Indian Journal of Science and Technology, *Vol 7(11), 1813–1819, November 2014*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Implementation of a Parallel XTS Encryption Mode of Operation

Mohammad Ahmed Alomari[1*], Khairulmizam Samsudin[1] and Abdul Rahman Ramli[2]

[1]Computer Systems Research Group, Faculty of Engineering, Universiti Putra Malaysia, 43400 Serdang, Malaysia, m.alomari@ieee.org, kmbs@eng.upm.edu.my
[2]Intelligent Systems and Robotics Lab, Universiti Putra Malaysia, 43400 Serdang, Malaysia, arr@eng.upm.edu.my

Abstract

Security of data stored inside storage devices is becoming one of the main issues in computer security now. It is known that the most efficient techniques to protect storage devices are using cryptography. Developing newer and more secure encryption algorithms and modes of operation might be critically important to protect these devices since conventional disk encryption algorithms, such as CBC mode, have shown serious security flaws. In this paper, the IEEE XTS encryption mode of operation for storage encryption (P1619 standard) has been implemented using parallel design. A performance comparison between the sequential and parallel algorithms of XTS mode is presented here. Parallel overheads that prevented from achieving perfect linear speedup are measured and minimized. The parallel XTS algorithm has shown a speedup of 1.80, with 90% efficiency, faster than the sequential algorithm. In these simulations, AES is used as encryption algorithm with 256-bit encryption key.

Keywords: XTS Mode, Disk Encryption, Encryption Modes, Parallel Processing

1. Introduction

Data security is an essential part of computer security whether it is data in transit (transmitted data through networks) or data at rest (stored data in storage devices). The importance of securing storage devices has grown rapidly in the recent years. These devices, ranging from portable storage devices (PSDs) to personal digital assistances (PDAs) and laptops, are subjected to theft or loss due to their small size. Disclosure of data by unauthorized people may lead to huge organizational loss in both governmental and private sectors. An efficient way to secure storage devices is through encryption techniques. Cryptography is known to be a mathematically intensive operation which may affect the speed of any system. The performance and security of cryptography depends mainly on encryption algorithm and mode of operation used during encryption process. An effective way to enhance the performance of crypto operations is through parallelism.

In an attempt to introduce more secure standards for encryption, the advanced encryption standard (AES) has been chosen as a secure encryption algorithm by National Institute of Science and Technology (NIST)[1]. Additionally, IEEE institute has launched SISWG (Security In Storage Working Group) task force which developed a standard called P1619 standard[2]. This standard has introduced an encryption mode of operation named XTS (XEX encryption mode with tweak and ciphertext stealing) so that it can be used as alternative to the current insecure modes of

*Author for correspondence

operations such as CBC and ECB. XTS mode is developed to be used for encryption of storage devices, specifically disk drives.

An important feature in XTS mode is that it has a complete support to parallelism in its structure, which is a feature not available in the current storage encryption modes (such as CBC)[3]. On the other hand, XTS mode has been criticized from different perspectives which introduced a great need for further investigation and experiments on this new mode. One important point against XTS mode is that it is slower in performance than CBC mode due to slightly increase in its complexity to ensure security[4].

This paper implements the XTS encryption mode of operation in a parallel design which efficiently enhances the performance of the mode. The performance of both sequential and proposed parallel XTS algorithms has been compared. Additionally, the parallel speedup is measured including the parallel overheads that affect the maximum speedup. A system with a dual-core processor is used as a parallel environment to simulate the experiments. To enable parallel operations, OpenMP specification has been used as an API interface for communication between processors. Overheads in parallel XTS algorithms are measured, using OmpP[5] profiling tool, and then minimized to improve speedup.

This simulation work has been performed in a Linux environment using a hardware oriented tools such as C language, GCC compiler, and OpenMP API which may allow the simulations code to be implemented in hardware such as a disk controller or FPGA.

The rest of this paper is structured as follows. In section 2, details of XTS mode structure will be explored. Section 3 will introduce OpenMP specification as a parallel API. The details of implementations in this work will be explained in section 4. Section 5 provides analysis of the parallel overheads that prevent from achieving the maximum speedup. Finally, the results and conclusion are presented in section 6 and 7 respectively.

2. XTS Encryption Mode

For several decades, the most common mode of operation used for disk encryption was cbc mode, and it is still continued to be used until now. The inherent features in cbc mode have driven the cryptography community to search for more secure and higher performance alternatives. IEEE was one of the leading institutes to launch the project p1619 siswg in order to find alternatives for cbc, focusing on the protection of data in storage devices. This project is mainly targeted toward narrow-block (typically 128-bit) encryption.

In its early drafts, SISWG group has proposed LRW mode[6] as the most promising mode in order to standardize it for narrow-block storage encryption. Due to security flaws appeared in LRW mode, this group has moved to develop a new mode of operation called XTS. In Dec 2007, SISWG group has announced the XTS-AES P1619 (XTS mode with AES cipher) as a standard for data protection on narrow-block storage devices. XTS-AES was approved by NIST in 2010 as a mode of operation under FIPS 140-2 [7, 8]. This standard provides an architecture for cryptographic protection of data on block-oriented storage devices[2].

XTS can operate in parallel, allowing scalability in today's environment. This is not true for other modes of operation like CBC, OFB and CFB. In today's world of multiple core processors and low cost programmable hardware (such as FPGA), it is increasingly important that a designer be able to increase performance by instantiating multiple instances of an encryption primitive instead of increasing the clock rate of an existing encryption primitive[9]. XTS can be efficiently implemented in hardware with a slightly more complexity as compared to CBC[10-12].

Finally, XTS is known to be slower in speed than other modes such as CBC due to its more complex structure (such as two XOR operations, Galois multiplication, and two keys). This complexity and slow speed might prevent XTS to be widely adopted when it is compared to other conventional modes.

2.1 XTS Structure

XTS is a tweakable mode that is based on the Rogaway XEX[13] mode of operation with CipherText Stealing (CTS) feature. It is slower than CBC due to additionally more mathematical operations (such as XORing and GF multiplication). This mode XORs the plaintext before and after encryption process (Xor Encrypt Xor) with the tweak value as shown in Figure 1. It uses two keys instead of one key as compared to XEX mode. These two keys are combined to construct the XTS key (i.e. Key = Key1 | Key2).

The main encryption algorithm to be used with XTS mode is AES algorithm. As explained in the P1619 standard[2], XTS is a tweakable block cipher that acts on data units of 128 bits or more and uses the AES block cipher as

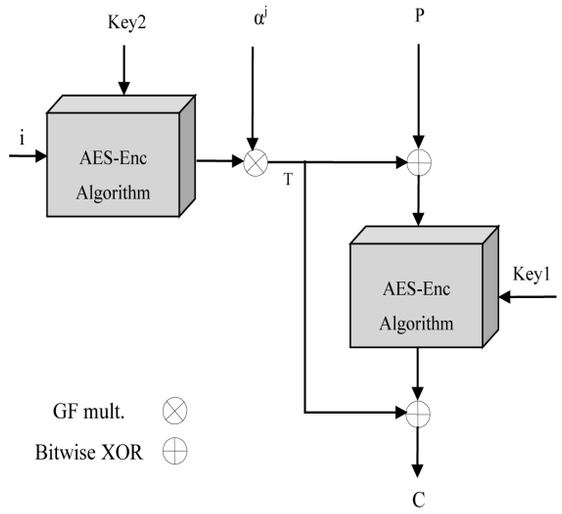

Figure 1. XTS-AES encryption process.

a subroutine. The key material for XTS consists of a data encryption key (used by the AES block cipher) as well as a tweak key that is used to incorporate the logical position (for example, inside the disk) of the data block into the encryption operation. As shown in Figure 1, the XTS-AES encryption procedure for a single 128-bit block can be represented by the following equation[2]:

$C \leftarrow XTS\text{-}AES\text{-}Enc(Key, P, i, j)$

Where:

Key The 256 or 512 bit XTS-AES key
P Block of 128-bits of plaintext
i The value of the 128-bit tweak
j Sequential no. of the 128-bit block inside the data unit
C Block of 128-bits of ciphertext

Figure 1 shows the encryption process of a single block of data. As shown in figure, the value of the 128-bit tweak *i* is encrypted using AES algorithm and the encryption key *Key2*. The result is multiplied (in Galois Field multiplication) with the *j-th* power of α (α^j) to produce *T* where α is a primitive element of GF. The plaintext *P* is encrypted using AES algorithm and encryption key *Key1*. Note that the plaintext *P* is XORed with the value *T* before and after this encryption process so that it can produce the final ciphertext *C*.

2.2 CipherText Stealing

Ciphertext Stealing (CTS) is a method used in modes of operations to deal with messages which can not be divided into a multiple of the block size (e.g. 128 bit for AES). The benefit of CTS is to prevent any expansion of the ciphertext, at a cost of slightly increase in complexity. For XTS, this is an important feature since it makes the output data (ciphertext) of this mode to be of same size as input data (plaintext) which is an important constraint for disk encryption[14].

Ciphertext Stealing in XTS is accomplished by processing the last two blocks of a message in a different way. If a message P with m number of blocks is to be encrypted and the last block is incomplete, then the last two blocks P_{m-1} and P_m will be encrypted and transferred in a reordered sequence. The encryption of the last two blocks is accomplished by padding the last block P_m (which is possibly incomplete) with the high order bits from the second last ciphertext block (ciphertext of P_{m-1}). By stealing these high order bits, the last partial block P_m will be completed and encrypted normally. Then the second last ciphertext block (ciphertext of P_{m-1}) is truncated to the length of the final plaintext block P_m. Finally, the ciphertext of the last two blocks will be reordered and transferred[7].

3. OpenMP Specification

OpenMP stands for Open Multi-Processing and can be defined as a multi-platform application programming interface (API) that is designed to be used for shared-memory parallel programming. It is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from desktops to supercomputers[15].

The OpenMP API uses the fork-join model of parallel execution which is based mainly on multithreading technology. In this model, an OpenMP program begins as a single thread of execution called the initial thread (or master thread). The initial thread executes sequentially until a parallel code construct is met. When initial thread encounters a parallel code, the thread creates a team of threads and becomes itself the master of the new team, as shown in Figure 2. All members of the new team execute the code inside the parallel region. Only the master thread continues execution of user code beyond the end of the parallel region. Any number of parallel regions can be specified in a single program[15].

Generally, parallel systems need to provide proper profiling tools to determine where the program spends most of its execution time and what resources are being

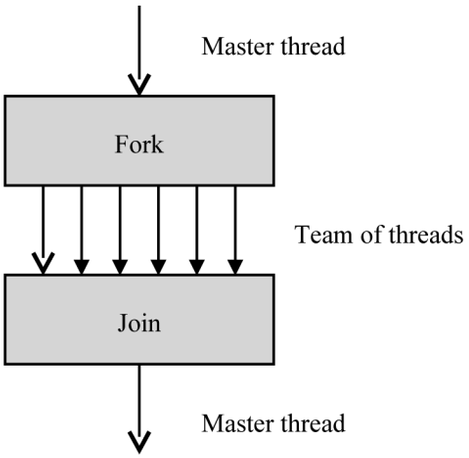

Figure 2. OpenMP Fork-Join Execution modelordering.

used. Profiling tools are usually based on modules to record performance-related information and to examine the results.

OmpP is a profiling tool for OpenMP applications designed for Unix-like systems. This tool is based on source code instrumentation by Opari[16]. OmpP determines the execution time and number of counts for all OpenMP constructs, such as parallel regions and critical sections that are available in executed application. OmpP consists of a monitoring library that is linked to an OpenMP application. Upon termination of the target application, OmpP writes a performance profiling report to a file that contains timing and counting data of program execution[17].

Most of the profiling tools collect performance data based on two approaches: sampling and code instrumentation. In this work, OmpP profiling tool has been used to debug the parallel code and analyze threads behavior during execution. This profiling tool implements the code instrumentation approach to collect data from executed code[5, 18].

4. Implementation Details

4.1 Implementation Environment

To parallelize XTS, a 2.4GHz Intel® dual-core system with 2GB of RAM has been used to implement the proposed parallel algorithm. The crypto part of this work has been implemented using LibTomCrypt encryption library which contains the AES encryption algorithm (128-bit block size and 256-bit key) and XTS encryption mode of operation. As a parallel API interface, OpenMP specification has been used to enable parallel operations and coordinate processors work. OmpP profiling tool is used to measure different parallel overheads so that the time-consuming regions can be minimized.

The simulations of both sequential and parallel XTS algorithms have been conducted using a group of six data samples with different sizes (128KB, 512KB, 1MB, 4MB, 8MB, and 16MB). Linux operating system is used as a platform for simulations. The presented values in results are the average of 10 repeated runs of encrypting each data sample. Execution time is measured in milliseconds accuracy using Linux time command[19].

4.2 Parallelization Process

The parallelization of XTS mode depends mainly on an important feature of this mode in which each block of data can be encrypted independently without any relation to other blocks. This feature allows the encrypted data to be divided into different portions in such a way that two successive blocks of data can be processed concurrently in two different processing units.

Dividing data into portions, where each portion is associated to a processor, is a parallelization strategy called data parallelism since the partition of the data is at the heart of this approach. This approach has been used in this work to parallelize XTS encryption mode. On the other hand, task parallelism strategy is based on the dividing of program code (instead of data) where each thread executes a portion of code. Since the parallel XTS-AES algorithm, in this work, will divide the data blocks into two almost equal parts, the expected performance speedup should be twice of the sequential algorithm. Actually, performance is subjected to different parallel overhead factors which may prevent from achieving perfect linear speedup.

5. Parallel Overhead Analysis

The parallel overheads in OmpP are divided into four categories: synchronization, imbalance, limited parallelism, and management overheads where each of them is caused by specific OpenMP constructs or pragmas. Synchronization overhead may occur due to threads waiting for each other in barriers or critical sections, whereas imbalance overhead is caused by improper distribution of work-load among threads. Forcing some parts of code to be executed only by a specific thread may cause limited parallelism overhead. Finally, a management overhead

might occur due to the time taken to repeatedly create, start, and stop threads[17].

In this work, to achieve a maximum performance speedup with the parallel XTS algorithm, a detailed overhead analysis has been performed using *OmpP* profiling tool. This analysis shows that a management overhead, caused by multiple startup and shutdown (fork-join) of threads, severely affects the execution time. This management overhead occurs due to the repeated creation and destruction of threads during each read of data. Imbalance overhead has also been reported in the analysis but with less severity which made this work to concentrate on minimizing the management overhead.

Figure 3 illustrates the parallel XTS algorithm when the parallel overheads severely affect the speedup. Imbalance and management overheads have limited the performance of XTS parallel algorithm with average percentages of 12% and 33% respectively. This shows that a total of 45% of the parallel algorithm execution time has been consumed in processing extra overheads rather than executing the intended task.

After analyzing and minimizing overheads in the parallel XTS algorithm, the management overhead has been reduced through eliminating the frequent startup/shutdown of threads. On the other hand, the imbalance overhead, which is caused by improper distribution of encryption load between threads, still remains as illustrated in Figure 4. The final average of management and imbalance overheads are 0.42% and 12% respectively. The total overhead of the improved XTS algorithm has dropped from 45% to 12% which enhanced the speedup significantly.

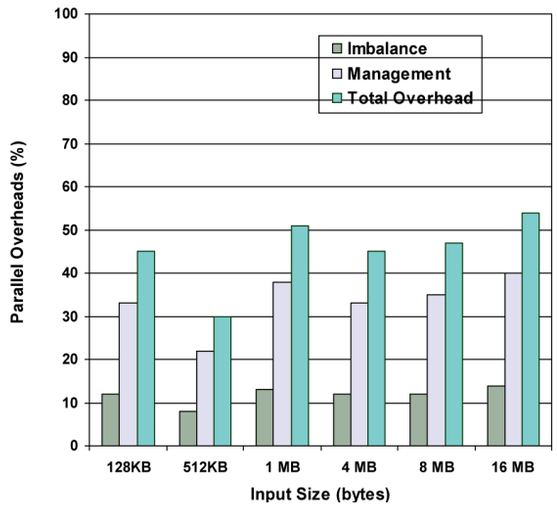

Figure 3. Parallel overheads with respect to XTS algorithm execution time.

6. Results Analysis

The performance of the parallel XTS-AES algorithm has been compared to the sequential algorithm as illustrated in Figure 5. For proper comparisons, both algorithms have been implemented using the same encryption algorithm (i.e. AES) and same data set (which range from 128 KB to 16 MB). Encryption time is measured here in milliseconds. As expected, Figure 5 shows that the XTS

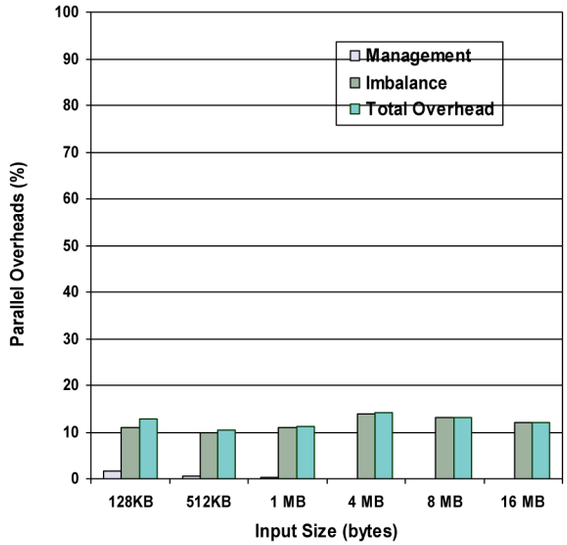

Figure 4. Parallel overheads with respect to improved execution time of XTS algorithm.

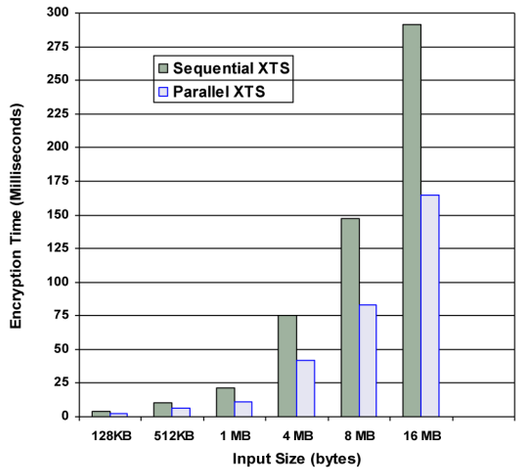

Figure 5. Performance Comparison of Sequential and Parallel XTS-AES Algorithms.

parallel algorithm, using two cores, demonstrates much higher performance than the sequential one while preserving XTS properties and security aspects.

Table 1 presents the encryption time for both XTS implementations. The sequential algorithm takes 291 milliseconds to encrypt 16MB of data, while parallel algorithm takes only 165 milliseconds. Hence, a parallel speedup of 1.76 and efficiency of 88% has been achieved. When calculating the average performance of parallel algorithm using all data samples, it achieves a parallel speedup of 1.80 with an efficiency of 90%.

7. XTS with Other Encryption Algorithms

For the completeness of this study, parallel XTS algorithm has been simulated using encryption algorithms other than AES. In this section, the parallel XTS is evaluated using Twofish and RC6 encryption algorithms. Figure 6 and 7 present the performance of both XTS-Twofish and XTS-RC6 algorithms respectively. Each of these algorithms is compared to its sequential algorithm to depict the improvement achieved in each case.

Table 1. Encryption Time (In Milliseconds) of XTS Algorithms

Data Size	128 K	512 K	1 MB	4 MB	8 MB	16 MB
Sequential XTS	004	010	020	075	147	291
Parallel XTS	002	006	011	042	083	165

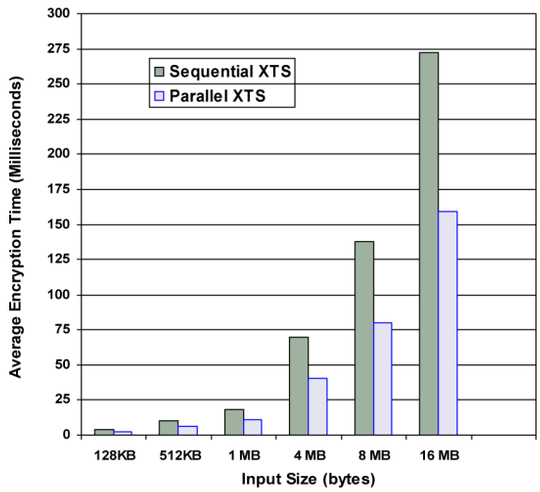

Figure 6. Performance Comparison of Sequential and Parallel XTS-Twofish Algorithms.

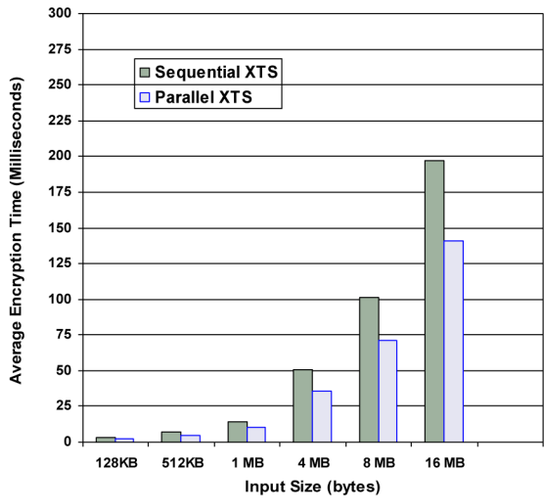

Figure 7. Performance Comparison of Sequential and Parallel XTS-RC6 Algorithms.

XTS-Twofish exhibits a higher speedup which is almost similar to XTS-AES algorithm. When calculating the average performance of the parallel XTS-Twofish algorithm, it achieves a speedup of 1.75 and an efficiency of 88%. On the other hand, although sequential XTS-RC6 algorithm provides better performance, the parallel XTS-RC6 produces low speedup as compared to other parallel XTS algorithms. The parallel XTS-RC6 achieves a speedup of 1.42 only with efficiency of 71%. This indicates that parallelizing XTS mode is less efficient using RC6 encryption algorithm, when compared to AES and Twofish. Although it produced low speedup, the parallel XTS-RC6 algorithm is still the fastest(less execution time) among all the evaluated algorithms as Figure 8 illustrates.

8. Conclusion and Future Work

In this paper, the XTS-AES encryption mode of operation has been successfully implemented in a parallel design while preserving its security aspects. The performance of parallel XTS algorithm is measured and compared to sequential algorithm which shows an efficient speedup of 1.8 due to the use of parallelism. Additionally, the parallel XTS is also simulated with other encryption algorithms as Twofish and RC6.

The results of this work demonstrate that it is highly practical to parallelize XTS-AES which will make this algorithm more suitable for storage encryption,

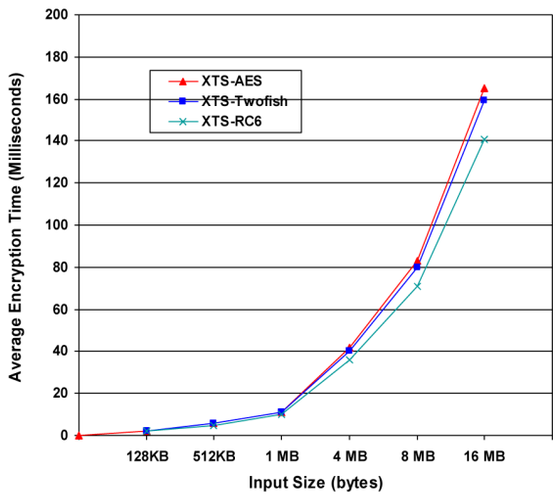

Figure 8. Performance Comparison of Parallel XTS Using Different Encryption Algorithms.

specifically disk encryption. With the massive data encryption operations of storage devices, parallelizing XTS-AES is an efficient way to enable the built-in disk encryption without affecting the overall performance.

Additionally, parallelizing XTS- AES can be greatly promising in the near future with the advent of GPU (graphics processing unit) computing, especially with the devices that have limited processing resources such as mobile devices. GPU computing allows the transfer of massively parallel encryption operations to be processed inside the GPU of a mobile device instead of the CPU which frees the CPU processor for user's needs and maintain a proper performance of the device.

9. References

1. NIST. FIPS-197, Announcing the ADVANCED ENCRYPTION STANDARD (AES). from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf 2001
2. IEEE. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. c1-32, 2008.
3. Fruhwirth, C. New Methods in Hard Disk Encryption. Institute for Computer Languages Theory and Logic Group, Vienna University of Technology, 2005.
4. Follow-up comments on NIST's consideration of XTS-AES (Draft 3). Retrieved August 01, 2011., from https:// www.siswg.net/index2.php?option=com_docman&task=doc_view&gid=169&Itemid=41
5. Furlinger, K., & Gerndt, M. ompP: A Profiling Tool for OpenMP. Lecture Notes in Computer Science, 2008; 4315, 15-23.
6. Liskov, M., Rivest, R., & Wagner, D. Tweakable block ciphers. Advances in Cryptology – CRYPTO '02, 2442 of Lecture Notes in Computer Science, 2002; 31-46.
7. Ball, M. V., Guyot, C., Hughes, J. P., Martin, L., & Noll, L. C. The XTS-AES disk encryption algorithm and the security of ciphertext stealing. Cryptologia 2012; 36(1) 70-79.
8. Dworkin, M. Recommendation for Block Cipher Mode of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, NIST Special Publication 2010; 800-38E.
9. Ball, M. NIST's Consideration of XTS-AES as standardized by IEEE Std 1619-2007. , 2008; from http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/XTS/XTS_comments-Ball.pdf
10. Storage Solutions by Helion. Accessed December 2, 2012. from http://www.heliontech.com/storage.htm
11. Ball, M. V. Follow-up to NIST's Consideration of XTS-AES as standardized by IEEE Std 1619-2007 (Draft 2). IEEE SISWG Retrieved March 13, 2011., 2009; from https:// www.siswg.net/index2.php?option=com_docman&task=doc_view&gid=166&Itemid=41
12. Ahmed, S., Samsudin, K., Ramli, A. R., & Rokhani, F. Z. An Effective Storage Encryption Solution. Indian Journal of Science and Technology, 2013; 6(4), 4384-4389.
13. Rogaway, P. Efficient Instantiations of Tweakable Block ciphers and Refinements to Modes OCB and PMAC. Advances in Cryptology – Asiacrypt, 3329 of Lecture Notes in Computer Science, 2004;16–31.
14. El-Fotouh, M. A., & Diepold, K. (2008). A New Narrow Block Mode of Operations for Disk Encryption. Paper presented at the The Fourth International Conference on Information Assurance and Security.
15. OpenMP. (2005). OpenMP Application Program Interface documentations. from http://www.openmp.org/mp-documents/spec25.pdf
16. Mohr, B., Malony, A. D., Shende, S., & Wolf, F. (2001). Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In Proceedings of the Third Workshop on OpenMP (EWOMP'01).
17. Furlinger, K., & Gerndt, M. Analyzing Overheads and Scalability Characteristics of OpenMP Applications. Lecture Notes in Computer Science, 2007; 4395, 39-51.
18. Chapman, B., Jost, G., & Pas, R. V. D. (2007). Using OpenMP: portable shared memory parallel programming: MIT press.
19. Andresen, R. Monitoring Linux with native tools. In the 30th Annual International Conference of the Computer Measurement Group, Inc., 2004;1, 345-354.