



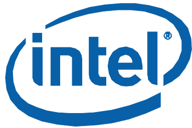
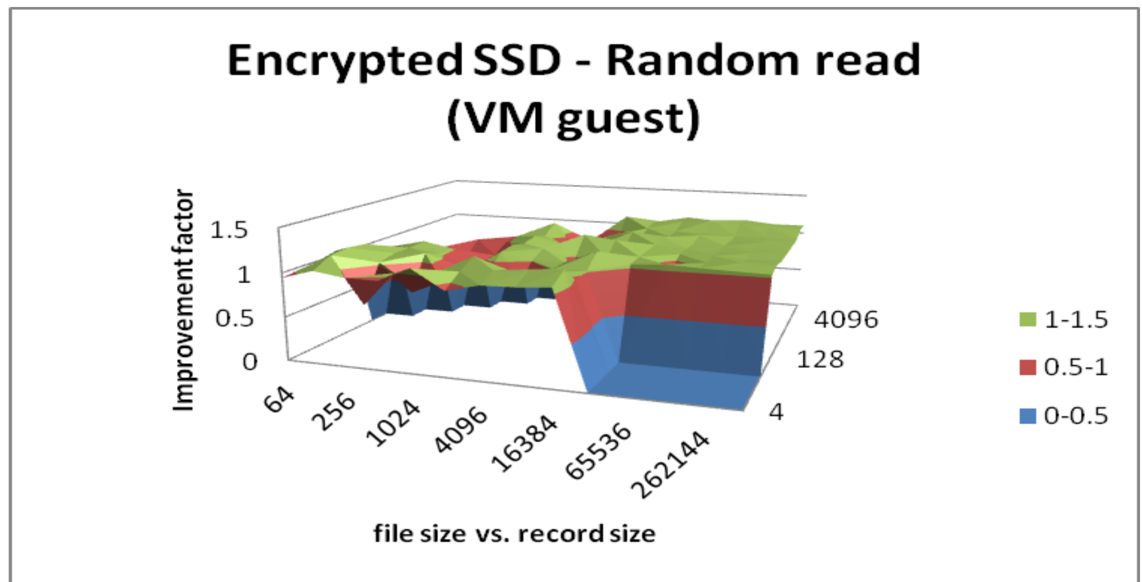
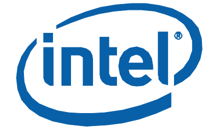


Figure 18. Encrypted SSD – Random Read (VM Guest)



The results above show the performance for an encrypted SSD partition accessed as a raw device by a virtual machine. As can be seen, the results are not significantly different from a non-virtualized case.



Challenges, Lessons, and Next Steps

Locking and Cloning

In the original dm-crypt code, the processing sequence was as follows: partition *bio*, spawn requests to LKCF sub-system, increment semaphore, decrement semaphore as requests are completed.

In the synchronous implementation, the whole *bio* is processed immediately. The semaphore is taken at the start and released at the end.

Cloning of a *bio* was not eliminated; however, with a synchronous API it may be possible to do so. This is a potential area for further investigation.

Alignment

To align variables of type __m128i correctly, the stack must be 16-byte-aligned.

Unfortunately the kernel Application Binary Interface (ABI) enforces only 8-byte-stack alignment. A possible solution (the one we adopted) may involve a pre-amble and post clean-up code to re-align the stack.

Trust the Compiler

Most third-party implementations resort to assembly language; however, for ease of development, this implementation was crafted in C.

Much effort spent hand-optimizing the C code proved unnecessary. Experience showed that both the GCC and the Intel ICC compiler could optimize the algorithm effectively.

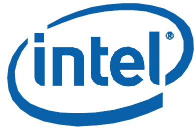
Examining the machine code output of the compiler is recommended. Also, using performance-profiling tools such as Intel® vTune™ (part of the Intel® Parallel Composer tool suite) are recommended when embarking on heavy optimizations.

While the C implementation has yielded respectable results, further optimization using assembly language is certainly possible.

Platform Differences

The processor architecture (differences in cache size, pipeline, and out of order logic behavior) has a great influence over performance and consequently the necessary approach to optimization and tuning.

A production-ready implementation would need to be adapted for the particular target.



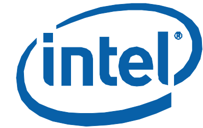
Virtual Machines

Performance in a virtualized guest environment proved comparable to host performance.

Note: Not all currently available hypervisors allow AES-NI support for guests, even when it is physically supported on the host CPU.

Read Performance

The potential benefit for read performance is not being realized due to a combination of small request sizes and disk caching. It may be possible to optimize Linux storage subsystems for read performance. This is a potential area for further investigation.



Conclusions

Intel® AES New Instructions provide an on-core crypto capability built into the instruction set architecture.

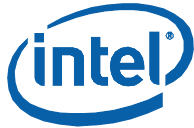
The accelerator driver model of the current kernel implementation is not able to take full advantage of this evolution.

By bypassing the legacy Linux crypto subsystem, using a synchronous API, an up to 6x performance gain may be realized.

By leveraging superscalar execution, deep pipelines, out of order execution, branch prediction and speculative execution logic, processor utilization can be maximized to exploit inherent parallelism in disk storage read, write, and flush operations.

By carefully considering the opportunities for parallelism inherent in the encryption algorithm and afforded by the processor architecture, it has been demonstrated that a well-crafted but relatively simple C implementation can achieve significant performance improvement, and further optimization may be possible using assembly language.

Performance increases for encrypted SDD drives of up to 2x for write operations (up to 6x for RAM disks) represent a huge potential system performance improvement, not only in terms of accelerated storage access, but also due to improved CPU utilization.



References

1. IEEE P1619™/D16 Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices
   http://grouper.ieee.org/groups/1619/email/pdf00086.pdf

2. Federal Information Processing Standards Publication 197
   Advanced Encryption Standard (AES)
   http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

3. IOzone* Filesystem Benchmark
   http://iozone.org/

4. Understanding the Linux Kernel, 3rd Edition. ISBN: 0596005652

5. Intel® 64 and IA-32 Architectures Software Developer's Manual
   http://download.intel.com/products/processor/manual/325462.pdf

6. Intel® Advanced Encryption Standard (AES) New Instructions Set
   http://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf

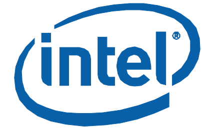
7. Intel® Architecture Instruction Set Extensions Programming Reference
   http://software.intel.com/sites/default/files/m/a/b/3/4/d/41604-319433-012a.pdf

8. Processing Multiple Buffers in Parallel to Increase Performance on Intel Architecture Processors White Paper
   http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf

Authors

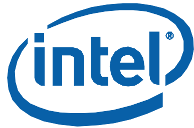
Adrian Hoban is a Software Architect in the Intel Communications and Storage Infrastructure Group.

Pierre Laurent, **Ian Betts**, and **Maryam Tahhan** are Software Engineers in the Intel Communications and Storage Infrastructure Group



Acronyms

ABI	Application Binary Interface
AES	Advanced Encryption Standard
API	Application Programming Interface
bio	Block I/O
CPU	Central Processing Unit
DM	Device Mapper
GCC	GNU Compiler Collection
ICC	Intel C/C++ Compiler
LKCF	Linux Kernel Crypto Framework
LVM	Logical Volume Manager
NI	New Instruction
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
SIMD	Single Instruction, Multiple Data
SSD	Solid State Drive
VM	Virtual Machine
XMM	128-bit data register
XTS	XOR-Encrypt-XOR with Cipher Text Stealing



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site (http://www.intel.com/).

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, Flexpipe, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel CoFluent, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel Xeon Phi, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Pentium, Pentium Inside, Puma, skoool, the skoool logo, SMARTi, Sound Mark, Stay With It, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

Copyright © 2013, Intel Corporation. All rights reserved.§

§